

3-е издание

Джеффри Рихтер



CLR via C#

**ПРОГРАММИРОВАНИЕ НА ПЛАТФОРМЕ
MICROSOFT .NET FRAMEWORK 4.0
НА ЯЗЫКЕ C#**

Microsoft®

ПИТЕР®



Jeffrey Richter

CLR via C#, Third Edition

Microsoft®

3-е издание

Джеффри Рихтер

CLR via C#

**ПРОГРАММИРОВАНИЕ НА ПЛАТФОРМЕ
MICROSOFT .NET FRAMEWORK 4.0
НА ЯЗЫКЕ C#**



**Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск**

2012

ББК 32.973.2-018.2

УДК 004.45

P55

Рихтер Дж.
P55 CLR via C#. Программирование на платформе Microsoft .NET Framework 4.0 на языке C#. 3-е изд. — СПб.: Питер, 2012. — 928 с.: ил.

ISBN 978-5-459-00297-3

Эта книга, выходящая в третьем издании и уже ставшая классическим учебником по программированию, подробно описывает внутреннее устройство и функционирование общезыковой исполняющей среды (CLR) Microsoft .NET Framework версии 4.0. Написанная признанным экспертом в области программирования Джеффри Рихтером, много лет являющимся консультантом команды разработчиков .NET Framework компании Microsoft, книга научит вас создавать по-настоящему надежные приложения любого вида, в том числе с использованием Microsoft Silverlight, ASP.NET, Windows Presentation Foundation и т. д.

Третье издание полностью обновлено в соответствии со спецификацией платформы .NET Framework 4.0 и принципами многоядерного программирования.

ББК 32.973.2-018.2

УДК 004.45

Права на издание получены по соглашению с Microsoft Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0735627048 англ.
ISBN 978-5-459-00297-3

© Microsoft Press, 2010

© Перевод на русский язык ООО Издательство «Питер», 2012

© Издание на русском языке, оформление ООО Издательство «Питер», 2012

Краткое оглавление

Предисловие	12
Введение	14
Часть I. Основы CLR	19
Глава 1. Модель выполнения кода в среде CLR	20
Глава 2. Компоновка, упаковка, развертывание и администрирование приложений и типов	34
Глава 3. Совместно используемые сборки и сборки со строгим именем	71
Часть II. Проектирование типов	99
Глава 4. Основы типов	100
Глава 5. Примитивные, ссылочные и значимые типы	121
Глава 6. Основные сведения о членах и типах	165
Глава 7. Константы и поля	191
Глава 8. Методы	196
Глава 9. Параметры	230
Глава 10. Свойства	248
Глава 11. События	272
Глава 12. Обобщения	289
Глава 13. Интерфейсы	322
Часть III. Основные типы данных	343
Глава 14. Символы, строки и обработка текста	344
Глава 15. Перечислимые типы и битовые флаги	393
Глава 16. Массивы	406
Глава 17. Делегаты	426
Глава 18. Настраиваемые атрибуты	458
Глава 19. Null-совместимые значимые типы	481
Часть IV. Ключевые механизмы	491
Глава 20. Исключения и управление состоянием	492
Глава 21. Автоматическое управление памятью (сборка мусора)	552
Глава 22. Хостинг CLR и домены приложений	633
Глава 23. Загрузка сборок и отражение	665
Глава 24. Сериализация	709
Часть V. Многопоточность	743
Глава 25. Потoki исполнения	744
Глава 26. Асинхронные вычислительные операции	771
Глава 27. Асинхронные операции ввода-вывода	813
Глава 28. Простейшие конструкции синхронизации потоков	852
Глава 29. Гибридные конструкции синхронизации потоков	889

Оглавление

Предисловие	12
Введение	14
Для кого эта книга	15
Посвящение	16
Благодарности	16
Поддержка	17
Мы хотим вас услышать	17
Часть I. Основы CLR	19
Глава 1. Модель выполнения кода в среде CLR	20
Компиляция исходного кода в управляемые модули	20
Объединение управляемых модулей в сборку	24
Загрузка CLR	25
Исполнение кода сборки	29
Глава 2. Компоновка, упаковка, развертывание и администрирование приложений и типов	34
Задачи развертывания в .NET Framework	34
Компоновка типов в модуль	36
Несколько слов о метаданных	40
Объединение модулей для создания сборки	47
Ресурсы со сведениями о версии сборки	58
Региональные стандарты	63
Развертывание простых приложений (закрытое развертывание сборок)	65
Простое средство администрирования (конфигурационный файл)	67
Глава 3. Совместно используемые сборки и сборки со строгим именем	71
Два вида сборок — два вида развертывания	72
Назначение сборке строгого имени	74
Глобальный кэш сборок	79
Компоновка сборки, ссылающейся на сборку со строгим именем	82
Устойчивость сборок со строгими именами к несанкционированной модификации	83
Отложенное подписание	85
Закрытое развертывание сборок со строгими именами	88
Как исполняющая среда разрешает ссылки на типы	89
Дополнительные административные средства (конфигурационные файлы)	93
Часть II. Проектирование типов	99
Глава 4. Основы типов	100
Все типы — производные от System.Object	100
Приведение типов	102

Пространства имен и сборки	106
Как разные компоненты взаимодействуют во время выполнения	111
Глава 5. Прimitives, ссылочные и значимые типы	121
Прimitives типы в языках программирования	121
Ссылочные и значимые типы	129
Упаковка и распаковка значимых типов	136
Хэш-коды объектов	155
Прimitives тип данных dynamic	158
Глава 6. Основные сведения о членах и типах	165
Члены типа	165
Видимость типа	168
Доступ к членам	170
Статические классы	172
Частичные классы, структуры и интерфейсы	174
Компоненты, полиморфизм и версии	175
Глава 7. Константы и поля	191
Константы	191
Поля	193
Глава 8. Методы	196
Конструкторы экземпляров и классы (ссылочные типы)	196
Конструкторы экземпляров и структуры (значимые типы)	200
Конструкторы типов	204
Методы перегруженных операторов	211
Методы операторов преобразования	215
Методы расширения	219
Частичные методы	225
Глава 9. Параметры	230
Необязательные и именованные параметры	230
Неявно типизированные локальные переменные	234
Передача параметров в метод по ссылке	236
Передача в метод переменного количества аргументов	242
Типы параметров и возвращаемых значений	245
Константность	247
Глава 10. Свойства	248
Свойства без параметров	248
Свойства с параметрами	264
Производительность при вызове методов-аксессоров свойств	270
Доступность аксессоров свойств	270
Обобщенные методы-аксессоры свойств	271
Глава 11. События	272
Разработка типа, поддерживающего событие	273
Как реализуются события	279
Создание типа, отслеживающего событие	282
Явное управление регистрацией событий	284

Глава 12. Обобщения	289
Обобщения в библиотеке FCL	294
Библиотека Power Collections производства Wintellect	295
Инфраструктура обобщений	296
Обобщенные интерфейсы	303
Обобщенные делегаты	304
Обобщенные методы	308
Обобщения и другие члены	311
Верификация и ограничения	311
Глава 13. Интерфейсы	322
Наследование в классах и интерфейсах	322
Определение интерфейсов	323
Наследование интерфейсов	324
Подробнее о вызовах интерфейсных методов	327
Явные и неявные реализации интерфейсных методов (что происходит за кулисами)	329
Обобщенные интерфейсы	331
Обобщения и ограничения интерфейса	333
Реализация нескольких интерфейсов с одинаковыми сигнатурами и именами методов	335
Совершенствование контроля типов за счет явной реализации интерфейсных методов	336
Опасности явной реализации интерфейсных методов	338
Дилемма разработчика: базовый класс или интерфейс?	341
Часть III. Основные типы данных	343
Глава 14. Символы, строки и обработка текста	344
Символы	344
Тип System.String	347
Эффективное создание строк	364
Получение строкового представления объекта	368
Получение объекта посредством разбора строки	378
Кодировки: преобразования между символами и байтами	381
Защищенные строки	389
Глава 15. Перечислимые типы и битовые флаги	393
Перечислимые типы	393
Битовые флаги	399
Добавление методов к перечислимым типам	404
Глава 16. Массивы	406
Инициализация элементов массива	408
Приведение типов в массивах	411
Базовый класс System.Array	413
Реализация интерфейсов IEnumerable, ICollection и IList	414
Передача и возврат массивов	415
Массивы с ненулевой нижней границей	416
Производительность доступа к массиву	418
Небезопасный доступ к массивам и массивы фиксированного размера	423

Глава 17. Делегаты	426
Знакомство с делегатами	426
Обратный вызов статических методов	429
Обратный вызов экземплярных методов	430
Раскрытие тайны делегатов	431
Обратный вызов нескольких методов (цепочки делегатов)	436
Обобщенные делегаты	443
Упрощенный синтаксис для работы с делегатами	445
Делегаты и отражение	454
Глава 18. Настраиваемые атрибуты	458
Сфера применения настраиваемых атрибутов	458
Определение класса атрибутов	462
Конструктор атрибута и типы данных полей и свойств	466
Выявление настраиваемых атрибутов	467
Сравнение экземпляров атрибута	472
Выявление настраиваемых атрибутов без создания производных от класса Attribute объектов	475
Условные атрибуты	479
Глава 19. Null-совместимые значимые типы	481
Поддержка в C# null-совместимых значимых типов	483
Оператор объединения null-совместимых значений	486
Поддержка в CLR null-совместимых значимых типов	488
Часть IV. Ключевые механизмы	491
Глава 20. Исключения и управление состоянием	492
Определение «исключения»	492
Механика обработки исключений	494
Класс System.Exception	501
Классы исключений, определенные в FCL	505
Вбрасывание исключений	508
Создание классов исключений	509
Продуктивность вместо надежности	512
Приемы работы с исключениями	521
Необработанные исключения	529
Отладка исключений	534
Скорость обработки исключений	536
Области ограниченного выполнения	539
Контракты кода	543
Глава 21. Автоматическое управление памятью (сборка мусора)	552
Работа на платформе, поддерживающей сборку мусора	552
Алгоритм сборки мусора	556
Сборка мусора и отладка	560
Освобождение ресурсов при помощи механизма финализации	564
Финализация управляемых ресурсов	571
Мотивы вызова методов финализации	574

Детали механизма финализации	576
Эталон освобождения ресурсов: принудительная очистка объекта	579
Типы, реализующие эталон освобождения ресурсов	584
Инструкция using языка C#	588
Интересные аспекты зависимостей	590
Мониторинг и контроль времени жизни объектов	592
Воскрешение	605
Поколения	607
Другие возможности сборщика мусора для работы с машинными ресурсами	614
Прогнозирование успеха операции, требующей много памяти	619
Программное управление сборщиком мусора	621
Захват потока	625
Режимы сборки мусора	626
Большие объекты	630
Мониторинг сборки мусора	630
Глава 22. Хостинг CLR и домены приложений	633
Хостинг CLR	633
Домены приложений	636
Выгрузка доменов	652
Мониторинг доменов	653
Уведомление о первом управляемом исключении домена	655
Использование хостами доменов приложений	656
Нетривиальное управление хостингом	659
Глава 23. Загрузка сборок и отражение	665
Загрузка сборок	665
Использование отражения для создания динамически расширяемых приложений	671
Производительность отражения	672
Создание приложений с поддержкой подключаемых компонентов	681
Нахождение членов типа путем отражения	684
Глава 24. Сериализация	709
Краткое руководство по сериализации/десериализации	710
Сериализуемые типы	715
Управление сериализацией и десериализацией	717
Сериализация экземпляров типа	721
Управление сериализованными и десериализованными данными	723
Контексты потока ввода-вывода	730
Сериализация в другой тип и десериализация в другой объект	732
Суррогаты сериализации	736
Переопределение сборки и/или типа при десериализации объекта	740
Часть V. Многопоточность	743
Глава 25. Поток выполнения	744
Зачем потоки в Windows?	744
Ресурсоемкость потоков	745
Так дальше не пойдет!	750

Тенденции развития процессоров	752
Неравномерный доступ к памяти	754
CLR- и Windows-потоки	756
Потоки для асинхронных вычислительных операций	757
Мотивы использования потоков	759
Порядок исполнения и приоритеты потоков	762
Фоновые и активные потоки	768
Что дальше?	769
Глава 26. Асинхронные вычислительные операции	771
Пул потоков в CLR	771
Простые вычислительные операции	773
Контексты исполнения	774
Скоординированная отмена	776
Задания	781
Методы For, ForEach и Invoke класса Parallel	796
Встроенный язык параллельных запросов	800
Периодические вычислительные операции	804
Как пул управляет потоками	807
Строки кэша и ложное разделение	810
Глава 27. Асинхронные операции ввода-вывода	813
Операции ввода-вывода в Windows	813
Модель асинхронного программирования в CLR	818
Класс AsyncEnumerator	823
Модель асинхронного программирования и исключения	827
Потоковые модели приложений	828
Асинхронная реализация сервера	832
Модель асинхронного программирования и вычислительные операции	833
Анализ модели асинхронного программирования	835
Приоритеты запросов ввода-вывода	840
Преобразование объекта IAsyncResult в объект Task	843
Эталон асинхронного программирования на базе событий	844
Сводная информация по моделям программирования	850
Глава 28. Простейшие конструкции синхронизации потоков	852
Библиотеки классов и безопасность потоков	854
Простейшие конструкции пользовательского режима и режима ядра	856
Конструкции пользовательского режима	857
Конструкции режима ядра	876
Глава 29. Гибридные конструкции синхронизации потоков	889
Простое гибридное записание	889
Зацикливание, владение потоком и рекурсия	891
Различные гибридные конструкции	894
Записание с двойной проверкой	910
Эталон условной переменной	915
Сокращение времени записания при помощи коллекций	918
Классы коллекций для параллельной обработки потоков	923

Предисловие

Поначалу, когда Джефф попросил меня написать предисловие для своей книги, я была очень польщена. Я подумала, что он по-настоящему меня уважает. Дорогие мои, поверьте, это общее заблуждение наивных людей, на самом деле, здесь нечто совсем иное. Я была под номером 14 в его списке потенциальных авторов предисловия для своей книги. И лишь когда Джеффу стало очевидно, что другие кандидаты из этого списка (Билл Гейтс, Стив Балмер, Кэтрин Зета-Джонс и т. д.) не согласятся, он повел меня ужинать.

Тем не менее никто не сможет рассказать вам об этой книге больше, чем я. Я имею в виду, что Кэтрин Зета-Джонс, конечно, могла бы подробно рассказать вам о сервисе «Mobile Makeover»¹, но я-то знаю обо всех отражениях, исключениях и модификациях языка C#, потому что мы с Джеффом обсуждали это годами. Для нас это было обычным разговором за ужином! Другие люди говорят о погоде, а мы — о .NET. Даже Адэн, наш шестилетний сын, задает вопросы о книге Джеффа. Правда, в основном о том, начнут ли они, наконец, во что-нибудь играть, когда Джефф напишет свою книгу. Двухлетний Грант пока не говорит, но, возможно, первым его словом будет «сериализация».

На самом деле, если вы хотите знать, как все начиналось, то это происходило следующим образом. Приблизительно 10 лет назад Джефф посетил «Секретное Совещание» в Microsoft. Компания привлекла группу экспертов (а на самом деле, откуда у Джеффа этот титул? На эксперта в школе не учат) и представила новую технологию взамен COM. Поздно ночью, в постели (у нас принято обсуждать это в постели) он сказал, что COM померла. При этом Джефф был просто потрясен новой технологией. Практически втрескался в нее. В течение нескольких дней он ходил вокруг строения 42 в Редмондском кампусе Microsoft, надеясь узнать о .NET побольше. Похоже, его «роман» продолжается до сих пор, раз он даже решил написать об этом книгу.

Годами Джефф рассказывал мне о многопоточности. Ему по-настоящему нравится эта тема. Однажды, в Новом Орлеане, мы с ним пошли на двухчасовую прогулку, и всю дорогу он твердил мне о том, что у него достаточно материала для книги «Искусство многопоточности». Как я могу не понимать многопоточность в Windows?! Это буквально разрывало его сердце. Как выполняются потоки? Почему они создаются, если нет четкого плана их работы? Эти вопросы составляют основу мироощущения Джеффа, они стали смыслом его жизни. В конце концов, всех их он включил в эту книгу. Они все здесь, и поверьте мне, если вы хотите знать о многопоточности и потоках, никто не расскажет вам о них лучше и больше, чем Джефф. И все потраченные на это часы его жизни (их уже не вернуть) здесь, в вашем распоряжении. Пожалуйста, прочтите эту книгу. А потом пошлите Джеффу электронное письмо с рассказом о том, как эта книга изменила вашу жизнь. Если вы этого не сделаете, он станет еще одним трагическим персонажем, чья жизнь прошла бессмысленно и беспросветно. И он ухнет до смерти диетическим лимонадом.

¹ Кэтрин Зета-Джонс рекламирует этот сервис на американском телевидении. — *Примеч. перев.*

Эта редакция книги включает в себя новую главу о сериализации во время выполнения программы. Оказывается, это совсем не так просто, как залить молоком кукурузные хлопья на завтрак для детей. Когда я пыталась в этом разобраться, то за компьютерными разговорами перестала замечать окружающий мир. И хотя я так и не поняла, что это, об этом написано в книге, и вы можете об этом прочитать (запивая молоком).

Сейчас у меня одна надежда, что дорогой Джефф перестанет рассуждать о теории сборки мусора, а реально соберет мусор и вынесет его на помойку. Действительно, ну неужели это так сложно?

И последний довод, дорогие мои, — это самое главное и самое последнее произведение Джефффри Рихтера. Больше книг не будет. Конечно, мы говорим это каждый раз, когда он заканчивает очередную книгу, но в этот раз мы *действительно* так думаем. Итак, было 13 книг, но эта — лучшая и последняя. Быстрее ее покупайте, потому что тираж книги ограничен, и если его весь раскупят, то больше книг не будет. Как у QVC¹. Ну, а мы из виртуальной жизни вернемся в реальную и сможем обсудить действительно важные вещи, например, что же сегодня разбили наши детки и чья очередь менять подгузники.

Кристин Трейс (жена Джефффри)

24 ноября 2009 года



Обычный завтрак в семье Рихтеров

¹ Qvc.com — сайт, на котором распродают товары со скидкой за ограниченное время по принципу «кто не успел, тот опоздал». — *Примеч. перев.*

Введение

В октябре 1999 года люди из Microsoft впервые продемонстрировали мне платформу .NET Framework, *общезыковую среду выполнения* (Common Language Runtime, CLR) и язык программирования C#. Когда я впервые это увидел, то был очень сильно впечатлен и сразу понял, что процесс создания мною программного обеспечения претерпит существенные изменения. Меня попросили проконсультировать команду разработчиков, и я немедленно дал свое согласие. Поначалу я думал, что платформа .NET Framework является некоей прослойкой между *интерфейсами программирования приложений* (Application Program Interface, API) Win32 и *объектной моделью компонентов* (Component Object Model, COM). Но чем больше я ее изучал, тем больше различий находил. В некотором смысле это операционная система. У нее собственные менеджер виртуальной памяти, система безопасности, файловый загрузчик, механизм обработки ошибок, изолированная среда, в которой выполняются домены приложений, модель многопоточности и многое другое. В этой книге все эти темы освещаются таким образом, чтобы вы могли эффективно проектировать и реализовывать программное обеспечение на этой платформе.

Я провел много времени, разбираясь в понятиях многопоточности, одно-временном выполнении, параллелизме, синхронизации и т. д. Сегодня, когда многоядерные компьютеры получили широкое распространение, эти понятия приобретают все большую важность. Несколько лет назад я решил написать книгу, посвященную многопоточности. Однако один вопрос тянул за собой другой, и в результате книгу о многопоточности я так и не написал. Однако когда пришло время переработать эту книгу, я решил включить в нее сведения о многопоточности. Таким образом, эта книга затрагивает платформу .NET Framework, язык программирования C# и многопоточность (см. часть V).

Я писал текст первого издания этой книги в октябре 2009 года, и уже прошло 10 лет с тех пор, как я работаю с платформой .NET Framework и языком программирования C#. За эти 10 лет я создал разнообразные приложения и как консультант компании Microsoft внес значительный вклад в разработку платформы .NET Framework. В качестве сотрудника своей компании Wintellect (<http://Wintellect.com>) я работал со многими заказчиками, помогая им проектировать программное обеспечение, отлаживать и оптимизировать программы, решать их проблемы, связанные с .NET Framework. Весь этот опыт помог мне реально узнать, какие именно проблемы возникают у людей при работе с платформой .NET Framework и как эти проблемы решать. Я попытался сопроводить этими знаниями все темы, представленные в этой книге.

Для кого эта книга

Цель этой книги — объяснить, как разрабатывать приложения и многократно используемые классы для .NET Framework. В частности, это означает, что я собираюсь рассказать, как работает среда CLR, какие возможности она предоставляет разработчику. В этой книге также описываются различные классы *стандартной библиотеки классов* (Framework Class Library, FCL). Ни в одной книге невозможно описать FCL полностью — она содержит тысячи типов, и их число продолжает расти ударными темпами. Так что я остановлюсь на основных типах, с которыми должен быть знаком каждый разработчик. И хотя в этой книге не рассматриваются отдельно подсистема графического интерфейса пользователя Windows Forms, система для построения клиентских приложений Windows Presentation Foundation (WPF), платформа Silverlight, XML веб-сервисы, веб-формы и т. д., технологии, описанные в ней, применимы ко всем этим видам приложений.

В книге используются системы Microsoft Visual Studio 2010, .NET Framework 4.0 и компилятор C# версии 4.0. В компании Microsoft стараются обеспечивать максимальную обратную совместимость разрабатываемых программных продуктов, поэтому многое из того, что обсуждается в данной книге, применимо и к ранним версиям. Все примеры в книге написаны на языке C#, но так как в среде CLR можно использовать различные языки программирования, книга пригодится также и тем, кто пишет на других языках программирования.

ПРИМЕЧАНИЕ

Вы можете загрузить примеры из книги с сайта компании Wintellect (<http://Wintellect.com>). В разных разделах книги я описываю классы из моей собственной библиотеки Power Threading Library. Эта библиотека свободно распространяется и ее тоже можно скачать с сайта компании Wintellect.

На сегодняшний день компания Microsoft предлагает несколько версий среды CLR. Это серверная версия, которая выполняется на 32-разрядной системе Windows под архитектуру x86, а также на 64-разрядной системе Windows под архитектуры x64 и IA64. Еще предлагается версия под платформу Silverlight, которая реализована на основе программного кода серверной версии среды CLR. Следовательно, все, что описано в данной книге, применимо и к построению приложений под платформу Silverlight, за исключением некоторых отличий, касающихся того, как Silverlight запускает сборки. Компания Microsoft также предлагает облегченную версию .NET Framework для мобильных телефонов и других устройств с операционными системами Windows Mobile и Windows CE, которая называется .NET Compact Framework. Многие из того, что рассмотрено в данной книге, применимо для разработки приложений под .NET Compact Framework, но это не основное, на чем был сделан акцент.

13 декабря 2001 года Европейская ассоциация по стандартизации информационных и вычислительных систем ECMA International (<http://www.ecma-international.org/>) приняла в качестве стандарта язык C# и некоторые компоненты среды CLR и библиотеки FCL. Этот стандарт позволил сторонним компаниям разработать ECMA-совместимые версии этих технологий под архитектуру других процессоров и операционных систем. В действительности, Novell выпускает платформу Moonlight (<http://mono-project.com/Moonlight>), реализацию платформы Silverlight с открытым кодом (<http://Silverlight.net>), которая, прежде всего, ориентирована на Linux и другие операционные системы на основе UNIX/X11. Платформа Moonlight основана на ECMA-спецификациях. Львиная доля этой книги посвящена описанию этих стандартов, поэтому многие найдут эту книгу полезной при работе с реализациями приложений и библиотек, совместимых со спецификацией ECMA.

ПРИМЕЧАНИЕ

Я и мои редакторы усердно потрудились, чтобы дать вам наиболее точную, свежую, исчерпывающую, удобную, понятную и безошибочную информацию. Однако даже эта фантастическая команда не может предусмотреть все. Если вы обнаружите в этой книге ошибки (особенно ошибки в программном коде) или же у вас появится конструктивное предложение, то я буду вам очень признателен за сообщение об этом по моему адресу: JeffreyR@Wintellect.com.

Посвящение

Кристин. Словами не выразить то, что я думаю о нашей совместной жизни. Я с нежностью отношусь к нашей семье и ко всем нашим семейным радостям. Каждый день моей жизни наполнен любовью к тебе.

Адэн (6 лет) и Грант (2 года). Вы оба вдохновляли меня, учили меня играть и весело проводить время. Наблюдения за тем, как вы росли, были прекрасны и полезны. Я рад быть частью вашей жизни. Я люблю и ценю вас больше всех на свете.

Благодарности

Я бы не написал эту книгу без помощи и технической поддержки многих людей. В частности, я хотел бы поблагодарить мою семью. Очень сложно измерить время и усилия, которые требуются для написания хорошей книги, но я знаю точно, что без поддержки моей жены Кристин и двух сыновей, Адэна и Гранта, я никогда не смог бы эту книгу написать. Очень часто наши планы

совместного времяпровождения срывались из-за жесткого графика работы над книгой. Теперь, по завершении работы над ней, я с радостью предвкушаю радость общения с семьей.

В написании книги мне помогали фантастические люди. Кристоф Насарр (Christophe Nasarre) выполнил титанический труд по проверке моей работы и позаботился о точности и корректности формулировок. Он внес очень большой вклад в повышение качества книги. Как всегда, было очень приятно работать с редакторской командой Microsoft Press. Отдельную благодарность я хочу выразить Бену Райану (Ben Ryan), Вэлери Вулли (Valerie Woolley) и Дэвону Масгрейву (Devon Musgrave). Спасибо также Джин Финдли (Jean Findley) и Сью МакКланг (Sue McClung) за редактуру и помощь при выпуске книги.

Поддержка

Мы приложили максимум усилий, чтобы эта книга была как можно более точной. Все исправления и изменения будут добавлены в соответствующую статью Базы знаний (Microsoft Knowledge Base), доступную на сайте «Справка и поддержка Microsoft». Издательство Microsoft Press постоянно обновляет список исправлений и дополнений к своим книгам, которые доступны по адресу <http://microsoft.com/learning/support/books/>.

Если у вас все же возникнут вопросы, ответы на которые вы не сможете найти на предложенных сайтах или в статьях Базы знаний, то можете задать эти вопросы в письме, отправив его по адресу mspinput@microsoft.com.

Пожалуйста, обратите внимание, что по указанным адресам техническая поддержка программных продуктов не предоставляется.

Мы хотим вас услышать

Мы приветствуем любые отзывы на эту книгу. Пожалуйста, оставьте ваши комментарии и предложения в опросе, который находится по адресу <http://microsoft.com/learning/booksurvey>.

Ваше участие поможет Microsoft Press создавать книги, которые соответствуют вашим запросам и ожиданиям.

ПРИМЕЧАНИЕ

Мы надеемся, что вы оставите в опросе подробный отзыв. Если у вас есть вопросы о нашей издательской программе, вскоре выпускаемых книгах или же издательстве в целом, мы рады ответить на них в Твиттере: <http://twitter.com/MicrosoftPress>. Для вопросов относительно издания используйте только вышеуказанную электронную почту.

От издателя перевода

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства <http://www.piter.com>.

ЧАСТЬ I

Основы CLR

Глава 1.	Модель выполнения кода в среде CLR	20
Глава 2.	Компоновка, упаковка, развертывание и администрирование приложений и типов	34
Глава 3.	Совместно используемые сборки и сборки со строгим именем	71

Глава 1. Модель выполнения кода в среде CLR

Microsoft .NET Framework представляет новые концепции, технологии и термины. Цель этой главы — дать обзор архитектуры .NET Framework, познакомить с новыми технологиями этой платформы и определить термины, с которыми вы столкнетесь при работе с ней. Также в этой главе изложен процесс построения приложения или набора распространяемых компонентов (файлов), которые содержат типы (классы, структуры и т. п.), и затем объяснено, как выполняется приложение.

Компиляция исходного кода в управляемые модули

Итак, вы решили использовать .NET Framework как платформу разработки. Отлично! Ваш первый шаг — определить вид создаваемого приложения или компонента. Предположим, что этот вопрос уже решен, все спроектировано, спецификации написаны и все готово для начала разработки.

Теперь надо выбрать язык программирования. И это непростая задача — ведь у разных языков имеются разные возможности. Например, с одной стороны, «неуправляемый код» C/C++ дает доступ к системе на низком уровне. Вы вправе распоряжаться памятью по своему усмотрению, создавать потоки и т. д. Но с другой стороны, Visual Basic 6 позволяет очень быстро строить пользовательские интерфейсы и легко управлять СОМ-объектами и базами данных.

Название среды — *общезыковая среда выполнения* (Common Language Runtime, CLR) — говорит само за себя: это среда выполнения, которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду. Например, при обработке ошибок среда выполнения опирается на исключения, а значит, во всех языках программирования, использующих эту среду выполнения, можно получать сообщения об ошибках при помощи механизма исключений. Или, например, среда выполнения позволяет создавать поток, а значит, во всех языках программирования, использующих эту среду, могут создаваться потоки.

Фактически во время выполнения программы в среде CLR неизвестно, на каком языке программирования разработчик написал исходный код. А это значит, что можно выбрать любой язык программирования, который позволяет

проще всего решить данную задачу. Разрабатывать программное обеспечение можно на любом языке программирования, если используемый компилятор этого языка предназначен для CLR.

Так в чем же тогда преимущество одного языка программирования перед другим? Преимущество в компиляторах, во встроенном в них механизме контроля синтаксиса и анализа *корректного кода*. Компиляторы проверяют исходный код, убеждаются, что все написанное имеет некий смысл, и затем генерируют код, описывающий решение данной задачи. Разные языки программирования позволяют разрабатывать программное обеспечение, используя различный синтаксис. Не стоит недооценивать значение выбора синтаксиса языка программирования. Например, для математических или финансовых приложений выражение мысли программиста на языке APL может сохранить много дней работы по сравнению с применением в данной ситуации языка Perl.

Компания Microsoft разработала компиляторы для следующих языков программирования, используемых на этой платформе: C++/CLI, C# (произносится «си шарп»), Visual Basic, JScript, J# (компилятор языка Java) и ассемблер Intermediate Language (IL). Кроме Microsoft, еще несколько компаний и университетов создали компиляторы, предназначенные для среды выполнения CLR. Мне известны компиляторы для APL, Caml, COBOL, Eiffel, Forth, Fortran, Haskell, Lexico, LISP, LOGO, Lua, Mercury, ML, Mondrian, Oberon, Pascal, Perl, Php, Prolog, Python, RPG, Scheme, Smalltalk и Tcl/Tk.

Рисунок 1.1 иллюстрирует процесс компиляции файлов с исходным кодом. Как видно из рисунка, исходный код программы может быть написан на любом языке, поддерживающем среду выполнения CLR. Затем соответствующий компилятор проверяет синтаксис и анализирует исходный код программы. Вне зависимости от типа используемого компилятора результатом компиляции будет являться *управляемый модуль* (managed module) — стандартный *переносимый исполняемый* (portable executable, PE) файл 32-разрядной (PE32) или 64-разрядной Windows (PE32+), который требует для своего выполнения



Рис. 1.1. Компиляция исходного кода в управляемые модули

CLR. Кстати, *управляемые сборки* всегда используют преимущества функции безопасности «предотвращения выполнения данных» (DEP, Data Execution Prevention) и технологию ASLR (Address Space Layout Optimization), применение этих технологий повышает информационную безопасность программного обеспечения.

В табл. 1.1 описаны составные части управляемого модуля.

Таблица 1.1. Части управляемого модуля

Часть	Описание
Заголовок PE32 или PE32+	Стандартный заголовок PE-файла Windows, аналогичный заголовку Common Object File Format (COFF). Файл с заголовком в формате PE32 может выполняться в 32- и 64-разрядной версиях Windows, а с заголовком PE32+ — только в 64-разрядной. Заголовок показывает тип файла: GUI, CUI или DLL, он также имеет временную метку, показывающую, когда файл был собран. Для модулей, содержащих только IL-код, основной объем информации в заголовке PE32(+) игнорируется. В модулях, содержащих машинный код, этот заголовок содержит сведения о машинном коде
Заголовок CLR	Содержит информацию (интерпретируемую CLR и утилитами), которая превращает этот модуль в управляемый. Заголовок включает нужную версию CLR, некоторые флаги, метку метаданных MethodDef точки входа в управляемый модуль (метод Main), а также месторасположение/размер метаданных модуля, ресурсов, строгого имени, некоторых флагов и пр.
Метаданные	Каждый управляемый модуль содержит таблицы метаданных. Есть два основных вида таблиц — это таблицы, описывающие типы данных и члены, определенные в исходном коде, и таблицы, описывающие типы данных и члены, на которые имеются ссылки в исходном коде
Код Intermediate Language (IL)	Код, создаваемый компилятором при компиляции исходного кода. Впоследствии CLR компилирует IL в команды процессора

Компиляторы машинного кода производят код, ориентированный на конкретную процессорную архитектуру, например x86, x64 или IA64. В отличие от этого, все CLR-совместимые компиляторы генерируют IL-код. (Подробнее об IL-коде рассказано далее в этой главе.) IL-код иногда называют *управляемым* (managed code), потому что CLR управляет его выполнением.

Каждый компилятор, предназначенный для CLR, помимо генерации IL-кода, должен также создавать полные *метаданные* (metadata) для каждого управляемого модуля. Если выражаться кратко, то метаданные — это набор таблиц данных, описывающих то, что определено в модуле, например типы и их члены. В метаданных также есть таблицы, указывающие, на что ссылается управляемый модуль, например на импортируемые типы и их члены. Метаданные расширяют возможности таких старых технологий, как библиотеки COM-типов и файлы *языка описания интерфейсов* (Interface Definition Language, IDL).

Важно отметить, что метаданные CLR значительно более детальные. И, в отличие от библиотек COM-типов и IDL-файлов, они всегда связаны с файлом, содержащим IL-код. Фактически метаданные всегда встроены в тот же EXE-или DLL-файл, что и код, так что их нельзя разделить. По причине того, что компилятор генерирует метаданные и код одновременно и привязывает их к конечному управляемому модулю, возможность рассинхронизации метаданных и описываемого ими IL-кода исключена.

Метаданные имеют несколько применений. Перечислим некоторые из них.

- ❑ Метаданные устраняют необходимость в заголовочных и библиотечных файлах при компиляции, так как все сведения о типах/членах, на которые есть ссылки, содержатся в файле с реализующим их IL-кодом. Компиляторы могут читать метаданные прямо из управляемых модулей.
- ❑ Microsoft Visual Studio использует метаданные для облегчения написания кода. Ее функция IntelliSense анализирует метаданные и сообщает, какие методы, свойства, события и поля предпочтительны в данном случае и какие именно параметры требуются конкретным методам.
- ❑ В процессе верификации кода CLR использует метаданные, чтобы убедиться, что код совершает только «безопасные» операции. (Проверка кода обсуждается далее.)
- ❑ Метаданные позволяют сериализовать поля объекта, затем передать эти данные по сети на удаленный компьютер и там провести процесс десериализации, восстановив объект и его состояние на удаленном компьютере.
- ❑ Метаданные позволяют сборщику мусора отслеживать жизненный цикл объектов. При помощи метаданных сборщик мусора может определить тип объектов и узнать, какие именно поля в них ссылаются на другие объекты.

В главе 2 метаданные описаны подробнее.

Языки программирования C#, Visual Basic, JScript, J# и IL-ассемблер всегда создают модули, содержащие управляемый код (IL) и управляемые данные (данные, поддерживающие сборку мусора). Для выполнения любого управляемого модуля на машине конечного пользователя должна быть установлена CLR (в составе .NET Framework), так же как для выполнения приложений MFC или Visual Basic 6 должны быть установлена библиотека классов Microsoft Foundation Class (MFC) или динамически подключаемые библиотеки Visual Basic.

По умолчанию компилятор Microsoft C++ создает EXE- и DLL-файлы, которые содержат неуправляемый код и неуправляемые данные. Для их выполнения CLR не требуется. Однако если вызвать компилятор C++ с параметром /CLR в командной строке, он создаст управляемые модули, для работы которых необходимо установить CLR. Компилятор C++ стоит особняком среди всех упомянутых компиляторов производства Microsoft — он единственный позволяет

разработчикам писать как управляемый, так и неуправляемый код и встраивать его в единый модуль. Это также единственный компилятор Microsoft, позволяющий программистам определять в исходном коде как управляемые, так и неуправляемые типы данных. Это очень важное свойство, поскольку оно позволяет разработчикам обращаться к существующему неуправляемому коду на C/C++ из управляемого кода и постепенно, по мере необходимости, переходить на управляемые типы.

Объединение управляемых модулей в сборку

На самом деле среда CLR работает не с модулями, а со сборками. *Сборка* (assembly) — это абстрактное понятие, осознание которого поначалу может вызвать затруднения. Во-первых, это логическая группировка одного или нескольких управляемых модулей или файлов ресурсов. Во-вторых, это самая маленькая единица с точки зрения многократного использования, безопасности и управления версиями. Сборка может состоять из одного или нескольких файлов — все зависит от выбранных средств и компиляторов. В контексте среды CLR сборку можно назвать *компонентом*.

О сборке довольно подробно рассказано в главе 2, а здесь достаточно подчеркнуть, что эта концепция предлагает способ объединения группы файлов в единую сущность.

Рисунок 1.2 помогает понять суть сборки. На этом рисунке показано, что некоторые управляемые модули и файлы ресурсов (или данных) создаются при помощи инструментального средства. Это средство создает единственный файл PE32(+), который представляет логическую группировку файлов. Рассмотрим, что происходит в случае, когда файл PE32(+) содержит блок данных, называемый *манифестом* (manifest). Манифест — это просто один из наборов таблиц метаданных. Эти таблицы описывают файлы, которые формируют сборку, общедоступные экспортируемые типы, реализованные в файлах сборки, а также относящиеся к сборке файлы ресурсов или данных.

По умолчанию компиляторы сами выполняют работу по превращению созданного управляемого модуля в сборку, то есть компилятор C# создает управляемый модуль с манифестом, указывающим, что сборка состоит только из одного файла. Таким образом, в проектах, где есть только один управляемый модуль и нет файлов ресурсов (или файлов данных), сборка и является управляемым модулем, поэтому прилагать дополнительных усилий по компоновке приложения не нужно. В случае если необходимо сгруппировать несколько файлов в сборку, потребуются дополнительные инструменты (например, компоновщик сборок *AL.exe*) со своими параметрами командной строки. О них подробно рассказано в главе 2.

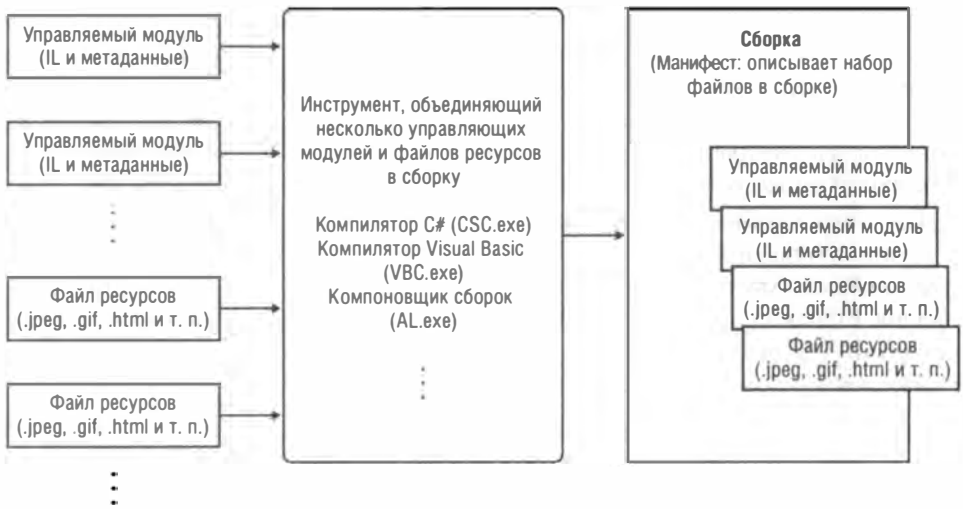


Рис. 1.2. Объединение управляемых модулей в сборку

Сборка позволяет разделить логическое и физическое представления компонента, поддерживающего многократное использование, безопасность и управление версиями. Разбиение программного кода и ресурсов на разные файлы полностью определяется желаниями разработчика. Например, редко используемые типы и ресурсы можно вынести в отдельные файлы сборки. Отдельные файлы могут загружаться по запросу из Интернета по мере необходимости в процессе выполнения программы. Если некоторые файлы не потребуются, то они не будут загружаться, что сохранит место на жестком диске и сократит время установки программы. Сборки позволяют разбить на части процесс развертывания файлов и в то же время рассматривать все файлы как одну коллекцию.

Модули сборки также содержат сведения о других сборках, на которые они ссылаются (в том числе номера их версий). Эти данные делают сборку *самоописываемой* (selfdescribing). Другими словами, среда CLR может определить по порядку все прямые зависимости данной сборки, которые необходимы для ее выполнения. Не нужно размещать никакой дополнительной информации ни в системном реестре, ни в доменной службе AD DS (Active Directory Domain Services). Вследствие этого развертывать сборки гораздо проще, чем неуправляемые компоненты.

Загрузка CLR

Каждая создаваемая сборка представляет собой либо исполняемое приложение, либо библиотеку DLL, содержащую набор типов (компонентов) для использования в исполняемом приложении. Разумеется, среда CLR отвечает за

управление исполнением кода. Это значит, что на компьютере, выполняющем данное приложение, должна быть установлена платформа .NET Framework. В компании Microsoft был создан дистрибутивный пакет .NET Framework для свободного распространения, который вы можете бесплатно поставлять своим клиентам. Некоторые версии операционной системы семейства Windows поставляются с уже установленной платформой .NET Framework.

Для того чтобы понять, установлена ли платформа .NET Framework на компьютере, нужно попробовать найти файл `MSCorEE.dll` в каталоге `%SystemRoot%\system32`. Если он есть, то платформа .NET Framework установлена. Однако на одном компьютере может быть установлено одновременно несколько версий .NET Framework. Чтобы определить, какие именно версии установлены, проверьте следующий подраздел системного реестра:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP
```

В комплекте .NET Framework SDK компания Microsoft предоставляет утилиту командной строки `CLRVer.exe`, позволяющую узнать, какие версии CLR установлены на машине, а также, какая именно версия среды CLR используется текущими процессами. Для этого нужно указать параметр `-all` или идентификатор интересующего процесса.

Прежде чем мы перейдем к загрузке среды CLR, поговорим подробнее об особенностях 32- и 64-разрядных версий операционной системы Windows. Если сборка содержит только управляемый код с контролем типов, она должна одинаково хорошо работать на обеих версиях системы. Дополнительной модификации исходного кода не требуется. Созданный компилятором готовый EXE- или DLL-файл будет выполняться как на 32-разрядной Windows, так и на версиях x64 и IA64 64-разрядной Windows! Другими словами, один и тот же файл будет работать на любом компьютере с установленной платформой .NET Framework.

В исключительно редких случаях разработчикам понадобится писать код, совместимый только с какой-то конкретной версией Windows. Обычно это требуется при работе с *небезопасным кодом* (unsafe code) или для взаимодействия с неуправляемым кодом, ориентированным на конкретную процессорную архитектуру. Для таких случаев у компилятора C# предусмотрен параметр командной строки `/platform`. Этот параметр позволяет указать конкретную версию целевой платформы, на которой планируется работа данной сборки: архитектуру x86, использующую только 32-разрядную систему Windows, архитектуру x64, использующую только 64-разрядную операционную систему Windows, или архитектуру Intel Itanium, использующую только 64-разрядную систему Windows. Если не указать платформу, компилятор задействует значение по умолчанию `anycpu`, которое означает, что сборка может выполняться на любой из данных операционных систем Windows. Пользователи Visual Studio могут указать целевую платформу в списке Platform Target на вкладке Build окна свойств проекта (рис. 1.3).

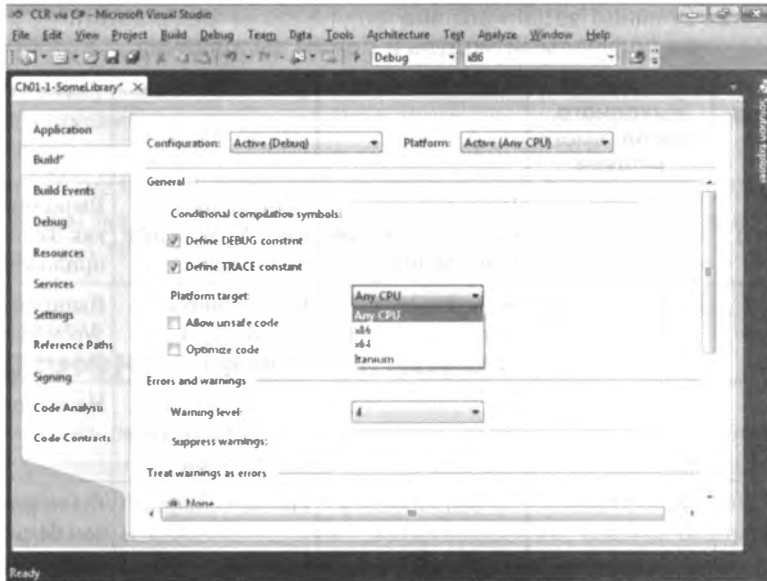


Рис. 1.3. Определение целевой платформы средствами Visual Studio

В зависимости от указанной целевой платформы C# генерирует заголовок — PE32 или PE32+, а также указывает требуемую процессорную архитектуру (или информирует о независимости от архитектуры) в заголовке. Для анализа заголовочной информации, созданной компилятором в управляемом модуле, Microsoft предоставляет две утилиты — `DumpBin.exe` и `CorFlags.exe`.

При запуске исполняемого файла Windows анализирует заголовок EXE-файла для определения того, какое именно адресное пространство необходимо для его работы — 32- или 64-разрядное. Файл с заголовком PE32 может выполняться в адресном пространстве любого из указанных двух типов, а файлу с заголовком PE32+ требуется 64-разрядное пространство. Windows также проверяет информацию о процессорной архитектуре на предмет совместимости с имеющейся конфигурацией. Наконец, 64-разрядные версии Windows поддерживают технологию выполнения 32-разрядных приложений в 64-разрядной среде, которая называется WoW64 (Windows on Windows64). Она даже позволяет выполнять 32-разрядные приложения на машине с процессором Itanium за счет эмуляции команд x86, но за это приходится расплачиваться снижением производительности.

Таблица 1.2 иллюстрирует две важные вещи. Во-первых, в ней показан тип получаемого управляемого модуля при указании разных параметров `/platform` командной строки компилятора C#. Во-вторых, в ней представлены режимы выполнения приложений в различных версиях Windows.

Таблица 1.2. Влияние заданного значения параметра /platform на получаемый модуль и режим выполнения

Значение параметра /platform	Тип выходного управляемого модуля	x86 Windows	x64 Windows	IA64 Windows
апусри (по умолчанию)	РЕ32/независимый от платформы	Выполняется как 32-разрядное приложение	Выполняется как 64-разрядное приложение	Выполняется как 32-разрядное приложение
x86	РЕ32/x86	Выполняется как 32-разрядное приложение	Выполняется как WoW64-приложение	Выполняется как WoW64-приложение
x64	РЕ32+/x64	Не выполняется	Выполняется как 64-разрядное приложение	Не выполняется
Itanium	РЕ32+/Itanium	Не выполняется	Не выполняется	Выполняется как 64-разрядное приложение

После анализа заголовка EXE-файла для выяснения того, какой процесс необходимо запустить — 32-, 64-разрядный или WoW64, — Windows загружает в адресное пространство процесса соответствующую (x86, x64 или IA64) версию библиотеки `MSCorEE.dll`. В системе Windows версии x86 одноименная версия `MSCorEE.dll` хранится в каталоге `C:\Windows\System32`. В системах x64 и IA64 версия x86 библиотеки находится в каталоге `C:\Windows\SysWow64`, а 64-разрядная версия `MSCorEE.dll` (x64 или IA64) размещается в каталоге `C:\Windows\System32` (это сделано из соображений обратной совместимости). Далее основной поток вызывает определенный в библиотеке `MSCorEE.dll` метод, который инициализирует CLR, загружает сборку EXE, а затем вызывает ее метод `Main`, в котором содержится точка входа. На этом процедура запуска управляемого приложения считается завершенной¹.

ПРИМЕЧАНИЕ

Сборки, созданные при помощи версий 7.0 и 7.1 компилятора C# от Microsoft, содержат заголовок PE32 и не зависят от архитектуры процессора. Тем не менее во время выполнения среда CLR считает их совместимыми только с архитектурой x86. Это повышает вероятность максимально корректной работы в 64-разрядной среде, так как исполняемый файл загружается в режиме WoW64, который обеспечивает процессу среду, максимально приближенную к существующей в 32-разрядной версии x86 Windows.

¹ Программный код может запросить переменную окружения `Is64BitOperatingSystem` для того, чтобы определить, выполняется ли данная программа в 64-разрядной системе Windows, а также запросить переменную окружения `Is64BitProcess`, чтобы определить, выполняется ли данная программа в 64-разрядном адресном пространстве.

Когда неуправляемое приложение вызывает LoadLibrary, Windows автоматически загружает и инициализирует CLR (если это еще не сделано) для обработки содержащегося в сборке кода. Ясно, что в такой ситуации предполагается, что процесс запущен и работает, и это сокращает область применимости сборки. В частности, управляемая сборка, скомпилированная с параметром /platform:x86, не сможет загрузиться в 64-разрядный процесс, а исполняемый файл с таким же параметром загрузится в режиме WoW64 на компьютере с 64-разрядной Windows.

Исполнение кода сборки

Как говорилось ранее, управляемые модули содержат метаданные и программный код, написанный на языке IL. Это не зависящий от процессора машинный язык, разработанный компанией Microsoft после консультаций с несколькими коммерческими и академическими организациями, специализирующимися на разработке языков и компиляторов. IL — язык более высокого уровня по сравнению с большинством других машинных языков. Он позволяет работать с объектами и имеет команды для создания и инициализации объектов, вызова виртуальных методов и непосредственного манипулирования элементами массивов. В нем даже есть команды выбрасывания и перехвата исключений для обработки ошибок. IL можно рассматривать как объектно-ориентированный машинный язык.

Обычно разработчики программируют на высокоуровневых языках, таких как C#, C++/CLI или Visual Basic. Компиляторы этих языков генерируют IL-код. Однако такой код может быть написан и на языке ассемблера, так, Microsoft предоставляет ассемблер IL (ILAsm.exe), а также дизассемблер IL (ILDasm.exe).

Имейте в виду, что любой язык высокого уровня, скорее всего, использует лишь часть возможностей, предоставляемых CLR. При этом язык ассемблера IL открывает доступ ко всем возможностям CLR. В случае если выбранный вами язык программирования не дает доступа именно к тем функциям CLR, которые необходимы, можно написать часть программного кода на языке ассемблера IL или на другом языке программирования, позволяющем их задействовать.

Единственный способ узнать о возможностях CLR, доступных при использовании конкретного языка, — изучить соответствующую документацию. В этой книге сделан акцент на возможностях среды CLR и на том, какие из этих возможностей доступны при программировании на C#. Можно сделать предположение, что в других книгах и статьях среда CLR рассмотрена с точки зрения других языков и разработчики получают представление лишь о тех ее функциях, которые доступны при использовании описанных там языков. По крайней мере, если выбранный язык решает поставленные задачи, такой подход не так уж плох.

ВНИМАНИЕ

Я думаю, что возможность легко переключаться между языками при их тесной интеграции — чудесное качество CLR. К сожалению, я также практически уверен, что разработчики часто будут проходить мимо нее. Такие языки, как C# и Visual Basic, прекрасно подходят для программирования ввода-вывода. Язык APL (A Programming Language) — замечательный язык для инженерных и финансовых расчетов. Среда CLR позволяет написать на C# часть приложения, отвечающую за ввод-вывод, а инженерные расчеты — на языке APL. Среда CLR предлагает беспрецедентный уровень интеграции этих языков, и во многих проектах стоит серьезно задуматься об использовании одновременно нескольких языков.

Для выполнения какого-либо метода его IL-код должен быть преобразован в машинные команды. Этим занимается JIT-компилятор (Just-In-Time) среды CLR.

На рис. 1.4 показано, что происходит при первом обращении к методу.

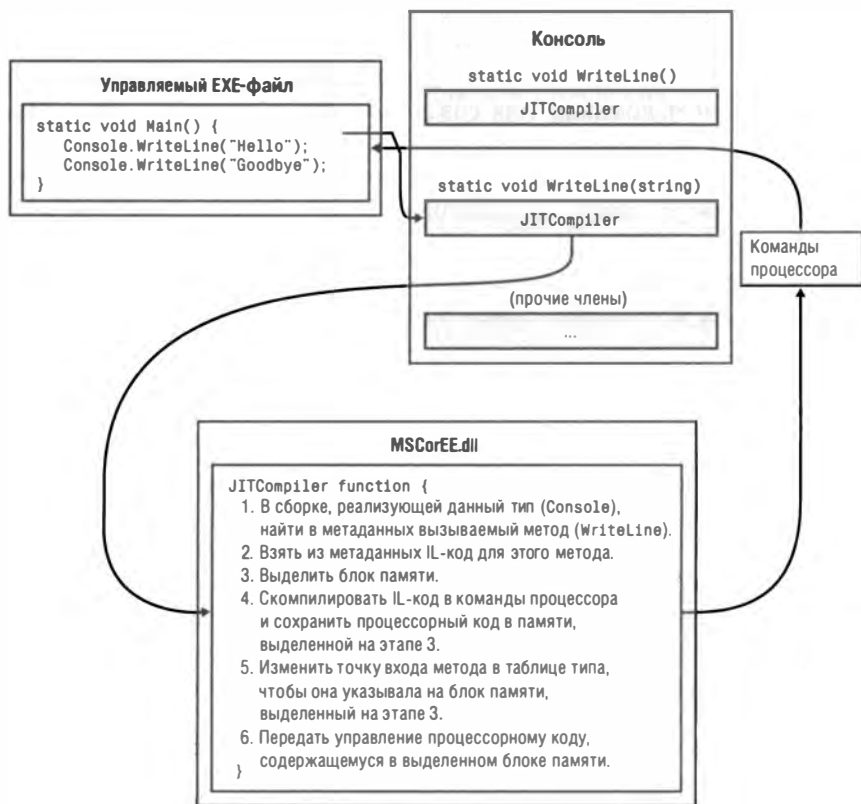


Рис. 1.4. Первый вызов метода

Непосредственно перед исполнением метода Main среда CLR находит все типы данных, на которые ссылается программный код метода Main. При этом CLR выделяет внутренние структуры данных, используемые для управления доступом к типам, на которые есть ссылки. На рисунке 1.4 метод Main ссылается на единственный тип — Console, и среда CLR выделяет единственную внутреннюю структуру. Эта внутренняя структура данных содержит по одной записи для каждого метода, определенного в типе Console. Каждая запись содержит адрес, по которому можно найти реализацию метода. При инициализации этой структуры CLR заносит в каждую запись адрес внутренней недокументированной функции, содержащейся в самой среде CLR. Эта функция называется JITCompiler.

Когда метод Main первый раз обращается к методу WriteLine, вызывается функция JITCompiler. Она отвечает за компиляцию IL-кода вызываемого метода в собственные команды процессора. Поскольку IL-код компилируется непосредственно перед выполнением («just in time»), этот компонент CLR часто называют *JIT-термом*, или *JIT-компилятором*.

ПРИМЕЧАНИЕ

Если приложение исполняется в x86 версии Windows или в режиме WoW64, JIT-компилятор генерирует команды для x86 архитектуры. Для приложений, выполняющихся как 64-разрядные в версии x64 или Itanium OC Windows, JIT-компилятор генерирует соответственно команды для архитектуры x64 или IA64.

Функции JITCompiler известен вызываемый метод и тип, в котором он определен. JITCompiler ищет в метаданных соответствующей сборки IL-код вызываемого метода. Затем JITCompiler проверяет и компилирует IL-код в собственные машинные команды, которые сохраняются в динамически выделенном блоке памяти. После этого JITCompiler возвращается к структуре внутренних данных типа, созданной средой CLR, и заменяет адрес вызываемого метода адресом блока памяти, содержащего готовые машинные команды. В завершение JITCompiler передает управление коду в этом блоке памяти. Этот программный код является реализацией метода WriteLine (вариант этого метода с параметром String). Из этого метода управление возвращается в метод Main, который продолжает выполнение в обычном порядке.

Рассмотрим повторное обращение метода Main к методу WriteLine. К этому моменту код метода WriteLine уже проверен и скомпилирован, так что обращение к блоку памяти производится, минуя вызов JITCompiler. Отработав, метод WriteLine возвращает управление методу Main. На рис. 1.5 показано, как выглядит ситуация при повторном обращении к методу WriteLine.

Снижение производительности наблюдается только при первом вызове метода. Все последующие обращения выполняются «на максимальной скорости», потому что повторная верификация и компиляция не производятся.

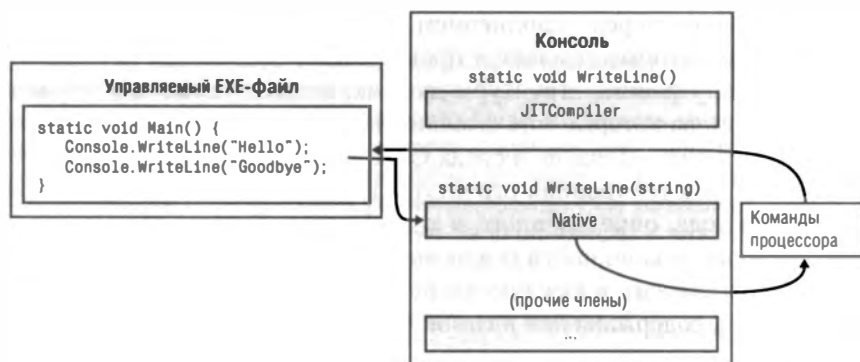


Рис. 1.5. Повторный вызов метода

JIT-компилятор хранит машинные команды в динамической памяти. Это значит, что скомпилированный код уничтожается по завершении работы приложения. Для повторного вызова приложения или для параллельного запуска его второго экземпляра (в другом процессе операционной системы) JIT-компилятору придется заново скомпилировать IL-код в машинные команды.

Для большинства приложений снижение производительности, связанное с работой JIT-компилятора, незначительно. Большинство приложений раз за разом обращается к одним и тем же методам. На производительности это сказывается только один раз во время выполнения приложения. К тому же больше времени занимает выполнение самого метода, а не обращение к нему.

Необходимо также знать, что JIT-компилятор среды CLR оптимизирует машинный код аналогично компилятору неуправляемого кода C++. И опять же: создание оптимизированного кода занимает больше времени, но при выполнении он гораздо производительнее, чем неоптимизированный.

Есть два параметра компилятора C#, влияющих на оптимизацию кода, — `/optimize` и `/debug`. В следующей таблице показано их влияние на качество

IL-кода, созданного компилятором C#, и машинного кода, сгенерированного JIT-компилятором.

Параметры компилятора	Качество IL-кода компилятора	Качество машинного JIT-кода
/optimize- /debug- (по умолчанию)	Неоптимизированный	Оптимизированный
/optimize- /debug(+ /full /pdbonly)	Неоптимизированный	Неоптимизированный
/optimize+ /debug(- /+ /full /pdbonly)	Оптимизированный	Оптимизированный

С параметром /optimize- компилятор C# генерирует неоптимизированный IL-код, содержащий множество пустых команд (no-operation, NOP). Эти команды предназначены для поддержки функции «редактирования с продолжением выполнения» (edit-and-continue) в Visual Studio во время процесса отладки. Они также упрощают процесс отладки, позволяя расставлять точки останова (breakpoints) на управляющих командах, таких как for, while, do, if, else, а также блоках try, catch и finally. Во время оптимизации IL-кода компилятор C# удаляет эти посторонние команды, усложняя процесс отладки кода, но зато оптимизируя поток управления программой. Кроме того, возможно, некоторые оценочные функции не выполняются во время отладки. Однако IL-код меньше по размерам, и это уменьшает результирующий размер EXE- или DLL-файлов; кроме того, IL-код легче читать тем, кто обожает исследовать IL-код, пытаясь понять, что именно породил компилятор (например, мне).

Глава 2. Компоновка, упаковка, развертывание и администрирование приложений и типов

Прежде чем перейти к главам, описывающим разработку программ для Microsoft .NET Framework, давайте обсудим вопросы создания, упаковки и развертывания приложений и их типов. В этой главе акцент сделан на основах создания компонентов, предназначенных исключительно для ваших приложений. В главе 3 рассказано о ряде более сложных, но очень важных концепций, в том числе способах создания и применения сборок, содержащих компоненты, предназначенные для использования совместно с другими приложениями. В этой и следующей главах также показано, как администратор может влиять на исполнение приложения и его типов.

Современные приложения состоят из типов, которые создаются самими разработчиками или компанией Microsoft. Помимо этого, процветает целая отрасль поставщиков компонентов, которые используются клиентами, чтобы сократить время на разработку проектов. Типы, реализованные при помощи языка, ориентированного на общезыковую исполняющую среду (CLR), способны легко работать друг с другом, базовый класс такого типа можно даже написать на другом языке программирования.

В этой главе объясняется, как эти типы создаются и упаковываются в файлы, предназначенные для развертывания. В процессе изложения дается краткий исторический обзор некоторых проблем, решенных с приходом .NET Framework.

Задачи развертывания в .NET Framework

Все годы своего существования операционная система Windows «славилась» нестабильностью и чрезмерной сложностью. Такая репутация, заслуженная или нет, сложилась по ряду причин. Во-первых, все приложения используют динамически подключаемые библиотеки (Dynamic Link Library, DLL), созданные Microsoft и другими производителями. Поскольку приложение исполняет код, написанный разными производителями, ни один разработчик какой-либо части

программы не может быть на 100 % уверен в том, что точно знает, как другие собираются применять созданный им код. В теории такая ситуация чревата любыми неполадками, но на практике взаимодействие кодов от разных производителей редко становится источником проблем, так как перед развертыванием приложения тестируют и отлаживают.

Однако пользователи часто сталкиваются с проблемами, когда производитель решает обновить поставленную им программу и предоставляет новые файлы. Предполагается, что новые файлы обеспечивают «обратную совместимость» с прежним программным обеспечением, но кто за это поручится? Одному производителю, выпускающему обновление своей программы, фактически не под силу заново протестировать и отладить все существующие приложения, чтобы убедиться, что изменения при обновлении не влекут за собой нежелательных последствий.

Уверен, что каждый читающий эту книгу сталкивался с той или иной разновидностью проблемы, когда после установки нового приложения нарушалась работа одной (или нескольких) из установленных ранее программ. Эта проблема получила название «ад DLL». Подобная уязвимость вселяет ужас в сердца и умы обычных пользователей компьютеров. В конечном итоге пользователи должны как следует обдумать, стоит ли устанавливать новое программное обеспечение на их компьютеры. Что касается меня, то я решил вовсе не пробовать устанавливать некоторые приложения из опасения, что они нанесут вред наиболее важным для меня программам.

Второй фактор, повлиявший на репутацию Windows, — сложности при установке приложений. Большинство приложений при установке умудряются «просочиться» во все части операционной системы. Например, при установке приложения происходит копирование файлов в разные каталоги, модификация параметров реестра, установка ярлыков и ссылок на рабочий стол (Desktop), в меню Пуск (Start) и на панель быстрого запуска. Проблема в том, что приложение — это не одиночная изолированная сущность. Нельзя легко и просто создать резервную копию приложения, поскольку, кроме файлов приложения, придется скопировать соответствующие части реестра. Вдобавок, нельзя просто взять и переместить приложение с одной машины на другую — для этого нужно запустить программу установки еще раз, чтобы корректно скопировать все файлы и параметры реестра. Наконец, приложение не всегда просто удалить — часто остается неприятное ощущение, что какая-то его часть затаилась где-то внутри компьютера.

Третий фактор — безопасность. При установке приложений записывается множество файлов, созданных самыми разными компаниями. Вдобавок, многие веб-приложения (например, ActiveX) зачастую содержат программный код, который сам загружается из Интернета, о чем пользователи даже не подозревают. На современном уровне технологий такой код может выполнять любые действия, включая удаление файлов и рассылку электронной почты. Пользователи справедливо опасаются устанавливать новые приложения из-за угрозы потен-

циального вреда, который может быть нанесен их компьютерам. Для того чтобы пользователи чувствовали себя спокойнее, в системе должны быть встроенные функции защиты, позволяющие явно разрешать или запрещать доступ к системным ресурсам коду, созданному теми или иными компаниями.

Как показано в этой и следующей главах, платформа .NET Framework устраняет «ад DLL» и делает существенный шаг вперед к решению проблемы, связанной с распределением данных приложения по всей операционной системе. Например, в отличие от модели COM компонентам больше не требуется хранить свои параметры в реестре. К сожалению, приложениям пока еще требуются ссылки и ярлыки. Совершенствование системы защиты связано с новой моделью безопасности платформы .NET Framework — *безопасностью доступа на уровне кода* (code access security). Если безопасность системы Windows основана на идентификации пользователя, то безопасность доступа к коду — на правах, которые контролируются хостом приложений, загружающим компоненты. Такой хост приложения, как Microsoft Silverlight, может предоставить совсем немного полномочий загруженному программному коду, в то время как локально установленное приложение во время своего выполнения может иметь уровень полного доверия (со всеми полномочиями). Как видите, платформа .NET Framework предоставляет пользователям намного больше возможностей по контролю над тем, что устанавливается и выполняется на их машинах, чем когда-либо давала им система Windows.

Компоновка типов в модуль

В этом разделе рассказывается, как превратить файл, содержащий исходный код с разными типами, в файл, пригодный для развертывания. Для начала рассмотрим следующее простое приложение:

```
public sealed class Program {
    public static void Main() {
        System.Console.WriteLine("Hi");
    }
}
```

Здесь определен тип Program с единственным статическим открытым методом Main. Внутри метода Main находится ссылка на другой тип — System.Console. Этот тип разработан в компании Microsoft, и его программный код на языке IL, реализующий его методы, находится в файле MSCorLib.dll. Таким образом, данное приложение определяет собственный тип, а также использует тип, созданный другой компанией.

Для того чтобы скомпоновать это приложение-пример, сохраните этот код в файле Program.cs, а затем наберите в командной строке следующее:

```
csc.exe /out:Program.exe /t:exe /r:MSCorLib.dll Program.cs
```

Эта команда указывает компилятору C# создать исполняемый файл **Program.exe** (имя задано параметром `/out:Program.exe`). Тип создаваемого файла — консольное приложение Win32 (тип задан параметром `/t[target]:exe`).

При обработке файла с исходным кодом компилятор C# обнаруживает ссылку на метод `WriteLine` типа `System.Console`. На этом этапе компилятор должен убедиться, что этот тип существует и у него есть метод `WriteLine`. Компилятор также проверяет, чтобы типы аргументов, предоставляемых программой, совпадали с ожидаемыми типами метода `WriteLine`. Поскольку тип не определен в исходном коде на C#, компилятору C# необходимо передать набор сборок, которые позволят ему разрешить все ссылки на внешние типы. В показанной команде параметр `/r[eference]:mscorlib.dll` приказывает компилятору вести поиск внешних типов в сборке, идентифицируемой файлом `mscorlib.dll`.

`mscorlib.dll` — это особый файл в том смысле, что в нем находятся все основные типы: `Byte`, `Char`, `String`, `Int32` и т. д. В действительности, эти типы используются так часто, что компилятор C# ссылается на эту сборку (`mscorlib.dll`) автоматически. Другими словами, следующая команда (в ней опущен параметр `/r`) даст тот же результат, что и предыдущая:

```
csc.exe /out:Program.exe /t:exe Program.cs
```

Более того, поскольку значения, заданные параметрами командной строки `/out:Program.exe` и `/t:exe`, совпадают со значениями по умолчанию, следующая команда даст аналогичный результат:

```
csc.exe Program.cs
```

Если по какой-то причине вы не хотите, чтобы компилятор C# ссылался на сборку `mscorlib.dll`, используйте параметр `/nostdlib`. В компании Microsoft задействуют именно этот параметр при компоновке сборки `mscorlib.dll`. Например, во время исполнения следующей команды при компиляции файла `Program.cs` генерируется ошибка, поскольку тип `System.Console` определен в сборке `mscorlib.dll`:

```
csc.exe /out:Program.exe /t:exe /nostdlib Program.cs
```

А теперь присмотримся поближе к файлу `Program.exe`, созданному компилятором C#. Что он из себя представляет? Для начала это стандартный файл в формате PE (portable executable). Это значит, что машина, работающая под управлением 32- или 64-разрядной версии Windows, способна загрузить этот файл и что-нибудь с ним сделать. Система Windows поддерживает два типа приложений: с консольными (Console User Interface, CUI) и графическими пользовательскими интерфейсами (Graphical User Interface, GUI). Параметр `/t:exe` указывает компилятору C# создать консольное приложение. Для создания приложения с графическим интерфейсом необходимо указать параметр `/t:winexe`.

Файл параметров

В завершение рассказа о параметрах компилятора хотелось бы сказать несколько слов о *файлах параметров* (response files) — текстовых файлах, содержащих набор параметров командной строки для компилятора. При выполнении компилятора **CSC.exe** открывается файл параметров и используются все указанные в нем параметры, как если бы они были переданы в составе командной строки. Файл параметров передается компилятору путем указания его в командной строке с префиксом **@**. Например, пусть есть файл параметров **MyProject.rsp** со следующим текстом:

```
/out:MyProject.exe /target:winexe
```

Для того чтобы компилятор (**CSC.exe**) использовал эти параметры, необходимо вызвать файл следующим образом:

```
csc.exe @MyProject.rsp CodeFile1.cs CodeFile2.cs
```

Эта строка сообщает компилятору **C#** имя выходного файла и тип скомпилированной программы. Очевидно, что файлы параметров исключительно полезны, так как избавляют от необходимости вручную вводить все аргументы командной строки каждый раз при компиляции проекта.

Компилятор **C#** поддерживает работу с несколькими файлами параметров. Помимо явно указанных в командной строке файлов, компилятор автоматически ищет файл с именем **CSC.rsp** в текущем каталоге, поэтому относящиеся к проекту параметры нужно указывать именно в этом файле. Компилятор также проверяет каталог с файлом **CSC.exe** на наличие глобального файла параметров **CSC.rsp**, в котором следует указывать параметры, относящиеся ко всем проектам. В процессе своей работы компилятор объединяет параметры из всех файлов и использует их. В случае конфликтующих параметров в глобальных и локальных файлах предпочтение отдается последним. Кроме того, любые явно заданные в командной строке параметры имеют более высокий приоритет, чем указанные в локальных файлах параметров.

При установке платформы **.NET Framework** по умолчанию глобальный файл **CSC.rsp** устанавливается в каталог `%SystemRoot%\Microsoft.NET\Framework\vX.X.X` (где **X.X.X** — версия устанавливаемой платформы **.NET Framework**). В версии 4.0 этот файл содержит следующие параметры:

```
# Этот файл содержит параметры командной строки,
# которые компилятор C# командной строки (CSC)
# будет обрабатывать в каждом сеансе компиляции,
# если только не задан параметр "/noconfig".
```

```
# Ссылки на стандартные библиотеки Framework
/r:Accessibility.dll
/r:Microsoft.CSharp.dll
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
```

```
/r:System.Core.dll  
/r:System.Data.dll  
/r:System.Data.DataSetExtensions.dll  
/r:System.Data.Linq.dll  
/r:System.Deployment.dll  
/r:System.Device.dll  
/r:System.DirectoryServices.dll  
/r:System.dll  
/r:System.Drawing.dll  
/r:System.EnterpriseServices.dll  
/r:System.Management.dll  
/r:System.Messaging.dll  
/r:System.Runtime.Remoting.dll  
/r:System.Runtime.Serialization.dll  
/r:System.Runtime.Serialization.Formatter.Soa.dll  
/r:System.Security.dll  
/r:System.ServiceModel.dll  
/r:System.ServiceProcess.dll  
/r:System.Transactions.dll  
/r:System.Web.Services.dll  
/r:System.Windows.Forms.dll  
/r:System.Xml.dll  
/r:System.Xml.Linq.dll
```

В глобальном файле **CSC.rsp** есть ссылки на все перечисленные сборки, поэтому нет необходимости указывать их явно с помощью параметра `/reference`. Этот файл параметров исключительно удобен для разработчиков, так как позволяет использовать все типы и пространства имен, определенные в различных опубликованных компанией Microsoft сборках, не указывая их все явно с применением параметра `/reference`.

Ссылки на все эти сборки могут немного замедлить работу компилятора, но если в исходном коде нет ссылок на типы или члены этих сборок, это никак не сказывается ни на результирующем файле сборки, ни на производительности его выполнения.

ПРИМЕЧАНИЕ

При использовании параметра `/reference` для ссылки на какую-либо сборку можно указать полный путь к конкретному файлу. Однако если такой путь не указать, компилятор будет искать нужный файл в следующих местах (в указанном порядке):

Рабочий каталог.

Каталог, содержащий файл самого компилятора (**CSC.exe**). Библиотека **MSCorLib.dll** всегда извлекается из этого каталога. Путь к нему имеет примерно следующий вид:

```
%SystemRoot%\Microsoft.NET\Framework\v4.0.####.
```

Все каталоги, указанные с использованием параметра `/lib` компилятора.

Все каталоги, указанные в переменной окружения **LIB**.

Конечно, вы вправе добавлять собственные параметры в глобальный файл `CSC.rsp`, но это сильно усложняет репликацию среды компоновки на разных машинах — приходится помнить про обновление файла `CSC.rsp` на всех машинах, используемых для сборки приложений. Можно также дать компилятору команду игнорировать как локальный, так и глобальный файлы `CSC.rsp`, указав в командной строке параметр `/noconfig`.

Несколько слов о метаданных

Что же именно находится в файле `Program.exe`? Управляемый PE-файл состоит из 4-х частей: заголовка `PE32(+)`, заголовка `CLR`, метаданных и кода на промежуточном языке (*intermediate language, IL*). Заголовок `PE32(+)` хранит стандартную информацию, ожидаемую Windows. Заголовок `CLR` — это небольшой блок информации, специфичной для модулей, требующих `CLR` (управляемых модулей). В него входит старший и младший номера версии `CLR`, для которой скомпонован модуль, ряд флагов и маркер `MethodDef` (о нем — чуть позже), указывающий метод точки входа в модуль, если это исполняемый `CUI`- или `GUI`-файл, а также необязательную сигнатуру строгого имени (она рассмотрена в главе 3). Наконец, заголовок содержит размер и смещение некоторых таблиц метаданных, расположенных в модуле. Для того чтобы узнать точный формат заголовка `CLR`, изучите структуру `IMAGE_COR20_HEADER`, определенную в файле `CorHdr.h`.

Метаданные — это блок двоичных данных, состоящий из нескольких таблиц. Существуют три категории таблиц: определений, ссылок и манифестов. В табл. 2.1 приводится описание некоторых наиболее распространенных таблиц определений, существующих в блоке метаданных модуля.

Таблица 2.1. Общие таблицы определений, входящих в метаданные

Имя таблицы определений	Описание
ModuleDef	Всегда содержит одну запись, идентифицирующую модуль. Запись включает имя файла модуля с расширением (без указания пути к файлу) и идентификатор версии модуля (в виде созданного компилятором значения GUID). Это позволяет переименовывать файл, не теряя сведений о его исходном имени. Однако настоятельно рекомендуется не переименовывать файл, иначе среда CLR может не найти сборку во время выполнения
TypeDef	Содержит по одной записи для каждого типа, определенного в модуле. Каждая запись включает имя типа, базовый тип, флаги сборки (<code>public</code> , <code>private</code> и т. д.) и указывает на записи таблиц <code>MethodDef</code> , <code>PropertyDef</code> и <code>EventDef</code> , содержащие соответственно сведения о методах, свойствах и событиях этого типа

Имя таблицы определений	Описание
MethodDef	Содержит по одной записи для каждого метода, определенного в модуле. Каждая строка включает имя метода, флаги (private, public, virtual, abstract, static, final и т. д.), сигнатуру и смещение в модуле, по которому находится соответствующий IL-код. Каждая запись также может ссылаться на запись в таблице ParamDef, где хранятся дополнительные сведения о параметрах метода
FieldDef	Содержит по одной записи для каждого поля, определенного в модуле. Каждая запись состоит из флагов (например, private, public и т. д.) и типа поля
ParamDef	Содержит по одной записи для каждого параметра, определенного в модуле. Каждая запись состоит из флагов (in, out, retval и т. д.), типа и имени
PropertyDef	Содержит по одной записи для каждого свойства, определенного в модуле. Каждая запись включает имя, флаги, тип и вспомогательное поле (оно может быть пустым)
EventDef	Содержит по одной записи для каждого события, определенного в модуле. Каждая запись включает имя и флаги

Для каждой сущности, определяемой в компилируемом исходном тексте, компилятор генерирует строку в одной из таблиц, перечисленных в табл. 2.1. В ходе компиляции исходного текста компилятор также обнаруживает типы, поля, методы, свойства и события, на которые имеются ссылки в исходном тексте. Все сведения о найденных сущностях регистрируются в нескольких таблицах ссылок, составляющих метаданные. В табл. 2.2 показаны некоторые наиболее распространенные таблицы ссылок, которые входят в состав метаданных.

Таблица 2.2. Общие таблицы ссылок, входящие в метаданные

Имя таблицы ссылок	Описание
AssemblyRef	Содержит по одной записи для каждой сборки, на которую ссылается модуль. Каждая запись включает сведения, необходимые для привязки к сборке: ее имя (без указания расширения и пути), номер версии, региональные стандарты и маркер открытого ключа (обычно это небольшой хэш, созданный на основе открытого ключа издателя и идентифицирующий издателя сборки, на которую ссылается модуль). Каждая запись также содержит несколько флагов и хэш. Этот хэш служит контрольной суммой битов, составляющих сборку, на которую ссылается код. Среда CLR полностью игнорирует этот хэш и, вероятно, будет игнорировать его в будущем

продолжение ➤

Таблица 2.2 (продолжение)

Имя таблицы ссылок	Описание
ModuleRef	Содержит по одной записи для каждого РЕ-модуля, реализующего типы, на которые он ссылается. Каждая запись включает имя файла сборки и его расширение (без указания пути). Эта таблица служит для привязки модуля вызывающей сборки к типам, реализованным в других модулях
TypeRef	Содержит по одной записи для каждого типа, на который ссылается модуль. Каждая запись включает имя типа и ссылку, по которой можно его найти. Если этот тип реализован внутри другого типа, запись содержит ссылку на соответствующую запись таблицы TypeRef. Если тип реализован в том же модуле, приводится ссылка на запись таблицы ModuleDef. Если тип реализован в другом модуле вызывающей сборки, приводится ссылка на запись таблицы ModuleRef. Если тип реализован в другой сборке, приводится ссылка на запись в таблице AssemblyRef
MemberRef	Содержит по одной записи для каждого члена (поля, метода, а также свойства или метода события), на который ссылается модуль. Каждая запись включает имя и сигнатуру члена и указывает на запись таблицы TypeRef, содержащую сведения о типе, определяющим этот член

На самом деле таблиц метаданных намного больше, чем показано в табл. 2.1 и 2.2, я просто хотел создать у вас представление об информации, на основании которой компилятор создает метаданные. Ранее уже упоминалось о том, что в состав метаданных входят также таблицы манифестов. О них мы поговорим чуть позже.

Метаданные управляемого РЕ-файла можно изучать при помощи различных инструментов. Лично я предпочитаю ILDasm.exe — дизассемблер языка IL. Для того чтобы увидеть содержимое таблиц метаданных, выполните следующую команду:

ILDasm Program.exe

Запустится файл ILDasm.exe и загрузится сборка Program.exe. Для того чтобы вывести метаданные в читабельном виде, выберите в меню команду **View ▶ MetaInfo ▶ Show!** (или нажмите клавиши **Ctrl+M**). В результате появится следующая информация:

```
=====
ScopeName : Program.exe
MVID : {CA73FFE8-0D42-4610-A8D3-9276195C35AA}
=====
Global functions
-----
Global fields
```

```
-----
Global MemberRefs
-----
```

```
TypeDef #1 (02000002)
-----
```

```
TypDefName: Program (02000002)
Flags      : [Public] [AutoLayout] [Class] [Sealed] [AntiClass]
             [BeforeFieldInit] (00100101)
Extends    : 01000001 [TypeRef] System.Object
Method #1 (06000001) [ENTRYPOINT]
-----
    MethodName: Main (06000001)
    Flags      : [Public] [Static] [HideBySig] [ReuseSlot] (00000096)
    RVA        : 0x00002050
    ImplFlags  : [IL] [Managed] (00000000)
    CallCnvtn: [DEFAULT]
    ReturnType: Void
    No arguments.
```

```
Method #2 (06000002)
-----
```

```
    MethodName: .ctor (06000002)
    Flags      : [Public] [HideBySig] [ReuseSlot] [SpecialName]
                 [RTSpecialName] [Ldc] (00001886)
    RVA        : 0x0000205c
    ImplFlags  : [IL] [Managed] (00000000)
    CallCnvtn: [DEFAULT]
    hasThis
    ReturnType: Void
    No arguments.
```

```
TypeRef #1 (01000001)
-----
```

```
Token:          0x01000001
ResolutionScope: 0x23000001
TypeRefName:    System.Object
MemberRef #1 (0a000004)
-----
```

```
    Member: (0a000004) .ctor:
    CallCnvtn: [DEFAULT]
    hasThis
    ReturnType: Void
    No arguments.
```

```
TypeRef #2 (01000002)
-----
```

```
Token:          0x01000002
ResolutionScope: 0x23000001
```


TypeRefName: System.Runtime.CompilerServices.CompilationRelaxationsAttribute
MemberRef #1 (0a000001)

```
-----
Member: (0a000001) .ctor:
CallCnvtn: [DEFAULT]
hasThis
ReturnType: Void
1 Arguments
Argument #1: I4
```

TypeRef #3 (01000003)

```
-----
Token:                0x01000003
ResolutionScope:     0x23000001
TypeRefName:        System.Runtime.CompilerServices.
RuntimeCompatibilityAttribute
MemberRef #1 (0a000002)
```

```
-----
Member: (0a000002) .ctor:
CallCnvtn: [DEFAULT]
hasThis
ReturnType: Void
No arguments.
```

TypeRef #4 (01000004)

```
-----
Token:                0x01000004
ResolutionScope:     0x23000001
TypeRefName:        System.Console
MemberRef #1 (0a000003)
```

```
-----
Member: (0a000003) WriteLine:
CallCnvtn: [DEFAULT]
ReturnType: Void
1 Arguments
Argument #1: String
```

Assembly

```
-----
Token: 0x20000001
Name : Program
Public Key :
Hash Algorithm : 0x00008004
Version: 0.0.0.0
Major Version: 0x00000000
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
```

```

Locale: <null>
Flags : [none] (00000000)
CustomAttribute #1 (0c000001)
-----
    CustomAttribute Type: 0a000001
    CustomAttributeName:
        System.Runtime.CompilerServices.CompilationRelaxationsAttribute ;;
        instance void .ctor(int32)
    Length: 8
    Value : 01 00 08 00 00 00 00 00          >          <
    ctor args: (8)

CustomAttribute #2 (0c000002)
-----
    CustomAttribute Type: 0a000002
    CustomAttributeName: System.Runtime.CompilerServices.
RuntimeCompatibilityAttribute ::
    instance void .ctor()
    Length: 30
    Value : 01 00 01 00 54 02 16 57 72 61 70 4e 6f 6e 45 78 >    T WrapNonEx<
           : 63 65 70 74 69 6f 6e 54 68 72 6f 77 73 01      >ceptionThrows    <
    ctor args: ()

AssemblyRef #1 (23000001)
-----
    Token: 0x23000001
    Public Key or Token: b7 7a 5c 56 19 34 e0 89
    Name: mscorlib
    Version: 2.0.0.0
    Major Version: 0x00000002
    Minor Version: 0x00000000
    Build Number: 0x00000000
    Revision Number: 0x00000000
    Locale: <null>
    HashValue Blob:
    Flags: [none] (00000000)

User Strings
-----
70000001 : ( 2) L"Hi"

Coff symbol name overhead:  0
=====
=====
=====
=====

```

К счастью, ILDasm самостоятельно обрабатывает таблицы метаданных и комбинирует информацию, поэтому пользователю не приходится заниматься

синтаксическим разбором необработанных табличных данных. Например, в приведенном фрагменте видно, что, показывая строку таблицы TypeDef, ILDasm выводит перед первой записью таблицы TypeRef определение соответствующего члена.

Не обязательно понимать, что означает каждая строка этого дампа, важно запомнить, что **Program.exe** содержит в таблице TypeDef описание типа Program. Этот тип идентифицирует открытый изолированный класс, производный от System.Object (то есть это ссылка на тип из другой сборки). Тип Program также определяет два метода: Main и .ctor (конструктор).

Метод Main — это статический открытый метод, чей программный код представлен на языке IL (а не в машинных кодах процессора, например x86). Main возвращает тип void и принимает единственный аргумент args — массив значений типа String. Метод-конструктор (всегда отображаемый под именем .ctor) является открытым, его код также записан на языке IL. Тип возвращаемого значения конструктора — void, у него нет аргументов, но есть указатель this, ссылающийся на область памяти, в которой должен создаваться экземпляр объекта при вызове конструктора.

Я настоятельно рекомендую вам поэкспериментировать с дизассемблером ILDasm. Он предоставляет массу сведений, и чем лучше вы в них разберетесь, тем быстрее изучите общезыковую исполняющую среду CLR и ее возможности. В этой книге еще не раз будет использоваться дизассемблер ILDasm.

Ради забавы посмотрим на некоторую статистику сборки Program.exe. Выбрав в меню программы ILDasm команду View ► Statistics, увидим следующее:

```
File size : 3072
PE header size      : 512 (496 used)(16.67%)
PE additional info  : 839 (27.31%)
Num.of PE sections  : 3
CLR header size     : 72 ( 2.34%)
CLR meta-data size  : 604 (19.66%)
CLR additional info : 0 ( 0.00%)
CLR method headers  : 2 ( 0.07%)
Managed code       : 18 ( 0.59%)
Data                : 1536 (50.00%)
Unaccounted         : -511 (-16.63%)

Num.of PE sections  : 3
  .text             - 1024
  .rsrc              - 1024
  .reloc             - 512

CLR metadata size   : 604
Module              - 1 (10 bytes)
TypeDef             - 2 (28 bytes) 0 interfaces, 0 explicit layout
```

TypeRef	- 4 (24 bytes)
MethodDef	- 2 (28 bytes) 0 abstract, 0 native, 2 bodies
MemberRef	- 4 (24 bytes)
CustomAttribute	- 2 (12 bytes)
Assembly	- 1 (22 bytes)
AssemblyRef	- 1 (20 bytes)
Strings	- 176 bytes
Blobs	- 68 bytes
UserStrings	- 8 bytes
Guids	- 16 bytes
Uncategorized	- 168 bytes

CLR method headers : 2
Num.of method bodies - 2
Num.of fat headers - 0
Num.of tiny headers - 2

Managed code : 18
Ave method size - 9

Здесь можно видеть как размеры самого файла (в байтах), так и размеры его составляющих частей (в байтах и процентах от размера файла). Приложение Program.cs очень маленькое, поэтому большая часть его файла занята заголовком PE и метаданными. Фактически IL-код занимает всего 18 байт. Конечно, чем больше размер приложения, тем чаще типы и ссылки на другие типы и сборки используются повторно, поэтому размеры метаданных и данных заголовка существенно уменьшаются по отношению к общему размеру файла.

ПРИМЕЧАНИЕ

Кстати (некстати?), в ILDasm.exe есть «жучок», искажающий отображаемую информацию о размере файла. В частности, нельзя доверять сведениям в строке Unaccounted.

Объединение модулей для создания сборки

Файл Program.exe — это не просто PE-файл с метаданными, а еще и *сборка* (assembly), то есть коллекция из одного или нескольких файлов с определениями типов и файлов ресурсов. Один из файлов сборки выбирают для хранения ее манифеста. *Манифест* (manifest) — это еще один набор таблиц метаданных, которые в основном содержат имена файлов, составляющих сборку. Кроме того, эти таблицы описывают версию и региональные стандарты сборки, ее издателя, общедоступные экспортируемые типы, а также все составляющие сборку файлы.

CLR работает со сборками, то есть сначала CLR всегда загружает файл с таблицами метаданных манифеста, а затем получает из манифеста имена остальных файлов сборки. Некоторые характеристики сборки стоит запомнить:

- в сборке определены многократно используемые типы;
- сборка помечена номером версии;
- со сборкой может быть связана информация безопасности.

У отдельных файлов сборки, кроме файла с таблицами метаданных манифеста, таких атрибутов нет.

Чтобы упаковать типы, сделать их доступными, а также обеспечить безопасность типов и управление их версиями, нужно поместить типы в модули, объединенные в сборку. Чаще всего сборка состоит из одного файла, как приложение `Program.exe` в рассмотренном примере, но могут быть и сборки из нескольких файлов: PE-файлов с метаданными и файлов ресурсов, например GIF- или JPG-файлов. Наверное, проще представлять себе сборку как «логический» EXE- или DLL-файл.

Уверен, многим читателям интересно, зачем компании Microsoft понадобилось вводить новое понятие — «сборка». Дело в том, что сборка позволяет разграничить логическое и физическое понятия многократно используемых типов. Допустим, сборка состоит из нескольких типов. При этом типы, применяемые чаще всех, можно поместить в один файл, а применяемые реже — в другой. Если сборка развертывается путем загрузки через Интернет, клиент может вовсе не загружать файл с редко используемыми типами, если он никогда их не задействует. Например, независимый поставщик ПО (independent software vendor, ISV), специализирующийся на разработке элементов управления пользовательского интерфейса, может реализовать в отдельном модуле активно используемые типы Active Accessibility (необходимые для соответствия требованиям логотипа Microsoft). Загружать этот модуль достаточно лишь тем, кому нужны специальные возможности.

Можно настроить приложение так, чтобы оно загружало файлы сборки, определив в его конфигурационном файле элемент `codeBase` (см. подробнее главу 3). Этот элемент идентифицирует URL-адрес, по которому можно найти все файлы сборки. При попытке загрузить файл сборки CLR получает URL из элемента `codeBase` и проверяет наличие нужного файла в локальном кэше загруженных файлов. Если он там есть, то он загружается, если нет — CLR использует для загрузки файла в кэш URL-адрес. Если не удастся найти нужный файл, CLR генерирует исключение `FileNotFoundException`.

У меня есть три аргумента в пользу применения многофайловых сборок.

- Можно распределять типы по нескольким файлам, допуская избирательную загрузку необходимых файлов из Интернета, а также частично упаковывать и развертывать типы, варьируя функциональность приложения.

- ❑ Можно добавлять к сборке файлы с ресурсами и данными. Допустим, имеется тип для расчета некоторой страховой суммы. Ему может потребоваться доступ к актуарным таблицам. Вместо встраивания актуарных таблиц в исходный текст можно включить соответствующий файл с данными в состав сборки (например, с помощью компоновщика сборок `AL.exe`, который рассмотрен далее). Можно включать в сборки данные в любом формате: в текстовом, в виде таблиц Microsoft Excel или Microsoft Word, а также в любом другом при условии, что приложение способно анализировать содержимое этого файла.
- ❑ Можно создавать сборки, состоящие из типов, написанных на разных языках программирования. При компиляции исходного текста на языке C# компилятор создает один модуль, а при компиляции исходного текста на Visual Basic — другой. Одна часть типов может быть написана на C#, другая — на Visual Basic, остальные — на других языках программирования. Затем при помощи соответствующего инструмента все эти модули можно объединить в одну сборку. Используя такую сборку разработчики увидят в ней лишь набор типов. Разработчики даже не заметят, что применялись разные языки программирования. Кстати, при желании с помощью `ILDasm.exe` можно получить файлы с исходным текстом всех модулей на языке IL. После этого можно запустить утилиту `ILAsm.exe` и передать ей полученные файлы, и утилита выдаст файл, содержащий все типы. Для этого компилятор исходного текста должен генерировать только IL-код, поэтому эту методику нельзя использовать с Visual C++.

ВНИМАНИЕ

Подводя итог, можно сказать, что сборка — это единица многократного использования, управления версиями и безопасности типов. Она позволяет распределять типы и ресурсы по отдельным файлам, чтобы ее пользователи могли решить, какие файлы упаковывать и развертывать вместе. Загрузив файл с манифестом, среда CLR может определить, какие файлы сборки содержат типы и ресурсы, на которые ссылается приложение. Любому потребителю сборки достаточно знать лишь имя файла, содержащего манифест, после чего он сможет, не нарушая работы приложения, абстрагироваться от особенностей распределения содержимого сборки по файлам, которое со временем может меняться.

При работе со многими типами, совместно использующими одну версию и набор параметров безопасности, по соображениям производительности рекомендуется размещать все типы в одном файле, не распределяя их по нескольким файлам, не говоря уже о разных сборках. На загрузку каждого файла или сборки CLR и Windows тратят значительное время: на поиск сборки, ее загрузку и инициализацию. Чем меньше файлов и сборок, тем быстрее загрузка, потому уменьшение числа сборок способствует сокращению рабочего пространства и степени фрагментации адресного пространства процесса. Ну, и наконец,

nGen.exe лучше оптимизирует код, если обрабатываемые файлы больше по размеру.

Чтобы скомпоновать сборку, нужно выбрать один из PE-файлов, который станет хранителем манифеста. Можно также создать отдельный PE-файл, в котором не будет ничего, кроме манифеста. В табл. 2.3 перечислены таблицы метаданных манифеста, наличие которых превращает управляемый модуль в сборку.

Таблица 2.3. Таблица метаданных манифеста

Имя таблицы метаданных манифеста	Описание
AssemblyDef	Состоит из единственной записи, если модуль идентифицирует сборку. Запись включает имя сборки (без расширения и пути), сведения о версии (старший и младший номера версии, номер компоновки и редакции), региональные стандарты, флаги, хэш-алгоритм и открытый ключ издателя (это поле может быть пустым — null)
FileDef	Содержит по одной записи для каждого PE-файла и файла ресурсов, входящих в состав сборки. В каждой записи содержится имя и расширение файла (без указания пути), хэш и флаги. Если сборка состоит из одного файла, таблица FileDef пуста
ManifestResourceDef	Содержит по одной записи для каждого ресурса, включенного в сборку. Каждая запись включает имя ресурса, флаги (public или private), а также индекс для таблицы FileDef, указывающий файл или поток с ресурсом. Если ресурс не является отдельным файлом (например, JPEG- или GIF-файлом), он хранится в виде потока в составе PE-файла. В случае встроенного ресурса запись также содержит смещение, указывающее начало потока ресурса в PE-файле
ExportedTypesDef	Содержит записи для всех открытых типов, экспортируемых всеми PE-модулями сборки. В каждой записи указано имя типа, индекс для таблицы FileDef (указывающий файл сборки, в котором реализован этот тип), а также индекс для таблицы TypeDef

Манифест позволяет потребителям сборки абстрагироваться от особенностей распределения ее содержимого и делает сборку самоописываемой. Обратите внимание, что в файле, который содержит манифест, находится также информация о том, какие файлы составляют сборку, но отдельные файлы «не знают», что они включены в сборку.

ПРИМЕЧАНИЕ

Файл сборки, содержащий манифест, содержит также таблицу AssemblyRef. В ней хранятся записи с описанием всех сборок, на которые ссылаются файлы данной сборки. Это позволяет инструментам, открыв манифест сборки, сразу увидеть весь набор сборок, на которые ссылается эта сборка, не открывая другие файлы сборки. И в этом случае данные AssemblyRef призваны сделать сборку самоописываемой.

Компилятор C# создает сборку, если указан любой из параметров командной строки — `/t[arget]:exe`, `/t[arget]:winexe` или `/t[arget]:library`. Каждый из этих параметров заставляет компилятор генерировать единый PE-файл с таблицами метаданных манифеста. В итоге генерируется соответственно консольное приложение, приложение с графическим интерфейсом или DLL-файл.

Кроме этих параметров компилятор C# поддерживает параметр `/t[arget]:module`, который заставляет компилятор создать PE-файл без таблиц метаданных. При использовании этого параметра всегда получается DLL-файл в формате PE. Для того чтобы получить доступ к типам такого файла, его необходимо поместить в сборку. При указании параметра `/t:module` компилятор C# по умолчанию присваивает выходному файлу расширение `.netmodule`.

ВНИМАНИЕ

К сожалению, в интегрированной среде разработки (Integrated Development Environment, IDE) Microsoft Visual Studio нет встроенной поддержки создания многофайловых сборок — для этого приходится использовать инструменты командной строки.

Существует несколько способов добавления модуля в сборку. Если PE-файл с манифестом собирается при помощи компилятора C#, можно применить параметр `/addmodule`. Для того чтобы понять, как создают многофайловые сборки, рассмотрим пример. Допустим, есть два файла с исходным текстом:

- ❑ файл `RUT.cs` содержит редко используемые типы;
- ❑ файл `FUT.cs` содержит часто используемые типы.

Скомпилируем редко используемые типы в отдельный модуль, чтобы пользователи сборки могли отказаться от развертывания этого модуля, если содержащиеся в нем типы им не нужны:

```
csc /t:module RUT.cs
```

Команда заставляет компилятор C# создать файл `RUT.netmodule`, который представляет собой стандартную PE-библиотеку DLL, но среда CLR не сможет просто загрузить ее.

Теперь скомпилируем в отдельном модуле часто используемые типы и сделаем его хранителем манифеста сборки, так как к расположенным в нем типам обращаются довольно часто. Фактически теперь этот модуль представляет собой целую сборку, поэтому я изменил имя выходного файла с `FUT.dll` на `JeffTypes.dll`:

```
csc /out:JeffTypes.dll /t:library /addmodule:RUT.netmodule FUT.cs
```

Эта команда приказывает компилятору C# при компиляции файла `FUT.cs` создать файл `JeffTypes.dll`. Поскольку указан параметр `/t:library`, результирующий PE-файл DLL с таблицами метаданных манифеста называется `JeffTypes.dll`.

Параметр `/addmodule:RUT.netmodule` указывает компилятору, что файл `RUT.netmodule` должен быть частью сборки. В частности, параметр `/addmodule` заставляет компилятор добавить к таблице `FileDef` в метаданных манифеста сведения об этом файле, а также занести в таблицу `ExportedTypesDef` сведения об открытых экспортируемых типах этого файла.

Завершив работу, компилятор создаст несколько файлов (рис. 2.1). Модуль справа содержит манифест.

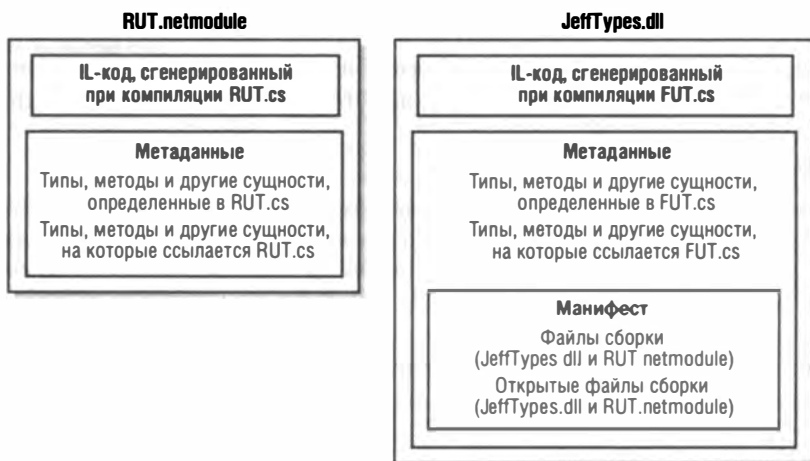


Рис. 2.1. Многофайловая сборка из двух управляемых модулей и манифеста

Файл `RUT.netmodule` содержит IL-код, сгенерированный при компиляции `RUT.cs`. Кроме того, этот файл содержит таблицы метаданных, описывающие типы, методы, поля, свойства, события и т. п., определенные в `RUT.cs`, а также типы, методы и др., на которые ссылается `RUT.cs`. `JeffTypes.dll` — это отдельный файл. Подобно `RUT.netmodule`, он включает IL-код, сгенерированный при компиляции `FUT.cs`, а также аналогичные метаданные в виде таблиц определений и ссылок. Однако `JeffTypes.dll` содержит дополнительные таблицы метаданных, которые и делают его сборкой. Эти дополнительные таблицы описывают все файлы, составляющие сборку (сам файл `JeffTypes.dll` и `RUT.netmodule`). Таблицы метаданных манифеста также включают описание всех открытых типов, экспортируемых файлами `JeffTypes.dll` и `RUT.netmodule`.

ПРИМЕЧАНИЕ

На самом деле в таблицах метаданных манифеста не описаны типы, экспортируемые PE-файлом, в котором находится манифест. Цель этой оптимизации — уменьшить число байт, необходимое для хранения данных манифеста в PE-файле. Таким образом, утверждения вроде «таблицы метаданных манифеста включают все открытые типы, экспортируемые `JeffTypes.dll` и `RUT.netmodule`» верны лишь отчасти. Однако это утверждение абсолютно точно отражает логический набор экспортируемых типов.

Скомпоновав сборку **JeffTypes.dll**, можно изучить ее таблицы метаданных манифеста при помощи **ILDasm.exe**, чтобы убедиться, что файл сборки действительно содержит ссылки на типы из файла **RUT.netmodule**. Если скомпоновать этот проект и затем проанализировать его метаданные при помощи утилиты **ILDasm.exe**, в выводимой информации вы увидите таблицы **FileDef** и **ExportedTypesDef**. Они выглядят следующим образом:

File #1 (26000001)

```
-----  
Token: 0x26000001  
Name : RUT.netmodule  
HashValue Blob : e6 e6 df 62 2c a1 2c 59 97 65 0f 21 44 10 15 96 f2 7e db  
c2  
Flags : [ContainsMetaData] (00000000)
```

ExportedType #1 (27000001)

```
-----  
Token: 0x27000001  
Name: ARarelyUsedType  
Implementation token: 0x26000001  
TypeDef token: 0x02000002  
Flags : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]  
[BeforeFieldInit](00100101)
```

Из этих сведений видно, что **RUT.netmodule** — это файл, который считается частью сборки с маркером **0x26000001**. Таблица **ExportedType** показывает наличие открытого экспортируемого типа **ARarelyUsedType**. Этот тип помечен *маркером реализации* (implementation token) **0x26000001**, означающим, что IL-код этого типа находится в файле **RUT.netmodule**.

ПРИМЕЧАНИЕ

Для любопытных: размер маркеров метаданных — 4 байта. Старший байт указывает тип маркера (**0x01=TypeRef**, **0x02=TypeDef**, **0x26=FileRef**, **0x27=ExportedType**). Полный список типов маркеров см. в перечислимом типе **CorTokenType** в заголовочном файле **CorHdr.h** из .NET Framework SDK. Три младших байта маркера просто идентифицируют запись в соответствующей таблице метаданных. Например, маркер реализации **0x26000001** ссылается на первую строку таблицы **FileRef** (нумерация строк начинается с 1, а не с 0). Кстати, в **TypeDef** нумерация строк начинается с 2.

Любой клиентский код, использующий типы сборки **JeffTypes.dll**, должен компоноваться с указанием параметра компилятора **/r[еference]:JeffTypes.dll**, который заставляет компилятор загрузить сборку **JeffTypes.dll** и все файлы, перечисленные в ее таблице **FileDef**. Компилятору необходимо, чтобы все файлы сборки были установлены и доступны. Если бы мы удалили файл

RUT.netmodule, компилятор C# сгенерировал бы следующее сообщение об ошибке:

```
fatal error CS0009: Metadata file 'C:\JeffTypes.dll' could not be opened—'Error
importing module 'rut.netmodule' of assembly 'C:\JeffTypes.dll'— The system
cannot find the file specified
```

Это означает, что при компоновке новой сборки *должны* присутствовать все файлы, на которые она ссылается.

Во время исполнения клиентский код вызывает разные методы. При первом вызове некоторого метода среда CLR определяет, на какие типы он ссылается как на параметр, возвращаемое значение или локальную переменную. Далее CLR пытается загрузить из сборки, на которую ссылается код, файл с манифестом. Если этот файл описывает типы, к которым обращается вызванный метод, срабатывают внутренние механизмы CLR, и нужные типы становятся доступными. Если в манифесте указано, что нужный тип находится в другом файле, CLR загружает этот файл, и внутренние механизмы CLR обеспечивают доступ к данному типу. CLR загружает файл сборки только при вызове метода, ссылающегося на расположенный в этом файле тип. Это значит, что наличие всех файлов сборки, на которую ссылается приложение, для его работы *не обязательно*.

Добавление сборок в проект в среде Visual Studio

Если проект создается в среде Visual Studio, необходимо добавить в проект все сборки, на которые он ссылается. Для этого откройте окно Solution Explorer, щелкните правой кнопкой мыши на проекте, на который нужно добавить ссылку, и выберите команду Add Reference. Откроется диалоговое окно Add Reference (рис. 2.2).

Для того чтобы добавить в проект ссылки на сборку, выберите ее в списке. Если в списке нет нужной сборки, то для того чтобы ее найти (файл с манифестом), щелкните на кнопке Browse. Вкладка COM в диалоговом окне Add Reference позволяет получить доступ к неуправляемому COM-серверу из управляемого кода через класс-представитель, автоматически генерируемый Visual Studio. Вкладка Projects служит для добавления в текущий проект ссылки на сборки, созданные в другом проекте этого же решения.

Чтобы сборки отображались в списке на вкладке .NET, добавьте в реестр подраздел:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ .NETFramework\AssemblyFolders\MyLibName
```

MyLibName — это созданное разработчиком уникальное имя, Visual Studio его не отображает. Создав такой подраздел, измените его строковое значение по умолчанию, чтобы оно указывало на каталог, в котором хранятся файлы сборок (например, C:\Program Files\MyLibPath).

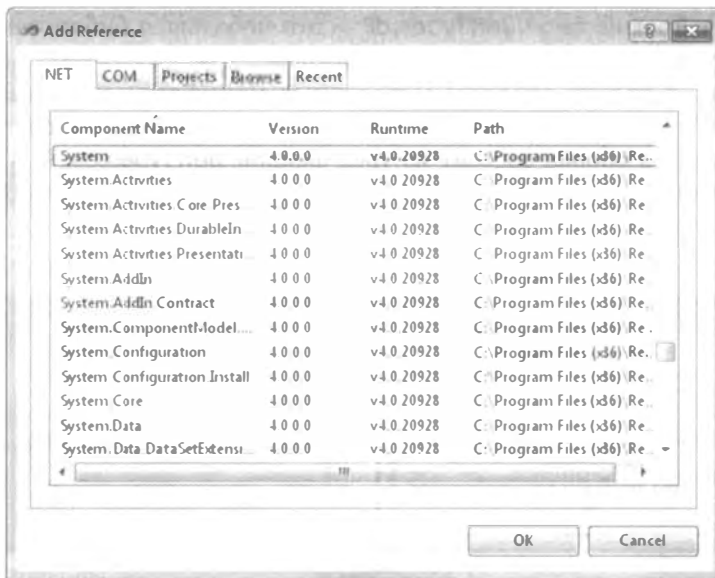


Рис. 2.2. Диалоговое окно Add Reference в Visual Studio

Использование утилиты Assembly Linker

Вместо компилятора C# для создания сборки можно задействовать компоновщик сборок (assembly linker) AL.exe. Эта утилита оказывается кстати, если нужно создавать сборки из модулей, скомпонованных разными компиляторами (если компилятор языка не поддерживает параметр, эквивалентный параметру /addmodule из C#), а также в случае, когда требования к упаковке сборки на момент компоновки просто не известны. Утилита AL.exe пригодна и для компоновки сборок, состоящих исключительно из ресурсов (или сопутствующих сборок — к ним мы еще вернемся), которые обычно используются для локализации ПО.

Утилита AL.exe может генерировать файлы формата EXE или DLL PE, которые не содержат ничего, кроме манифеста, описывающего типы из других модулей. Чтобы понять, как работает AL.exe, скомпонуем сборку JeffTypes.dll по-другому:

```
csc /t:module RUT.cs
csc /t:module FUT.cs
al /out:JeffTypes.dll /t:library FUT.netmodule RUT.netmodule
```

Файлы, генерируемые в результате исполнения этих команд, показаны на рис. 2.3.

В этом примере два из трех отдельных модулей, RUT.netmodule и FUT.netmodule, сборками не являются (так как не содержат таблиц метаданных

манифеста). Третий же — **JeffTypes.dll** — это небольшая библиотека PE DLL (поскольку она скомпонована с параметром `/t[target]:library`), в которой нет IL-кода, а только таблицы метаданных манифеста, указывающие, что файлы **JeffTypes.dll**, **RUT.netmodule** и **FUT.netmodule** входят в состав сборки. Результирующая сборка состоит из трех файлов: **JeffTypes.dll**, **RUT.netmodule** и **FUT.netmodule**, так как компоновщик сборок не «умеет» объединять несколько файлов в один.

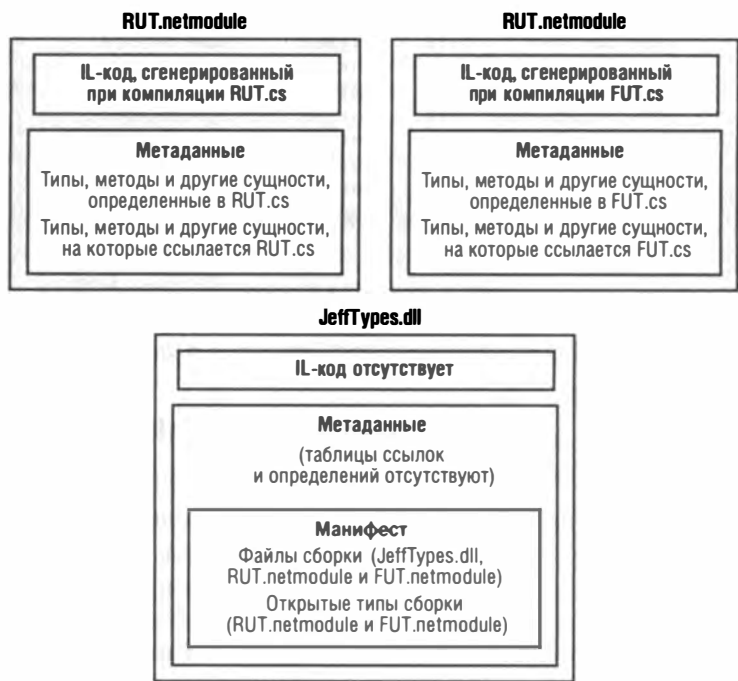


Рис. 2.3. Многофайловая сборка из трех управляемых модулей и манифеста

Утилита **AL.exe** может генерировать консольные PE-файлы и PE-файлы с графическим интерфейсом (с помощью параметра `/t[target]:exe` или `/t[target]:winexe`). Однако это довольно необычно, поскольку означает, что будет сгенерирован исполняемый PE-файл, содержащий не больше IL-кода, чем нужно для вызова метода из другого модуля. Можно указать, какой метод должен использоваться в качестве входной точки, задав при вызове компоновщика сборок параметр командной строки `/main`. Приведем пример вызова **AL.exe** с этим параметром:

```
csc /t:module App.cs al /out:App.exe /t:exe /main:Program.Main app.netmodule
```

Первая строка компоует **App.cs** в модуль, а вторая генерирует небольшой PE-файл **App.exe** с таблицами метаданных манифеста. В нем также находится

небольшая глобальная функция, сгенерированная **AL.exe** благодаря параметру `/main: App.Main`. Эта функция, `__EntryPoint`, содержит следующий IL-код:

```
.method privatescope static void __EntryPoint$PST06000001() cil managed
{
    .entrypoint
    // Code size      8 (0x8)
    .maxstack 8
    IL_0000: tail.
    IL_0002: call     void [module 'Program.netmodule']Program::Main()
    IL_0007: ret
} // end of method 'Global Functions':::__EntryPoint
```

Как видите, этот код просто вызывает метод `Main`, содержащийся в типе `Program`, который определен в файле **App.netmodule**. Параметр `/main`, указанный при вызове **AL.exe**, здесь не слишком полезен, так как вряд ли вы когда-либо будете создавать приложение, у которого точка входа расположена не в PE-файле с таблицами метаданных манифеста. Здесь этот параметр упомянут лишь для того, чтобы вы знали о его существовании.

В программном коде для данной книги имеется файл **Ch02-3-BuildMultiFileLibrary.bat**, в котором инкапсулированы последовательно все шаги, показывающие, как создать многофайловую сборку. **Ch02-4-AppUsingMultiFileLibrary project** в Visual Studio выполняет данный файл на этапе предварительной сборки. Вы можете изучить этот пример, чтобы понять, как интегрировать многофайловую сборку из Visual Studio.

Включение в сборку файлов ресурсов

Если сборка создается при помощи **AL.exe**, параметр `/embed[resource]` позволяет добавить в сборку файлы ресурсов (файлы в формате, отличном от PE). Параметр принимает любой файл и включает его содержимое в результирующий PE-файл. Таблица `ManifestResourceDef` в манифесте обновляется, отражая наличие нового ресурса.

Утилита **AL.exe** поддерживает также параметр `/link[resource]`, который принимает файл с ресурсами. Однако параметр только обновляет таблицы манифеста `ManifestResourceDef` и `FileDef` сведениями о ресурсе и о том, в каком файле сборки он находится. Сам файл с ресурсами не внедряется в PE-файл сборки, а хранится отдельно и подлежит упаковке и развертыванию вместе с остальными файлами сборки.

Подобно **AL.exe**, **CSC.exe** позволяет объединять ресурсы со сборкой, генерируемой компилятором C#. Параметр `/resource` компилятора C# включает указанный файл с ресурсами в результирующий PE-файл сборки и обновляет таблицу `ManifestResourceDef`. Параметр компилятора `/linkresource` добавляет в таблицы `ManifestResourceDef` и `FileDef` записи со ссылкой на отдельный файл с ресурсами.

И последнее: в сборку можно включить стандартные ресурсы Win32. Это легко сделать, указав при вызове AL.exe или CSC.exe путь к RES-файлу и параметр /win32res. Кроме того, можно легко включить стандартный ресурс значка Win32 в файл сборки, указав при вызове AL.exe или CSC.exe путь к ICO-файлу и параметр /win32icon. В Visual Studio файл ресурсов добавляют в сборку на вкладке Application в диалоговом окне свойств проекта. Обычно значки включают, чтобы Проводник Windows (Windows Explorer) мог отображать значок для управляемого исполняемого файла.

ПРИМЕЧАНИЕ

В файлах управляемой сборки содержится также файл манифеста Win32. По умолчанию компилятор C# автоматически создает файл манифеста, однако ему можно запретить это делать при помощи параметра /nowin32manifest. Программный код манифеста, генерируемого компилятором C# по умолчанию, выглядит следующим образом:

```
// Создание и инициализация массива String
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity version="1.0.0.0" name="MyApplication.app" />
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
<security>
<requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
<requestedExecutionLevel level="asInvoker" uiAccess="false"/>
</requestedPrivileges>
</security>
</trustInfo>
</assembly>
```

Ресурсы со сведениями о версии сборки

Когда утилита AL.exe или CSC.exe генерирует сборку в виде PE-файла, она также включает в этот файл стандартную Win32-версию. Пользователи могут увидеть версию, просматривая свойства файла. Для получения этой информации из программы служит статический метод GetVersionInfo типа System.Diagnostics.FileVersionInfo. На рис. 2.4 показана вкладка Details диалогового окна свойств файла JeffTypes.dll.

При компоновке сборки следует задавать значения полей версии в исходном тексте программы с помощью специализированных атрибутов, применяемых на уровне сборки. Вот как выглядит код, генерирующий информацию о версии, показанную на рис. 2.4.



Рис. 2.4. Вкладка Details диалогового окна свойств файла JeffTypes.dll

using System.Reflection;

```
// Задать значения поля FileDescription:  
[assembly: AssemblyTitle("JeffTypes.dll")]
```

```
// Задать значения поля Comments:  
[assembly: AssemblyDescription("This assembly contains Jeff's types")]
```

```
// Задать значения поля CompanyName:  
[assembly: AssemblyCompany("Wintellect")]
```

```
// Задать значения поля ProductName:  
[assembly: AssemblyProduct("Wintellect (R) Jeff's Type Library")]
```

```
// Задать значения поля LegalCopyright:  
[assembly: AssemblyCopyright("Copyright (c) Wintellect 2010")]
```

```
// Задать значения поля LegalTrademarks:  
[assembly: AssemblyTrademark("JeffTypes is a registered trademark of  
Wintellect")]
```

продолжение ➤


```
// Задать значения поля AssemblyVersion:
[assembly: AssemblyVersion("3.0.0.0")]

// Задать значения поля AssemblyFileVersion:
[assembly: AssemblyFileVersion("1.0.0.0")]

// Задать значения поля ProductVersion:
[assembly: AssemblyInformationalVersion("2.0.0.0")]

// Задать значения поля (подробнее см. раздел "Региональные стандарты")
[assembly: AssemblyCulture("")]
```

ВНИМАНИЕ

К сожалению, в диалоговом окне свойств Проводника Windows отсутствуют поля для некоторых атрибутов. В частности, было бы замечательно, если бы был отображен атрибут AssemblyVersion, потому что среда CLR использует значение этого атрибута при загрузке сборки (об этом рассказано в главе 3).

В табл. 2.4 перечислены поля ресурса со сведениями о версии и соответствующие им атрибуты, определяемые пользователем. Если сборка компонуется утилитой AL.exe, сведения о версии можно задать, применяя параметры командной строки вместо атрибутов. Во втором столбце табл. 2.4 показаны параметры командной строки для каждого поля ресурса со сведениями о версии. Обратите внимание на отсутствие аналогичных параметров у компилятора C#, поэтому сведения о версии обычно задают, применяя специализированные атрибуты.

Таблица 2.4. Поля ресурса со сведениями о версии и соответствующие им параметры AL.exe и пользовательские атрибуты

Поле ресурса со сведениями о версии	Параметр AL.exe	Атрибут/комментарий
FILEVERSION	/fileversion	System.Reflection.AssemblyFileVersionAttribute
PRODUCTVERSION	/productversion	System.Reflection.AssemblyInformationalVersionAttribute
FILEFLAGSMASK	Нет	Всегда задается равным VS_FF_FILEFLAGSMASK (определяется в WinVer.h как 0x0000003F)
FILEFLAGS	Нет	Всегда равен 0
FILEOS	Нет	В настоящее время всегда равен VOS__WINDOWS32

Поле ресурса со сведениями о версии	Параметр AL.exe	Атрибут /комментарий
FILETYPE	/target	Задается равным VFT_APP, если задан параметр /target.exe или /target:winexe. При наличии параметра /target:library приравнивается VFT_DLL
FILESUBTYPE	Нет	Всегда задается равным VFT2_UNKNOWN (это поле не имеет значения для VFT_APP и VFT_DLL)
AssemblyVersion	/version	System.Reflection. AssemblyVersionAttribute
Comments	/description	System.Reflection. AssemblyDescriptionAttribute
CompanyName	/company	System.Reflection. AssemblyCompanyAttribute
FileDescription	/title	System.Reflection.AssemblyTitleAttribute
FileVersion	/version	System.Reflection. AssemblyVersionAttribute
InternalName	/out	Задается равным заданному имени выходного файла (без расширения)
LegalCopyright	/copyright	System.Reflection. AssemblyCopyrightAttribute
LegalTrademarks	/trademark	System.Reflection. AssemblyTrademarkAttribute
OriginalFilename	/out	Задается равным заданному имени выходного файла (без пути)
PrivateBuild	Нет	Всегда остается пустым
ProductName	/product	System.Reflection. AssemblyProductAttribute
ProductVersion	/productversion	System.Reflection.AssemblyInformational-VersionAttribute
SpecialBuild	Нет	Всегда остается пустым

ВНИМАНИЕ

При создании нового проекта C# в Visual Studio файл AssemblyInfo.cs генерируется автоматически. Он содержит все атрибуты сборки, описанные в этом разделе, а также несколько дополнительных — о них речь идет в главе 3. Можно просто открыть файл AssemblyInfo.cs и изменить относящиеся к конкретной сборке сведения. Visual Studio также предоставляет диалоговое окно для редактирования информации о версии сборки, которое представлено на рис. 2.5.

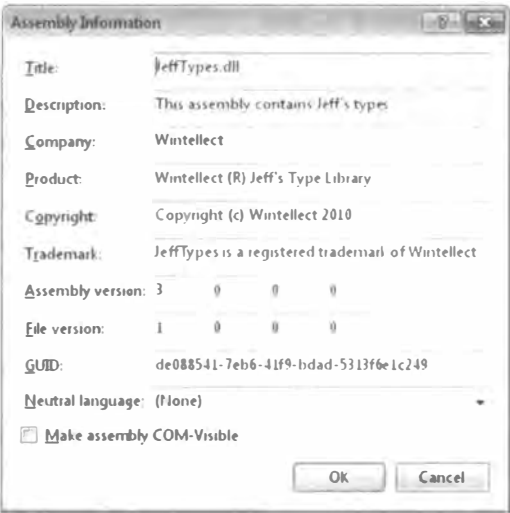


Рис. 2.5. Диалоговое окно с информацией о сборке в Visual Studio

Номера версии

Ранее было показано, что сборка может идентифицироваться по номеру версии. У частей этого номера одинаковый формат: каждая состоит из 4 частей, разделенных точками (табл. 2.5).

Таблица 2.5. Формат номеров версии

	Старший номер	Младший номер	Номер компоновки	Номер редакции
Пример	2	5	719	2

В табл. 2.5 показан пример номера версии 2.5.719.2. Первые две цифры составляют то, что обычно понимают под номером версии: пользователи будут считать номером версии 2.5. Третье число, 719, указывает номер компоновки. Если в вашей компании сборка компонуется каждый день, увеличивать этот номер надо ежедневно. Последнее число 2 — номер редакции сборки. Если в компании сборка компонуется дважды в день (скажем, после исправления критической и обязательной к немедленному исправлению ошибки, тормозившей всю работу над проектом), надо увеличивать номер редакции.

Такая схема нумерации версий принята в компании Microsoft, и я настоятельно рекомендую ей следовать. В следующих версиях CLR предполагается обеспечить более совершенную поддержку механизма загрузки новых версий сборок и отката к предыдущим версиям сборки, если новая несовместима с имеющимся приложением. В рамках механизма поддержки версий среда CLR будет ожидать, что у версии сборки, в которой устранены определенные неполадки, будут те же старший и младший номера версий, а номера компоновки

и редакции будут указывать на *служебную версию* (servicing version) с исправлениями. При загрузке сборки CLR будет автоматически искать самую последнюю служебную версию (с теми же старшим и младшим номерами версий) нужной сборки.

Со сборкой ассоциированы три номера версии. Это очень неудачное решение стало источником серьезной путаницы. Попробую объяснить, для чего нужен каждый из этих номеров и как его правильно использовать.

- ❑ **AssemblyFileVersion** — этот номер версии хранится в ресурсе версии Win32 и предназначен лишь для информации, CLR его полностью игнорирует. Обычно устанавливают старший и младший номера версии, определяющие отображаемый номер версии. Далее при каждой компоновке увеличивают номер компоновки и редакции. В идеале инструмент от компании Microsoft (например, CSC.exe или AL.exe) должен автоматически обновлять номера компоновки и редакции (в зависимости от даты и времени на момент компоновки), но этого не происходит. Этот номер версии отображается Проводником Windows и служит для определения точного времени компоновки сборки.
- ❑ **AssemblyInformationalVersion** — этот номер версии также хранится в ресурсе версии Win32 и, как и предыдущий, предназначен лишь для информации; для CLR он абсолютно безразличен. Этот номер служит для указания версии продукта, в который входит сборка. Например, продукт версии 2.0 может состоять из нескольких сборок. Одна из них может отмечаться как версия 1.0, если это новая сборка, не входившая в комплект поставки продукта версии 1.0. Обычно отображаемый номер версии формируется при помощи старшего и младшего номеров версии. Затем номера компоновки и редакции увеличивают при каждой упаковке всех сборок готового продукта.
- ❑ **AssemblyVersion** — этот номер версии хранится в манифесте, в таблице метаданных AssemblyDef. CLR использует этот номер версии для привязки к сборкам, имеющим строгие имена (о них рассказано в главе 3). Этот номер версии чрезвычайно важен и уникально идентифицирует сборку. Начиная разработку сборки, следует задать старший и младший номера версии, а также номера компоновки и редакции; не меняйте их, пока не будете готовы начать работу над следующей версией сборки, пригодной для развертывания. При создании сборки, ссылающейся на другую, этот номер версии включается в нее в виде записи таблицы AssemblyRef. Это значит, что сборка жестко привязана к конкретной версии.

Региональные стандарты

Помимо номера версии, сборки идентифицируют *региональными стандартами* (culture). Например, одна сборка может быть исключительно на немецком языке,

другая — на швейцарском варианте немецкого, третья — на американском английском и т. д. Региональные стандарты идентифицируются строкой, содержащей основной и вспомогательный теги (как описано в RFC1766). Несколько примеров приведено в табл. 2.6.

Таблица 2.6. Примеры тегов, определяющих региональные стандарты сборки

Основной тег	Вспомогательный тег	Региональные стандарты
de	Нет	Немецкий
de	AT	Австрийский немецкий
de	CH	Швейцарский немецкий
en	Нет	Английский
en	GB	Английский
en	US	Английский

В общем случае сборкам с кодом не назначают региональные стандарты, так как код обычно не содержит зависящих от них встроенных параметров. Сборку, для которой не определен региональный стандарт, называют сборкой с *нейтральными региональными стандартами* (culture neutral).

При создании приложения, ресурсы которого привязаны к региональным стандартам, компания Microsoft настоятельно рекомендует объединять программный ресурс и ресурсы приложения по умолчанию в одной сборке и не назначать ей региональных стандартов при компоновке. Другие сборки будут ссылаться на нее при создании и работе с типами, которые она предоставляет для общего доступа.

После этого можно создать одну или несколько отдельных сборок, содержащих только ресурсы, зависящие от региональных стандартов, и никакого программного кода. Сборки, помеченные для применения в определенных региональных стандартах, называют *сопутствующими* (satellite assemblies). Региональные стандарты, назначенные такой сборке, в точности отражают региональные стандарты размещенного в ней ресурса. Следует создавать отдельные сборки для каждого регионального стандарта, который планируется поддерживать.

Обычно сопутствующие сборки komponуются при помощи утилиты AL.exe. Не стоит использовать для этого компилятор — ведь в сопутствующей сборке не должно быть программного кода. Применяя утилиту AL.exe, можно задать желаемые региональные стандарты параметром /c[ulture]:text, где text — это строка (например, en-US, представляющая американский вариант английского языка). При развертывании сопутствующие сборки следует помещать в подкаталог, имя которого совпадает с текстовой строкой, идентифицирующей региональные стандарты. Например, если базовым каталогом приложения является C:\MyApp, сопутствующая сборка для американского варианта английского

языка должна быть в каталоге `C:\MyApp\en-US`. Во время выполнения доступ к ресурсам сопутствующей сборки осуществляют через класс `System.Resources.ResourceManager`.

ПРИМЕЧАНИЕ

Хотя это и не рекомендуется, можно создавать сопутствующие сборки с программным кодом. При желании вместо параметра `/culture` утилиты `AL.exe` региональный стандарт можно указать в атрибуте `System.Reflection.AssemblyCulture`, определяемом пользователем, например, следующим образом:

```
// Назначить для сборки региональный стандарт Swiss German  
[assembly:AssemblyCulture("de-CH")]
```

Лучше не создавать сборки, ссылающиеся на сопутствующие сборки. Другими словами, все записи таблицы `AssemblyRef` должны ссылаться на сборки с нейтральными региональными стандартами. Если необходимо получить доступ к типам или членам, расположенным в сопутствующей сборке, следует воспользоваться методом отражения (см. главу 23).

Развертывание простых приложений (закрытое развертывание сборок)

Ранее в этой главе было показано, как компоновать модули и объединять их в сборки. Теперь можно приступить к изложению того, как упаковывать и развертывать сборки, чтобы пользователь мог работать с приложением.

Особых средств для упаковки сборки не требуется. Легче всего упаковать набор сборок, просто скопировав все их файлы. Например, можно поместить все файлы сборки на компакт-диск и передать их пользователю вместе с программой установки, написанной в виде пакетного файла. Такая программа просто копирует файлы с компакт-диска в каталог на жестком диске пользователя. Поскольку сборка включает все ссылки и типы, определяющие ее работу, ему достаточно запустить приложение, а CLR найдет в каталоге приложения все сборки, на которые ссылается данная сборка. Так что для работы приложения не нужно модифицировать реестр, а чтобы удалить приложение, достаточно просто удалить его файлы — и все!

Конечно, можно применять для упаковки и установки сборок другие механизмы, например CAB-файлы (они обычно используются в сценариях с загрузкой из Интернета для сжатия файлов и сокращения времени загрузки). Можно также упаковать файлы сборки в MSI-файл, предназначенный для службы установщика Windows (Windows Installer), `MSIExec.exe`. MSI позволяет установить сборку по требованию при первой попытке CLR ее загрузить.

Эта функция не нова для службы MSI, она также поддерживает аналогичную функцию для неуправляемых EXE- и DLL-файлов.

ПРИМЕЧАНИЕ

Пакетный файл или подобная простая «установочная программа» скопирует приложение на машину пользователя, однако для создания ярлыков на рабочем столе, в меню Пуск (Start) и на панели быстрого запуска понадобится программа посложнее. Кроме того, скопировать, восстановить или переместить приложение с одной машины на другую легко, но ссылки и ярлыки потребуют специального обращения.

Естественно, в Visual Studio есть встроенные механизмы, которые можно задействовать для публикации приложений, — это делается на вкладке Publish страницы свойств проекта. Можно использовать ее, чтобы заставить Visual Studio создать MSI-файл и скопировать его на веб-сайт, FTP-сайт или в заданную папку на диске. MSI-файл также может установить все необходимые компоненты, такие как .NET Framework или Microsoft SQL Server 2005 Express Edition. Наконец, приложение может автоматически проверять наличие обновлений и устанавливать их на пользовательской машине посредством технологии ClickOnce.

Сборки, развертываемые в том же каталоге, что и приложение, называют *сборками с закрытым развертыванием* (privately deployed assemblies), так как файлы сборки не используются совместно другими приложениями (если только другие приложения не развертывают в том же каталоге). Сборки с закрытым развертыванием — серьезное преимущество для разработчиков, конечных пользователей и администраторов, поскольку достаточно скопировать такие сборки в базовый каталог приложения, и CLR сможет загрузить и исполнить содержащийся в них код. Кроме того, легко удалить приложение, просто удалив сборки из его каталога. Также легко создавать резервные копии подобных сборок и восстанавливать их.

Несложный сценарий установки/перемещения/удаления приложения стал возможным благодаря наличию в каждой сборке метаданных. Метаданные указывают, какую сборку, на которую они ссылаются, нужно загрузить — для этого не нужны параметры реестра. Кроме того, область видимости сборки охватывает все типы. Это значит, что приложение всегда привязывается именно к тому типу, с которым оно было скомпоновано и протестировано. CLR не может загрузить другую сборку просто потому, что она предоставляет тип с тем же именем. Этим CLR отличается от COM, где типы регистрируются в системном реестре, что делает их доступными любому приложению, работающему на машине.

В главе 3 рассказано о развертывании совместно используемых сборок, доступных нескольким приложениям.

Простое средство администрирования (конфигурационный файл)

Пользователи и администраторы лучше всех могут определять разные аспекты работы приложения. Например, администратор может решить переместить файлы сборки на жесткий диск пользователя или заменить данные в манифесте сборки. Есть и другие сценарии управления версиями и удаленного администрирования, о некоторых из них рассказано в главе 3.

Для того чтобы предоставить администратору контроль над приложением, можно разместить в каталоге приложения конфигурационный файл. Его может создать и упаковать издатель приложения, после чего программа установки запишет конфигурационный файл в базовый каталог приложения. Кроме того, администратор или конечный пользователь машины может сам создать или модифицировать этот файл. CLR интерпретирует его содержимое для изменения политики поиска и загрузки файлов сборки.

Конфигурационные файлы содержат XML-теги и могут быть ассоциированы с приложением или с компьютером. Использование отдельного файла (вместо параметров, хранимых в реестре) позволяет легко создать резервную копию файла, а администратору — без труда копировать файлы с машины на машину: достаточно скопировать нужные файлы — в результате будет также скопирована административная политика.

В главе 3 такой конфигурационный файл рассматривается подробно, а пока кратко обсудим его. Допустим, издатель хочет развернуть приложение вместе с файлами сборки JeffTypes, но в отдельном каталоге. Желаемая структура каталогов с файлами выглядит следующим образом:

Каталог AppDir (содержит файлы сборки приложения)

App.exe

App.exe.config (обсуждается ниже)

Подкаталог AuxFiles (содержит файлы сборки JeffTypes)

JeffTypes.dll

FUT.netmodule

RUT.netmodule

Поскольку файлы сборки JeffTypes более не находятся в базовом каталоге приложения, CLR не сможет найти и загрузить их, и при запуске приложения будет сгенерировано исключение `System.IO.FileNotFoundException`. Чтобы избежать этого, издатель создает конфигурационный файл в формате XML и размещает его в базовом каталоге приложения. Имя этого файла должно совпадать с именем главного файла сборки и иметь расширение `config`, в данном

случае — **App.exe.config**. Содержимое этого конфигурационного файла должно выглядеть примерно следующим образом:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="AuxFiles" />
    </assemblyBinding>
  </runtime>
</configuration>
```

Пытаясь найти файл сборки, CLR всегда сначала ищет в каталоге приложения, и если поиск заканчивается неудачей, продолжает искать в подкаталоге **AuxFiles**. В атрибуте `privatePath` элемента, направляющего поиск, можно указать несколько путей, разделенных точкой с запятой. Считается, что все пути заданы относительно базового каталога приложения. Идея здесь в том, что приложение может управлять своим каталогом и его подкаталогами, но не может управлять другими каталогами.

Алгоритм поиска файлов сборки

В поиске сборки среда CLR просматривает несколько подкаталогов. Порядок при поиске сборки с нейтральными региональными стандартами таков (при условии, что параметры `firstPrivatePath` и `secondPrivatePath` определены в атрибуте `privatePath` конфигурационного файла):

```
AppDir\AsmName.dll AppDir\AsmName\AsmName.dll AppDir\firstPrivatePath\
AsmName.dll AppDir\firstPrivatePath\AsmName\AsmName.dll AppDir\
secondPrivatePath\AsmName.dll AppDir\secondPrivatePath\AsmName\
AsmName.dll
...
```

В этом примере конфигурационный файл вовсе не понадобится, если файлы сборки **JeffTypes** развернуты в подкаталоге **JeffTypes**, так как CLR автоматически проверяет подкаталог, имя которого совпадает с именем искомой сборки.

Если ни в одном из упомянутых каталогов сборка не найдена, CLR начинает поиск заново, но теперь ищет файл с расширением **EXE** вместо **DLL**. Если и на этот раз поиск оканчивается неудачей, генерируется исключение `FileNotFoundException`.

В отношении сопутствующих сборок действуют те же правила поиска за одним исключением: ожидается, что сборка находится в подкаталоге базового каталога приложения, имя которого совпадает с названием региональных стандартов. Например, если для файла **AsmName.dll** назначен региональный стандарт «en-US», порядок просмотра каталогов таков:

```
C:\AppDir\en-US\AsmName.dll
C:\AppDir\en-US\AsmName\AsmName.dll
```

```
C:\AppDir\firstPrivatePath\en-US\AsmName.dll
C:\AppDir\firstPrivatePath\en-US\AsmName\AsmName.dll
C:\AppDir\secondPrivatePath\en-US\AsmName.dll
C:\AppDir\secondPrivatePath\en-US\AsmName\AsmName.dll
C:\AppDir\en-US\AsmName.exe
C:\AppDir\en-US\AsmName\AsmName.exe
C:\AppDir\firstPrivatePath\en-US\AsmName.exe
C:\AppDir\firstPrivatePath\en-US\AsmName\AsmName.exe
C:\AppDir\secondPrivatePath\en-US\AsmName.exe
C:\AppDir\secondPrivatePath\en-US\AsmName\AsmName.exe
C:\AppDir\en\AsmName.dll
C:\AppDir\en\AsmName\AsmName.dll
C:\AppDir\firstPrivatePath\en\AsmName.dll
C:\AppDir\firstPrivatePath\en\AsmName\AsmName.dll
C:\AppDir\secondPrivatePath\en\AsmName.dll
C:\AppDir\secondPrivatePath\en\AsmName\AsmName.dll
C:\AppDir\en\AsmName.exe
C:\AppDir\en\AsmName\AsmName.exe
C:\AppDir\firstPrivatePath\en\AsmName.exe
C:\AppDir\firstPrivatePath\en\AsmName\AsmName.exe
C:\AppDir\secondPrivatePath\en\AsmName.exe
C:\AppDir\secondPrivatePath\en\AsmName\AsmName.exe
```

Как видите, CLR ищет файлы с расширением EXE или DLL. Поскольку поиск может занимать значительное время (особенно когда CLR пытается найти файлы в сети), в конфигурационном XML-файле можно указать один или несколько элементов региональных стандартов, чтобы ограничить круг проверяемых каталогов при поиске сопутствующих сборок.

Имя и расположение конфигурационного XML-файла может различаться в зависимости от типа приложения.

- ❑ Для исполняемых приложений (EXE) конфигурационный файл должен располагаться в базовом каталоге приложения. У него должно быть то же имя, что и у EXE-файла, но с расширением config.
- ❑ Для приложений ASP.NET Web Form конфигурационный файл всегда должен находиться в виртуальном корневом каталоге веб-приложения и называться **Web.config**. Кроме того, в каждом вложенном каталоге может быть собственный файл **Web.config** с унаследованными параметрами конфигурации. Например, веб-приложение, расположенное по адресу <http://www.Wintellect.com/Training>, будет использовать параметры из файлов **Web.config**, расположенных в виртуальном корневом каталоге и в подкаталоге **Training**.

Как уже было сказано, параметры конфигурации применяют к конкретному приложению и конкретному компьютеру. При установке платформа .NET Framework создает файл **Machine.config**. Существует по одному файлу **Machine.config**

на каждую версию среды CLR, установленную на данной машине. Файл **Machine.config** расположен в следующем каталоге:

`%SystemRoot%\Microsoft.NET\Framework\версия\CONFIG`

Естественно, `%SystemRoot%` — это каталог, в котором установлена система Windows (обычно `C:\Windows`), а *версия* — номер версии, идентифицирующий определенную версию платформы .NET Framework (например, `v4.0.####`).

Параметры файла **Machine.config** заменяют параметры конфигурационного файла конкретного приложения. Администраторам и пользователям следует избегать модификации файла **Machine.config**, поскольку в нем хранятся многие параметры, связанные с самыми разными аспектами работы системы, что серьезно затрудняет ориентацию в его содержимом. Кроме того, требуется резервное копирование и восстановление конфигурационных параметров приложения, что возможно лишь при использовании конфигурационных файлов, специфичных для приложения.

Глава 3. Совместно используемые сборки и сборки со строгим именем

В главе 2 рассказано о компоновке, упаковке и развертывании сборок. При этом основное внимание уделено *закрытому развертыванию* (private deployment), при котором сборки, предназначенные исключительно для одного приложения, помещают в базовый каталог приложения или в его подкаталог. Закрытое развертывание сборок дает компаниям большие возможности в плане управления именованием, версиями и особенностями работы сборок.

В этой главе внимание сосредоточено на создании сборок, которые могут совместно использоваться несколькими приложениями. Замечательный пример глобально развертываемых сборок — это сборки, поставляемые вместе с Microsoft .NET Framework, поскольку почти все управляемые приложения используют типы, определенные Microsoft в библиотеке классов .NET Framework Class Library (FCL).

Как уже было отмечено в главе 2, операционная система Windows получила репутацию нестабильной главным образом из-за того, что для создания и тестирования приложений приходится использовать чужой код. В конце концов, любое приложение для Windows, которое вы пишете, вызывает код, созданный разработчиками Microsoft. Более того, самые разные компании производят элементы управления, которые разработчики затем встраивают в свои приложения. Фактически такой подход стимулирует сама платформа .NET Framework, а со временем, вероятно, число производителей элементов управления возрастет.

Время не стоит на месте, как и разработчики из Microsoft, как и сторонние производители элементов управления: они устраняют ошибки, добавляют в свой код новые возможности и т. п. В конечном счете, на жесткий диск пользовательского компьютера попадает новый код. В результате давно установленное и прекрасно работавшее пользовательское приложение начинает задействовать уже не тот код, с которым оно создавалось и тестировалось. И поведение такого приложения становится непредсказуемым, что, в свою очередь, негативно влияет на стабильность Windows.

Решить проблему управления версиями файлов чрезвычайно трудно. На самом деле, я готов спорить, что если взять любой файл и изменить в нем значение одного-единственного бита с 0 на 1 или наоборот, то никто не сможет гарантировать, что программы, использовавшие исходную версию этого файла,

будут работать с новой версией файла, как ни в чем не бывало. Это утверждение верно хотя бы потому, что множество программ случайно или преднамеренно учитывает ошибки других программ. Если в более поздней версии кода исправляется какая-либо ошибка, то использующее его приложение начинает работать непредсказуемо.

Итак, вопрос в следующем: как, устраняя ошибки и добавляя к программам новые функции, гарантировать, что эти изменения не нарушат работу других приложений? Я долго думал над этим и пришел к выводу — это просто невозможно. Но, очевидно, такой ответ не устроит никого, поскольку в поставляемых файлах всегда будут ошибки, а разработчики всегда будут одержимы желанием добавлять новые функции. Должен все же быть способ распространения новых файлов, позволяющий надеяться, что любое приложение после обновления продолжит замечательно работать, а если нет, то позволяющий *легко* вернуть приложение в последнее состояние, в котором оно прекрасно работало.

В этой главе рассказано об инфраструктуре .NET Framework, призванной решить проблемы управления версиями. Позвольте сразу предупредить: речь идет о сложных материях. Нам придется рассмотреть массу алгоритмов, правил и политик, встроенных в общезыковую исполняющую среду (CLR). Помимо этого, упомянуты многие инструменты и утилиты, которыми приходится пользоваться разработчику. Все это достаточно сложно, поскольку, как я уже сказал, проблема управления версиями непростая сама по себе и то же можно сказать о подходах к ее решению.

Два вида сборок — два вида развертывания

Среда CLR поддерживает два вида сборок: с *нестрогими именами* (weakly named assemblies) и со *строгими именами* (strongly named assemblies).

ВНИМАНИЕ

Вы никогда не встретите термин «сборка с нестрогим именем» в документации по .NET Framework. Почему? А потому, что я сам его придумал. В действительности в документации нет термина для обозначения сборки, у которой отсутствует строгое имя. Я решил обозначить такие сборки специальным термином, чтобы потом можно было недвусмысленно сказать, о каких сборках идет речь.

Сборки со строгими и нестрогими именами идентичны по структуре, то есть в них используется файловый формат PE (portable executable), заголовок PE32(+), CLR-заголовок, метаданные, таблицы манифеста, а также IL-код,

рассмотренный в главах 1 и 2. Оба типа сборок komponуются при помощи одних и тех же инструментов, например компилятора C# или AL.exe. В действительности сборки со строгими и нестрогими именами отличаются тем, что первые подписаны при помощи пары ключей, уникально идентифицирующей издателя сборки. Эта пара ключей позволяет уникально идентифицировать сборку, обеспечивать ее безопасность, управлять ее версиями, а также развертывать в любом месте пользовательского жесткого диска или даже в Интернете. Возможность уникально идентифицировать сборку позволяет CLR при попытке привязки приложения к сборке со строгим именем применить определенные политики, которые гарантируют безопасность. Эта глава посвящена разъяснению сущности сборок со строгим именем и политик, применяемых к ним со стороны CLR.

Развертывание сборки может быть закрытым или глобальным. Сборку первого типа развертывают в базовом каталоге приложения или в одном из его подкаталогов. Для сборки с нестрогим именем возможно лишь закрытое развертывание. О сборках с закрытым развертыванием речь шла в главе 2. Сборку с глобальным развертыванием устанавливают в каком-либо общеизвестном каталоге, который CLR проверяет при поиске сборок. Такие сборки можно развертывать как закрыто, так и глобально. В этой главе объяснено, как создают и развертывают сборки со строгим именем. Сведения о типах сборок и способах их развертывания представлены в табл. 3.1.

Таблица 3.1. Возможные способы развертывания сборок со строгими и нестрогими именами

Тип сборки	Закрытое развертывание	Глобальное развертывание
Сборка с нестрогим именем	Да	Нет
Сборка со строгим именем	Да	Да

ПРИМЕЧАНИЕ

Настоятельно рекомендую назначать сборкам строгие имена. Вполне вероятно, что будущие версии CLR потребуют, чтобы все сборки получали строгие имена, а сборки с нестрогими именами останутся «вне закона». Проблема с нестрогими сборками в том, что можно создать несколько разных сборок с одним нестрогим именем. В то же время присвоение строгого имени позволяет уникально идентифицировать ее. Если среда CLR сможет уникально идентифицировать сборку, она будет в состоянии применить больше политик, связанных с управлением версиями и обратной совместимостью. По большому счету, устранение возможности создавать сборки с нестрогими именами упрощает понимание политик управления версиями в CLR.

Назначение сборке строгого имени

Если планируется предоставить доступ к сборке нескольким приложениям, ее следует поместить в общеизвестный каталог, который среда CLR должна автоматически проверять, обнаружив ссылку на сборку. Однако с этим связана проблема — возможен вариант, когда две (или больше) компании сделают сборки с одинаковыми именами. Тогда, если обе эти сборки будут скопированы в один общеизвестный каталог, «победит» последняя из них, а работа приложений, использовавших первую, нарушится — ведь первая при копировании заменяется второй (это и является причиной «ада DLL» в современных системах Windows — все библиотеки DLL копируются в папку System32).

Очевидно, одного имени файла мало, чтобы различать две сборки. Среда CLR должна поддерживать некий механизм, позволяющий уникально идентифицировать сборку. Именно для этого и служат *строгие имена*. У сборки со строгим именем четыре атрибута, уникально ее идентифицирующих: имя файла (без расширения), номер версии, идентификатор регионального стандарта и открытый ключ. Поскольку открытые ключи представляют собой очень большие числа, вместо последнего атрибута используется небольшой хэш открытого ключа, который называют *маркером открытого ключа* (public key token). Следующие четыре строки, которые иногда называют отображаемым именем сборки (assembly display name), идентифицируют совершенно разные файлы сборки:

```
"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
"MyTypes, Version=1.0.8123.0, Culture="en-US", PublicKeyToken=b77a5c561934e089"
"MyTypes, Version=2.0.1234.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
```

Первая строка идентифицирует файл сборки **MyTypes.exe** или **MyTypes.dll** (на самом деле, по строке идентификации нельзя узнать расширение файла сборки). Компания-производитель назначила сборке номер версии 1.0.8123.0, в ней нет компонентов, зависящих от региональных стандартов, так как атрибут Culture определен как neutral. Но сделать сборку **MyTypes.dll** (или **MyTypes.exe**) с номером версии 1.0.8123.0 и нейтральными региональными стандартами может любая компания.

Должен быть способ отличить сборку, созданную этой компанией, от сборок других компаний, которым случайно были назначены те же атрибуты. В силу ряда причин компания Microsoft предпочла другим способам идентификации (при помощи GUID, URL и URN) стандартные криптографические технологии, основанные на паре из закрытого и открытого ключей. В частности, криптографические технологии позволяют проверять целостность данных сборки при установке ее на жесткий диск, а также ставят права доступа к сборке в зависимость от ее издателя. Все эти методики обсуждаются далее.

Итак, компания, желающая снабдить свои сборки уникальной меткой, должна получить пару ключей — открытый и закрытый, после чего открытый ключ можно будет связать со сборкой. У всех компаний будут разные пары ключей,

поэтому они смогут создавать сборки с одинаковыми именами, версиями и региональными стандартами, не опасаясь возникновения конфликтов.

ПРИМЕЧАНИЕ

Вспомогательный класс `System.Reflection.AssemblyName` позволяет легко генерировать имя для сборки, а также получать отдельные части имени сборки. Он поддерживает ряд открытых экземплярных свойств: `CultureInfo`, `FullName`, `KeyPair`, `Name` и `Version` — и предоставляет открытые экземплярные методы, такие как `GetPublicKey`, `GetPublicKey-Token`, `SetPublicKey` и `SetPublicKeyToken`.

В главе 2 я продемонстрировал механизм назначения имени файлу сборки и применение номера версии и идентификатора региональных стандартов. У сборки с нестрогим именем атрибуты номера версии и региональных стандартов могут быть включены в метаданные манифеста. Однако в этом случае CLR всегда игнорирует номер версии, а при поиске сопутствующих сборок использует лишь идентификатор региональных стандартов. Поскольку сборки с нестрогими именами всегда развертываются в закрытом режиме, для поиска файла сборки в базовом каталоге приложения или в одном из его подкаталогов, указанном атрибутом `privatePath` конфигурационного XML-файла, CLR просто берет имя сборки (добавляя к нему расширение DLL или EXE).

Кроме имени файла, у сборки со строгим именем есть номер версии и идентификатор региональных стандартов. Кроме того, она подписана при помощи закрытого ключа издателя.

Первый этап создания такой сборки — получение ключа при помощи утилиты **Strong Name (SN.exe)**, поставляемой в составе .NET Framework SDK и Microsoft Visual Studio. Эта утилита поддерживает множество функций, которыми пользуются, задавая в командной строке соответствующие параметры. Заметьте: все параметры командной строки **SN.exe** чувствительны к регистру. Чтобы сгенерировать пару ключей, выполните следующую команду:

```
SN -k MyCompany.snk
```

Эта команда заставит **SN.exe** создать файл **MyCompany.snk**, содержащий открытый и закрытый ключи в двоичном формате.

Числа, образующие открытый ключ, очень велики. При необходимости после создания этого файла можно использовать **SN.exe**, чтобы увидеть открытый ключ. Для этого нужно выполнить **SN.exe** дважды. Сначала — с параметром `-p`, чтобы создать файл, содержащий только открытый ключ (`MyCompany.PublicKey`):

```
SN -p MyCompany.keys MyCompany.PublicKey
```

А затем требуется выполнить **SN.exe** с параметром `-tp` и указать файл, содержащий открытый ключ:

```
SN -tp MyCompany.PublicKey
```


На своем компьютере я получил следующий результат:

```
Microsoft (R) .NET Framework Strong Name Utility Version 2.0.50727.42
Copyright (c) Microsoft Corporation. All rights reserved.
```

Public key is

```
002400000048000009400000006020000002400005253413100040000010001003f9d621b702111
850be453b92bd6a58c020eb7b804f75d67ab302047fc786ffa3797b669215afb4d814a6f294010
b233bac0b8c8098ba809855da256d964c0d07f16463d918d651a4846a62317328cac893626a550
69f21a125bc03193261176dd629eace6c90d36858de3fcb781bfc8b817936a567cad608ae672b6
1fb80eb0
```

Public key token is 3db32f38c8b42c9a

Вместе с тем невозможно заставить **SN.exe** аналогичным образом отобразить закрытый ключ.

Большой размер открытых ключей затрудняет работу с ними. Чтобы облегчить жизнь разработчику (и конечному пользователю), были созданы маркеры открытого ключа. Маркер открытого ключа — это 64-разрядный хэш открытого ключа. Если вызвать утилиту **SN.exe** с параметром `-tp`, то после значения ключа она выводит соответствующий маркер открытого ключа.

Теперь мы знаем, как создать криптографическую пару ключей, и получение сборки со строгим именем не должно вызывать затруднений. При компиляции сборки необходимо задать компилятору параметр `/keyfile:имя_файла`:

```
csc /keyfile:MyCompany.snk Program.cs
```

Обнаружив в исходном тексте этот параметр, компилятор открывает заданный файл (**MyCompany.snk**), подписывает сборку закрытым ключом и встраивает открытый ключ в манифест сборки. Заметьте: подписывается лишь файл сборки, содержащий манифест, другие файлы сборки нельзя подписать явно.

В Visual Studio новая пара ключей создается в окне свойств проекта. Для этого перейдите на вкладку **Signing**, установите флажок **Sign the assembly**, а затем в поле со списком **Choose a strong name key file** выберите вариант **<New...>**.

Слова «подписание файла» означают здесь следующее: при компоновке сборки со строгим именем в таблицу метаданных манифеста **FileDef** заносится список всех файлов, составляющих эту сборку. Каждый раз, когда к манифесту добавляется имя файла, рассчитывается хэш содержимого этого файла, и полученное значение сохраняется вместе с именем файла в таблице **FileDef**. Можно заменить алгоритм расчета хэша, используемый по умолчанию, вызвав **AL.exe** с параметром `/algid` или задав на уровне сборки следующий атрибут, определяемый пользователем, — **System.Reflection.AssemblyAlgorithmidAttribute**. По умолчанию для расчета хэша используется алгоритм **SHA-1**, возможностей которого должно хватать практически для любого приложения.

После компоновки PE-файла с манифестом рассчитывается хэш всего содержимого этого файла (за исключением подписи Authenticode Signature, строгого имени сборки и контрольной суммы заголовка PE), как показано на рис. 3.1. Для этой операции применяется алгоритм SHA-1, здесь его нельзя заменить никаким другим. Значение хэша подписывается закрытым ключом издателя, а полученная в результате цифровая подпись RSA заносится в зарезервированный раздел PE-файла (при расчете хэша PE-файла этот раздел исключается), и в CLR-заголовок PE-файла записывается адрес, по которому встроенная цифровая подпись находится в файле.

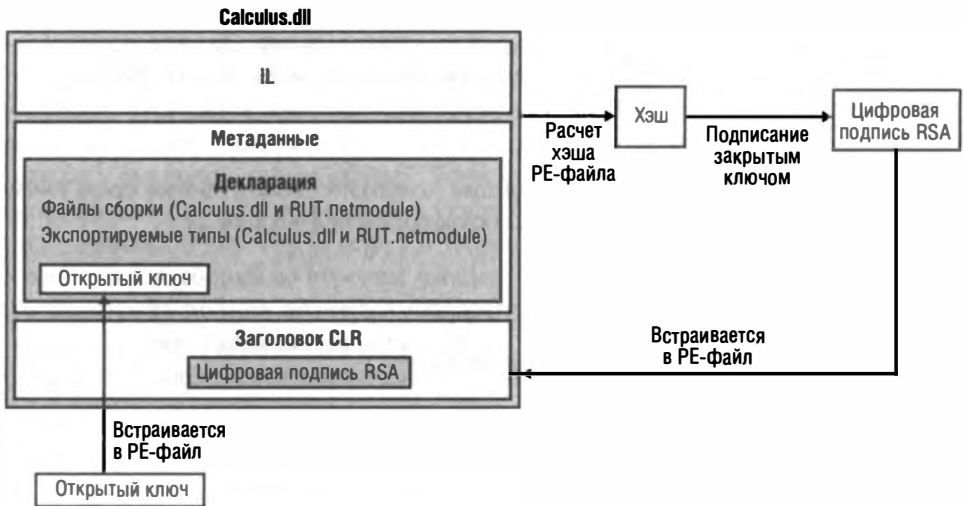


Рис. 3.1. Подписание сборки

В этот PE-файл также встраивается открытый ключ издателя (он записывается в таблицу AssemblyDef метаданных манифеста). Комбинация имени файла, версии сборки, региональных стандартов и значения открытого ключа составляет строгое имя сборки, которое гарантированно является уникальным. Ни одна компания ни при каких обстоятельствах не сможет создать две одинаковые сборки, скажем, с именем *Calculus*, с той же парой ключей.

Теперь сборка и все ее файлы готовы к упаковке и распространению.

Как отмечено в главе 2, при компиляции исходного текста компилятор обнаруживает все типы и члены, на которые ссылается исходный текст. Также компилятору необходимо указать все сборки, на которые ссылается данная сборка. В случае компилятора C# для этого применяется параметр */reference*. В задачу компилятора входит генерация таблицы метаданных AssemblyRef и размещение ее в результирующем управляемом модуле. Каждая запись таблицы метаданных AssemblyRef описывает файл сборки, на которую ссылается данная сборка, и состоит из имени файла сборки (без расширения), номера версии, регионального стандарта и значения открытого ключа.

ВНИМАНИЕ

Поскольку значение открытого ключа велико, в том случае, когда сборка ссылается на множество других сборок, значения открытых ключей могут занять значительную часть результирующего файла. Для экономии места в компании Microsoft рассчитывают хэш открытого ключа и берут последние 8 байт полученного хэша. В таблице AssemblyRef на самом деле хранятся именно такие усеченные значения открытого ключа — маркеры открытого ключа. В общем случае разработчики и конечные пользователи намного чаще встречаются с маркерами, чем с полными значениями ключа.

Вместе с тем нужно иметь в виду, что среда CLR никогда не использует маркеры открытого ключа в процессе принятия решений, касающихся безопасности или доверия, потому что одному маркеру может соответствовать несколько открытых ключей.

Далее приведены метаданные таблицы AssemblyRef (полученные средствами ILDasm.exe) для файла JeffTypes.dll, обсуждавшегося в главе 2:

AssemblyRef #1 (23000001)

```
-----
Token: 0x23000001
Public Key or Token: b7 7a 5c 56 19 34 e0 89
Name: mscorlib
Version: 4.0.0.0
Major Version: 0x00000004
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)
```

Из этих сведений видно, что файл JeffTypes.dll ссылается на тип, расположенный в сборке со следующими атрибутами:

"MSCorLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

К сожалению, в утилите ILDasm.exe используется термин Locale, хотя на самом деле там должно быть слово Culture.

Взглянув на содержимое таблицы метаданных AssemblyDef файла JeffTypes.dll, мы увидим следующее:

```
Assembly
-----
Token: 0x20000001
Name : JeffTypes
Public Key :
```

```
Hash Algorithm : 0x00008004
Version: 3.0.0.0
Major Version: 0x00000003
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
Flags : [none] (00000000)
```

Это эквивалентно следующей строке:

```
"JeffTypes, Version=3.0.0.0, Culture=neutral, PublicKeyToken=null"
```

Здесь открытый ключ не определен, поскольку сборка **JeffTypes.dll**, созданная в главе 2, не была подписана открытым ключом и, следовательно, является сборкой с нестрогим именем. Если бы я создал файл с ключами при помощи утилиты **SN.exe**, а затем скомпилировал сборку с параметром `/keyfile`, то получилась бы подписанная сборка. Если просмотреть метаданные полученной таким образом сборки при помощи утилиты **ILDasm.exe**, в соответствующей записи таблицы **AssemblyDef** обнаружится заполненное поле **Public Key**, говорящее о том, что это сборка со строгим именем. Кстати, запись таблицы **AssemblyDef** всегда хранит полное значение открытого ключа, а не его маркер. Полный открытый ключ гарантирует целостность файла. Позже я объясню принцип, лежащий в основе устойчивости к несанкционированной модификации сборок со строгими именами.

Глобальный кэш сборок

Теперь мы умеем создавать сборки со строгим именем — пора узнать, как развертывают такие сборки и как CLR использует метаданные для поиска и загрузки сборки.

Если сборка предназначена для совместного использования несколькими приложениями, ее нужно поместить в общеизвестный каталог, который среда CLR должна автоматически проверять, обнаружив ссылку на сборку. Место, где располагаются совместно используемые сборки, называют *глобальным кэшем сборок* (*global assembly cache, GAC*). Обычно это каталог `C:\Windows\Assembly` (предполагается, что система Windows установлена в каталог `C:\Windows`).

GAC обладает особой структурой и содержит множество вложенных каталогов, имена которых генерируются по определенному алгоритму. Ни в коем случае не следует копировать файлы сборок в GAC вручную — вместо этого надо использовать инструменты, созданные специально для этой цели. Эти инструменты «знают» внутреннюю структуру GAC и умеют генерировать надлежащие имена подкаталогов.

В период разработки и тестирования сборок со строгими именами для установки их в каталог GAC чаще всего применяют инструмент **GACUtil.exe**. Запущенный без параметров, он отобразит следующие сведения:

Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.20928.1
Copyright (c) Microsoft Corporation. All rights reserved.

Usage: Gacutil <command> [<options>]

Commands:

/i <assembly_path> [/r <...>] [/f]

Installs an assembly to the global assembly cache.

/il <assembly_path_list_file> [/r <...>] [/f]

Installs one or more assemblies to the global assembly cache.

/u <assembly_display_name> [/r <...>]

Uninstalls an assembly from the global assembly cache.

/ul <assembly_display_name_list_file> [/r <...>]

Uninstalls one or more assemblies from the global assembly cache.

/l [<assembly_name>]

List the global assembly cache filtered by <assembly_name>

/lr [<assembly_name>]

List the global assembly cache with all traced references.

/cdl

Deletes the contents of the download cache

/ldl

Lists the contents of the download cache

/?

Displays a detailed help screen

Options:

/r <reference_scheme> <reference_id> <description>

Specifies a traced reference to install (/i, /il) or uninstall (/u, /ul).

/f

Forces reinstall of an assembly.

/nologo

Suppresses display of the logo banner

/silent

Suppresses display of all output

Вызвав утилиту `GACUtil.exe` с параметром `/i`, можно установить сборку в каталог GAC, а если задать параметр `/u`, сборка будет удалена из GAC. Обратите внимание, что сборку с нестрогим именем нельзя поместить в GAC. Если передать `GACUtil.exe` файл сборки с нестрогим именем, утилита выдаст следующее сообщение об ошибке (ошибка добавления сборки в кэш: попытка установить сборку без строгого имени):

```
Failure adding assembly to the cache: Attempt to install an assembly without  
a strong name
```

ПРИМЕЧАНИЕ

По умолчанию манипуляции с каталогом GAC могут осуществлять лишь члены группы Windows Administrators (администраторы) или Power Users (опытные пользователи). `GACUtil.exe` не сможет установить или удалить сборку, если вызвавший утилиту пользователь не входит в эту группу.

Параметр `/i` утилиты `GACUtil.exe` очень удобен для разработчика во время тестирования. Однако при использовании `GACUtil.exe` для развертывания сборки в рабочей среде рекомендуется применять параметр `/r` в дополнение к `/i` — при установке и `/u` — при удалении сборки. Параметр `/r` обеспечивает интеграцию сборки с механизмом установки и удаления программ Windows. В сущности, утилита, вызванная с этим параметром, сообщает системе, для какого приложения требуется эта сборка, и связывает ее с ним.

ПРИМЕЧАНИЕ

Если сборка со строгим именем упакована в CAB-файл или сжата иным способом, то, прежде чем устанавливать файл сборки в каталог GAC при помощи утилиты `GACUtil.exe`, следует распаковать его во временный файл, который нужно удалить после установки сборки.

Утилита `GACUtil.exe` не входит в состав свободно распространяемого пакета .NET Framework, предназначенного для конечного пользователя. Если в приложении есть сборки, которые должны развертываться в каталоге GAC, используйте программу Windows Installer (MSI), так как это единственный инструмент, способный установить сборки в GAC и гарантированно присутствующий на машине конечного пользователя.

ВНИМАНИЕ

Глобальное развертывание сборки путем размещения ее в каталог GAC — это один из видов регистрации сборки в системе, хотя это никак не затрагивает реестр Windows. Установка сборок в GAC делает невозможным простые установку, копирование, восстановление, перенос и удаление приложения. По этой причине рекомендуется избегать глобального развертывания и использовать закрытое развертывание сборок всюду, где это только возможно.

Зачем «регистрировать» сборку в каталоге GAC? Представьте себе, что две компании сделали каждая свою сборку *Calculus*, состоящую из единственного файла: *Calculus.dll*. Очевидно, эти файлы нельзя записывать в один каталог, поскольку файл, копируемый последним, перезапишет первый и тем самым нарушит работу какого-нибудь приложения. Если для установки в GAC использовать специальный инструмент, он создаст в каталоге *C:\Windows\Assembly* отдельные папки для каждой из этих сборок и скопирует каждую сборку в свою папку.

Обычно пользователи не просматривают структуру каталогов GAC, поэтому для вас она не имеет реального значения. Довольно того, что структура каталогов GAC известна CLR и инструментам, работающим с GAC.

Компоновка сборки, ссылающейся на сборку со строгим именем

Какую бы сборку вы ни компоновали, в результате всегда получается сборка, ссылающаяся на другую сборку со строгим именем. Это утверждение верно хотя бы потому, что класс *System.Object* определен в *mscorlib.dll*, сборке со строгим именем. Однако велика вероятность того, что сборка также будет ссылаться на типы из других сборок со строгими именами, изданными Microsoft, сторонними разработчиками либо созданными в вашей организации. В главе 2 показано, как использовать компилятор *CSC.exe* с параметром */reference* для определения сборки, на которую должна ссылаться компонуемая сборка. Если вместе с именем файла задать полный путь к нему, *CSC.exe* загрузит указанный файл и использует его метаданные для компоновки сборки. Как отмечено в главе 2, если задано имя файла без указания пути, *CSC.exe* пытается найти нужную сборку в следующих каталогах (просматривая их в том порядке, в каком они здесь приводятся).

1. Рабочий каталог.
2. Каталог, где находится файл *CSC.exe*. Этот каталог также содержит DLL-библиотеки CLR.
3. Каталоги, заданные параметром командной строки */lib* при вызове *CSC.exe*.
4. Каталоги, указанные в переменной окружения *LIB*.

Таким образом, чтобы скомпоновать сборку, ссылающуюся на файл *System.Drawing.dll* разработки Microsoft, при вызове *CSC.exe* можно задать параметр */reference:System.Drawing.dll*. Компилятор проверит перечисленные каталоги и обнаружит файл *System.Drawing.dll* в одном каталоге с файлом среды CLR, которую сам использует для создания сборки. Однако несмотря на то, что при

компиляции сборка находится в этом каталоге, во время выполнения эта сборка загружается из другого каталога.

Во время установки .NET Framework все файлы сборок, созданных Microsoft, устанавливаются в двух экземплярах. Один набор файлов заносится в один каталог с CLR, а другой — в каталог GAC. Файлы в каталоге CLR облегчают компоновку пользовательских сборок, а их копии в GAC предназначены для загрузки во время выполнения.

Утилита CSC.exe не ищет нужные для компоновки сборки в GAC, чтобы вам не пришлось задавать громоздкие пути к файлам сборки. CSC.exe также позволяет задавать сборки при помощи не менее длинной, но чуть более изящной строки вида:

```
System.Drawing, Version=4.0.0.0, Culture=neutral, PublicKeyToken=
b03f5f7f11d50a3a
```

Оба способа столь неуклюжи, что было решено предпочесть им установку на пользовательский жесткий диск двух копий файлов сборок.

ПРИМЕЧАНИЕ

При компоновке сборки иногда требуется сослаться на другую сборку, существующую в двух версиях — x86 и x64. К счастью, подкаталоги каталога GAC могут хранить версии x86 и x64 одной сборки. Но, поскольку у этих сборок одно имя, их нельзя разместить в одном каталоге с CLR. Но это и неважно. При установке .NET Framework версии x86, x64 или IA64 сборок устанавливаются в каталоге CLR. При компоновке сборки можно ссылаться на любую версию ранее установленных файлов, так как все версии содержат одинаковые метаданные и различаются только кодом. Во время выполнения нужная версия сборки будет загружена из подкаталога GAC_32 или GACJ54. Далее в этой главе показано, как во время выполнения CLR определяет, откуда загружать сборку.

Устойчивость сборок со строгими именами к несанкционированной модификации

Подписание файла закрытым ключом гарантирует, что именно держатель соответствующего открытого ключа является производителем сборки. При установке сборки в GAC система рассчитывает хэш содержимого файла с манифестом и сравнивает полученное значение с цифровой подписью RSA, встроенной в PE-файл (после извлечения подписи с помощью открытого ключа). Идентичность значений означает, что содержимое файла не было модифицировано, а также то, что открытый ключ подписи соответствует закрытому

ключу издателя. Кроме того, система рассчитывает хэш содержимого других файлов сборки и сравнивает полученные значения с таковыми из таблицы манифеста FileDef. Если хоть одно из значений не совпадает, значит, хотя бы один из файлов сборки был модифицирован и установка сборки в каталог GAC закончится неудачей.

ВНИМАНИЕ

Этот механизм гарантирует лишь неприкосновенность содержимого файла; подлинность издателя он гарантирует, только если вы совершенно уверены, что обладаете открытым ключом, созданным издателем, и закрытый ключ издателя не был скомпрометирован. Если издатель желает связать со сборкой свои идентификационные данные, он должен дополнительно воспользоваться технологией Microsoft Authenticode.

Когда приложению требуется привязка к сборке, на которую оно ссылается, CLR использует для поиска этой сборки в GAC ее свойства (имя, версию, региональные стандарты и открытый ключ). Если нужная сборка обнаруживается, возвращается путь к каталогу, в котором она находится, и загружается файл с ее манифестом. Такой механизм поиска сборок гарантирует вызывающей стороне, что во время выполнения будет загружена сборка издателя, создавшего ту сборку, с которой компилировалась программа. Такая гарантия возможна благодаря соответствию маркера открытого ключа, хранящегося в таблице AssemblyRef ссылающейся сборки, открытому ключу из таблицы AssemblyDef сборки, на которую ссылаются. Если вызываемой сборки нет в GAC, CLR сначала ищет ее в базовом каталоге приложения, затем проверяет все закрытые пути, указанные в конфигурационном файле приложения, потом, если приложение установлено при помощи MSI, CLR просит MSI найти нужную сборку. Если ни в одном из этих вариантов сборка не находится, привязка заканчивается неудачей и генерируется исключение `System.IO.FileNotFoundException`.

ПРИМЕЧАНИЕ

Когда сборка со строгим именем загружается из каталога GAC, система гарантирует, что файлы, содержащие манифест, устойчивы к несанкционированной модификации. Эта проверка происходит только один раз на этапе установки. Для улучшения производительности среда CLR не проверяет, были ли файлы несанкционированно модифицированы, и загружает их в домен приложений с полными правами. В то же время, когда сборка со строгим именем загружается не из каталога GAC, среда CLR проверяет файл манифеста сборки дабы удостовериться в том, что он устойчив к несанкционированной модификации, занимая дополнительное время для проверки каждый раз во время загрузки этого файла.

При загрузке сборки со строгим именем не из GAC, а из другого каталога (каталога приложения или каталога, заданного значением элемента `codeBase` в конфигурационном файле) CLR проверяет ее хэш. Иначе говоря, в данном

случае расчет хэша для файла выполняется при каждом запуске приложения. Хотя при этом несколько снижается быстродействие, без таких мер нельзя гарантировать, что содержимое сборки не подверглось несанкционированной модификации. Обнаружив во время выполнения несоответствие значений хэша, CLR генерирует исключение `System.IO.FileLoadException`.

Отложенное подписание

Ранее в этой главе обсуждался способ получения криптографической пары ключей при помощи утилиты `SN.exe`. Эта утилита генерирует ключи, вызывая функции предоставленного Microsoft криптографического API-интерфейса под названием `CryptoAPI`. Полученные в результате ключи могут сохраняться в файлах на любых запоминающих устройствах. Например, в крупных организациях (вроде Microsoft) генерируемые закрытые ключи хранятся на аппаратных устройствах в сейфах, и лишь несколько человек из штата компании имеют доступ к закрытым ключам. Эти меры предосторожности предотвращают компрометацию закрытого ключа и обеспечивают его целостность. Ну, а открытый ключ, естественно, общедоступен и распространяется свободно.

Подготовившись к компоновке сборки со строгим именем, надо подписать ее закрытым ключом. Однако при разработке и тестировании сборки очень неудобно то и дело доставать закрытый ключ, который хранится «за семью печатями», поэтому .NET Framework поддерживает *отложенное* (delayed signing), или *частичное* (partial signing), *подписание*. Отложенное подписание позволяет компоновать сборку с открытым ключом компании, не требуя закрытого ключа. Открытый ключ дает возможность встраивать в записи таблицы `AssemblyRef` сборки, ссылающиеся на вашу сборку, получать правильное значение открытого ключа, а также корректно размещать эти сборки во внутренней структуре GAC. Не подписывая файл закрытым ключом, вы полностью лишаетесь защиты от несанкционированной модификации, так как при этом не рассчитывается хэш сборки и цифровая подпись не включается в файл. Однако на данном этапе это не проблема, поскольку подписание сборки откладывается лишь на время ее разработки, а готовая к упаковке и развертыванию сборка подписывается закрытым ключом.

Обычно открытый ключ компании получают в виде файла и передают его утилитами, компоноющим сборку. (Как уже отмечалось в этой главе, чтобы извлечь открытый ключ из файла, содержащего пару ключей, можно вызвать утилиту `SN.exe` с параметром `-p`.) Следует также указать компоноющей программе сборку, подписание которой будет отложено, то есть ту, что будет скомпонована без закрытого ключа. В компиляторе C# для этого служит параметр `/delaysign`. В Visual Studio в окне свойств проекта нужно перейти на вкладку **Signing** и установить флажок **Delay sign only**. При использовании утилиты `AL.exe` необходимо задать параметр `/delay[sign]`.

Обнаружив, что подписание сборки откладывается, компилятор или утилита `AL.exe` генерирует в таблице метаданных сборки `AssemblyDef` запись с открытым ключом сборки. Как обычно, наличие открытого ключа позволяет разместить эту сборку в GAC, а также создавать другие сборки, ссылающиеся на нее, при этом у них в записях таблицы метаданных `AssemblyRef` будет верное значение открытого ключа. При компоновке сборки в результирующем PE-файле остается место для цифровой подписи RSA. (Компонующая утилита определяет размер необходимого свободного места, исходя из размера открытого ключа.) Кстати, и на этот раз хэш файла не рассчитывается.

На этом этапе результирующая сборка не имеет действительной цифровой подписи. Попытка установки такой сборки в GAC окончится неудачей, так как хэш содержимого файла не был рассчитан, что создает видимость повреждения файла. Для того чтобы установить такую сборку в GAC, нужно запретить системе проверку целостности файлов сборки, вызвав утилиту `SN.exe` с параметром командной строки `-Vr`. Вызов `SN.exe` с таким параметром также вынуждает CLR пропустить проверку значения хэша для всех файлов сборки при ее загрузке во время выполнения. С точки зрения внутренних механизмов системы, параметр `-Vr` утилиты `SN.exe` обеспечивает размещение идентификационной информации сборки в разделе реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\StrongName\Verification`.

ВНИМАНИЕ

При использовании любой утилиты, имеющей доступ к реестру, необходимо убедиться в том, что для 64-разрядной платформы используется соответствующая 64-разрядная утилита. По умолчанию утилита под 32-разрядную платформу x86 установлена в каталоге `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\NETFX 4.0 Tools`, а утилита под 64-разрядную платформу x64 — в каталоге `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin\NETFX 4.0 Tools\x64`.

Окончательно протестированную сборку надо официально подписать, чтобы сделать возможными ее упаковку и развертывание. Чтобы подписать сборку, снова вызовите утилиту `SN.exe`, но на этот раз с параметром `-R`, указав имя файла, содержащего настоящий закрытый ключ. Параметр `-R` заставляет `SN.exe` рассчитать хэш содержимого файла, подписать его закрытым ключом и встроить цифровую подпись RSA в зарезервированное свободное место. После этого подписанная по всем правилам сборка готова к развертыванию. Можно также отменить проверку сборки, вызвав `SN.exe` с параметром `-Vu` или `-Vx`.

Полная последовательность действий по созданию сборки с отложенным подписанием выглядит следующим образом.

1. Во время разработки сборки следует получить файл, содержащий лишь открытый ключ компании, и добавить в строку компиляции сборки параметры `/keyfile` и `/delaysign`:

```
csc /keyfile:MyCompany.PublicKey /delaysign MyAssembly.cs
```

2. После компоновки сборки надо выполнить показанную далее команду, чтобы получить возможность тестирования этой сборки, установки ее в каталог GAC и компоновки других сборок, ссылающихся на нее. Эту команду достаточно исполнить лишь раз, не нужно делать это при каждой компоновке сборки.

```
SN.exe -Vr MyAssembly.dll
```

3. Подготовившись к упаковке и развертыванию сборки, надо получить закрытый ключ компании и выполнить приведенную далее команду. При желании можно установить новую версию GAC, но не пытайтесь это сделать до выполнения шага 4.

```
SN.exe -R MyAssembly.dll MyCompany.PrivateKey
```

4. Для тестирования сборки в реальных условиях, чтобы снова включить проверку, выполните следующую команду:

```
SN -Vu MyAssembly.dll
```

В начале раздела говорилось о хранении ключей организации на аппаратных носителях, например на смарт-картах. Для того чтобы обеспечить безопасность ключей, необходимо следить, чтобы они никогда не записывались на диск в виде файлов. Криптографические провайдеры (Cryptographic Service Providers, CSP) операционной системы предоставляют *контейнеры*, позволяющие абстрагироваться от физического места хранения ключей. Например, Microsoft использует CSP-провайдер, который при обращении к контейнеру считывает закрытый ключ со смарт-карты.

Если пара ключей хранится в CSP-контейнере, необходимо использовать другие параметры при обращении к утилитах **CSC.exe**, **AL.exe** и **SN.exe**. При компиляции (**CSC.exe**) вместо `/keyfile` нужно задействовать параметр `/keycontainer`, при компоновке (**AL.exe**) — параметр `/keyname` вместо `/keyfile`, а при вызове **SN.exe** для добавления закрытого ключа к сборке, подписание которой было отложено, — параметр `-Re` вместо `-R`. **SN.exe** поддерживает дополнительные параметры для работы с CSP.

ВНИМАНИЕ

Откладывать подписание удобно, когда необходимо выполнить какие-либо действия над сборкой до ее развертывания. Например, может понадобиться применить к сборке защитные утилиты, модифицирующие до неузнаваемости код. После подписания сборки это сделать уже не удастся, так как хэш станет недействительным. Так что если после компоновки сборки нужно ее защитить от декомпиляции или выполнить над ней другие действия, надо применить методику отложенного подписания. В конце нужно запустить утилиту **SN.exe** с параметром `-R` или `-Re`, чтобы завершить подписание сборки и рассчитать все необходимые хэш-значения.

Закрытое развертывание сборок со строгими именами

Установка сборок в каталог GAC дает несколько преимуществ. GAC позволяет нескольким приложениям совместно использовать сборки, сокращая в целом количество обращений к физической памяти. Кроме того, при помощи GAC легче развертывать новую версию сборки и заставить все приложения использовать новую версию сборки посредством реализации политики издателя (см. далее). GAC также обеспечивает совместное управление несколькими версиями сборки. Однако GAC обычно находится под защитой механизмов безопасности, поэтому устанавливать сборки в GAC может только администратор. Кроме того, установка сборки в GAC делает невозможным развертывание сборки простым копированием.

Хотя сборки со строгими именами могут устанавливаться в GAC, это вовсе не обязательно. В действительности рекомендуется развертывать сборки в GAC, только если они предназначены для совместного использования несколькими приложениями. Если сборка не предназначена для этого, следует развертывать ее закрыто. Это позволяет сохранить возможность установки путем «просто-го» копирования и лучше изолирует приложение с его сборками. Кроме того, GAC не задуман как замена каталогу C:\Windows\System32 в качестве «общей помойки» для хранения общих файлов. Это позволяет избежать затирания одних сборок другими путем установки их в разные каталоги, но «отъедает» дополнительное место на диске.

ПРИМЕЧАНИЕ

На самом деле элемент `codeBase` конфигурационного файла задает URL-адрес, который может ссылаться на любой каталог пользовательского жесткого диска или на адрес в Web. В случае веб-адреса CLR автоматически загрузит указанный файл и сохранит его в кэше загрузки на пользовательском жестком диске (в подкаталоге C:\Documents and Settings\<UserName>\Local Settings\ApplicationData\Assembly, где <UserName> — имя учетной записи пользователя, вошедшего в систему). В дальнейшем при ссылке на эту сборку CLR сверит метку времени локального файла и файла по указанному URL-адресу. Если последний новее, CLR загрузит файл только раз (это сделано для повышения производительности). Пример конфигурационного файла с элементом `codeBase` будет продемонстрирован позже.

Помимо развертывания в GAC или закрытого развертывания, сборки со строгими именами можно развертывать в произвольном каталоге, известном лишь небольшой группе приложений. Допустим, вы создали три приложения, совместно использующие одну и ту же сборку со строгим именем. После установки можно создать по одному каталогу для каждого приложения и дополнительный каталог для совместно используемой сборки. При установке приложений в их каталоги также записывается конфигурационный XML-файл,

а в элемент `codeBase` для совместно используемой сборки заносится путь к ней. В результате при выполнении CLR будет знать, что совместно используемую сборку надо искать в каталоге, содержащем сборку со строгим именем. Обратите внимание, что эту методику применяют довольно редко и в силу ряда причин не рекомендуют. Дело в том, что в таком сценарии ни одно отдельно взятое приложение не в состоянии определить, когда именно нужно удалить файлы совместно используемой сборки.

Как исполняющая среда разрешает ссылки на типы

В начале главы 2 вы видели следующий исходный текст:

```
public sealed class Program {
    public static void Main() {
        System.Console.WriteLine("Hi");
    }
}
```

В результате компиляции и компоновки этого кода получалась сборка, например `Program.exe`. При запуске приложения происходит загрузка и инициализация CLR. Затем CLR сканирует CLR-заголовок сборки в поисках атрибута `MethodDefToken`, идентифицирующего метод `Main`, представляющий точку входа в приложение. CLR находит в таблице метаданных `MethodDef` смещение, по которому в файле находится IL-код этого метода, и компилирует его в машинный код процессора при помощи JIT-компилятора. Этот процесс включает в себя проверку безопасности типов в компилируемом коде, после чего начинается исполнение полученного машинного кода. Далее показан IL-код метода `Main`. Чтобы получить его, я запустил `ILDasm.exe`, выбрал в меню **View** команду **Show Bytes** и дважды щелкнул на методе `Main` в дереве просмотра.

```
.method public hidebysig static voidMain() cil managed
// SIG: 00 00 01
{
    .entrypoint
    // Method begins at RVA 0x2050
    // Code size      11 (0xb)
    .maxstack 8
    IL_0000: /* 72 | (70)000001 */
               ldstr      "Hi"
    IL_0005: /* 28 | (0A)000003 */
               call       void [mscorlib]System.Console::WriteLine(string)
    IL_000a: /* 2A |
               ret
} // end of method Program::Main
```

Во время JIT-компиляции этого кода CLR обнаруживает все ссылки на типы и члены и загружает сборки, в которых они определены (если они еще не загружены). Как видите, показанный код содержит ссылку на метод `System.Console.WriteLine`: команда `Call` ссылается на маркер метаданных 0A000003. Этот маркер идентифицирует запись 3 таблицы метаданных `MemberRef` (таблица 0A). Просматривая эту запись, CLR видит, что одно из ее полей ссылается на элемент таблицы `TypeRef` (описывающий тип `System.Console`). Запись таблицы `TypeRef` направляет CLR к следующей записи в таблице `AssemblyRef`:

`mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089`

На этом этапе CLR уже знает, какая сборка нужна, и ей остается лишь найти и загрузить эту сборку.

При разрешении ссылки на тип CLR может найти нужный тип в одном из следующих мест.

- ❑ **В том же файле.** Обращение к типу, расположенному в том же файле, определяется при компиляции (данный процесс иногда называют *ранним связыванием*). Этот тип загружается прямо из этого файла, и исполнение продолжается.
- ❑ **В другом файле той же сборки.** Исполняющая среда проверяет, что файл, на который ссылаются, описан в таблице `FileRef` в манифесте текущей сборки. При этом исполняющая среда ищет его в каталоге, откуда был загружен файл, содержащий манифест сборки. Файл загружается, проверяется его хэш, чтобы гарантировать его целостность, затем CLR находит в нем нужный член типа, и исполнение продолжается.
- ❑ **В файле другой сборки.** Когда тип, на который ссылаются, находится в файле другой сборки, исполняющая среда загружает файл с манифестом этой сборки. Если в файле с манифестом необходимого типа нет, загружается соответствующий файл, CLR находит в нем нужный член типа, и исполнение продолжается.

ПРИМЕЧАНИЕ

Таблицы метаданных `ModuleDef`, `ModuleRef` и `FileDef` ссылаются на файлы по имени и расширению. Однако таблица метаданных `AssemblyRef` ссылается на сборки только по имени, без расширения. Во время привязки к сборке система автоматически добавляет к имени файла расширение DLL или EXE, пытаясь найти файл путем проверки каталогов по алгоритму, описанному в главе 2.

Если во время разрешения ссылки на тип возникают ошибки (не удастся найти или загрузить файл, не совпадает значение хэша и т. п.), генерируется соответствующее исключение.

ПРИМЕЧАНИЕ

При желании можно зарегистрировать в вашем коде методы обратного вызова с событиями из `System.AppDomain.AssemblyResolve`, `System.AppDomain.ReflectionOnlyAssemblyResolve` и `System.AppDomain.TypeResolve`. В методах обратного вызова вы можете выполнить программный код, который решает эту проблему и позволяет приложению выполняться без выбрасывания исключения.

В предыдущем примере среда CLR обнаруживала, что тип `System.Console` реализован в файле другой сборки. CLR должна найти эту сборку и загрузить PE-файл, содержащий ее манифест. После этого манифест сканируется в поисках сведений о PE-файле, в котором реализован искомый тип. Если необходимый тип содержится в том же файле, что и манифест, все замечательно, а если в другом файле, то CLR загружает этот файл и рассматривает его метаданные в поисках нужного типа. После этого CLR создает свою внутреннюю структуру данных для представления типа и JIT-компилятор завершает компиляцию метода `Main`. В завершение процесса начинается исполнение метода `Main`.

Рисунок 3.2 иллюстрирует процесс привязки к типам.

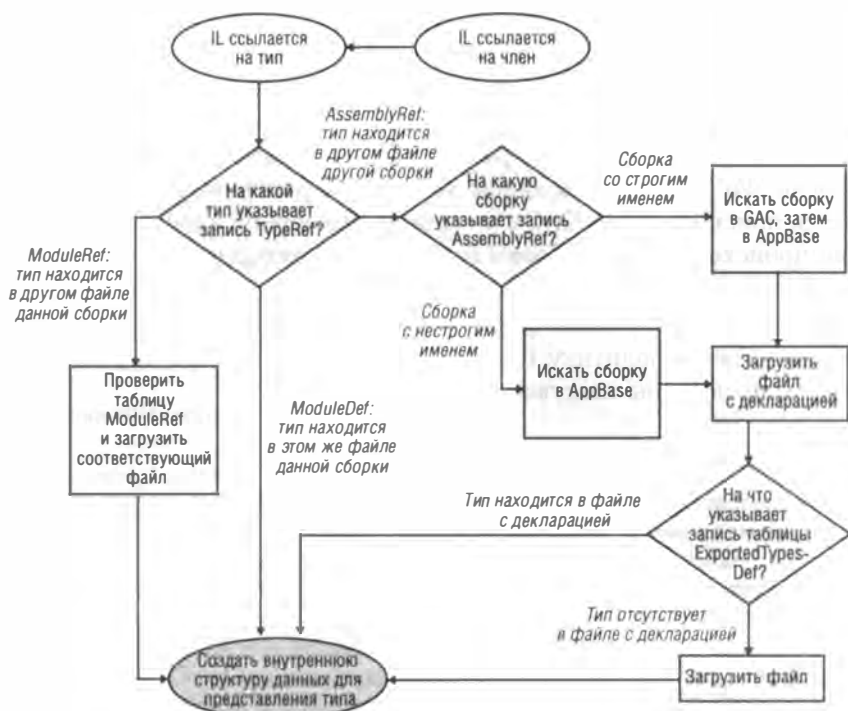


Рис. 3.2. Блок-схема алгоритма поиска метаданных, используемых CLR, файла сборки, где определен тип или метод, на который ссылается IL-код

Обратите внимание на то, что если какая-либо операция заканчивается неудачей, то выбрасывается соответствующее исключение.

ВНИМАНИЕ

Строго говоря, приведенный пример не является стопроцентно верным. Для ссылок на методы и типы, определенные в сборке, поставляемой в составе .NET Framework, все сказанное верно. Однако сборки .NET Framework (в том числе MSCorLib.dll) тесно связаны с работающей версией CLR. Любая сборка, ссылающаяся на сборки .NET Framework, всегда привязывается к соответствующей версии CLR. Этот процесс называют унификацией (unification), и Microsoft его поддерживает, потому что в этой компании все сборки .NET Framework тестируются во вполне определенной версии CLR. Поэтому унификация стека кода гарантирует корректную работу приложений.

В предыдущем примере ссылка на метод WriteLine объекта System.Console привязывается к любой версии MSCorLib.dll, совпадающей с CLR, независимо от того, на какую версию MSCorLib.dll ссылается таблица AssemblyRef в метаданных сборки.

Есть еще один нюанс: CLR идентифицирует все сборки по имени, версии, региональному стандарту и открытому ключу. Однако GAC различает сборки по имени, версии, региональному стандарту, открытому ключу и процессорной архитектуре. При поиске сборки в GAC среда CLR выясняет, в каком процессе выполняется приложение — 32-разрядном x86 (возможно, с использованием технологии WoW64), 64-разрядном x64 или 64-разрядном IA64. Сначала выполняется поиск сборки в GAC с учетом процессорной архитектуры. В случае неудачи происходит поиск сборки без учета процессорной архитектуры.

Из этого раздела мы узнали, как CLR ищет сборки, когда действует политика, предлагаемая по умолчанию. Однако администратор или издатель сборки может заменить эту политику. Способу изменения политики привязки CLR по умолчанию посвящены следующие два раздела.

ПРИМЕЧАНИЕ

CLR поддерживает возможность перемещения типа (класса, структуры, перечисляемого типа, интерфейса или делегата) из одной сборки в другую. Например, в .NET 3.5 класс System.TimeZoneInfo определен в сборке System.Core.dll. Но в .NET 4.0 компания Microsoft переместила этот класс в сборку MsCorLib.dll. В стандартной ситуации перемещение типа из одной сборки в другую нарушает работу приложения. Однако CLR предлагает воспользоваться атрибутом System.Runtime.CompilerServices.TypeForwardedToAttribute, который применяется в оригинальной сборке (например, System.Core.dll). Параметр, который необходимо указать, — System.Type. Он показывает новый тип (он будет определен в MSCorLib.dll), который теперь должно использовать приложение. С того момента, как

конструктор `TypeForwardedToAttribute` принимает этот тип, содержащая его сборка будет зависеть от сборки, в которой он определен.

Если вы воспользуетесь этим преимуществом, нужно также применить атрибут `System.Runtime.CompilerServices.TypeForwardedToAttribute` в новой сборке и указать конструктору атрибута полное имя сборки, которая служит для определения типа. Этот атрибут обычно используется для инструментальных средств, утилит и сериализации. Как только конструктор `TypeForwardedToAttribute` получает строку с этим именем, сборка, содержащая этот атрибут, становится независимой от сборки, определяющей тип.

Дополнительные административные средства (конфигурационные файлы)

В разделе «Простое средство администрирования (конфигурационный файл)» главы 2 мы кратко познакомились со способами изменения администратором алгоритма поиска и привязки к сборкам, используемого CLR. В том же разделе я показал, как перемещать файлы сборки, на которую ссылаются, в подкаталог базового каталога приложения и как CLR использует конфигурационный XML-файл приложения для поиска перемещенных файлов.

Поскольку в главе 2 нам удалось обсудить лишь атрибут `privatePath` элемента `probing`, здесь мы рассмотрим остальные элементы конфигурационного XML-файла:

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="AuxFiles;bin\subdir" />

      <dependentAssembly>

        <assemblyIdentity name="JeffTypes"
          publicKeyToken="32ab4ba45e0a69a1" culture="neutral"/>
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0" />
        <codeBase version="2.0.0.0"
          href="http://www.Wintellect.com/JeffTypes.dll" />
      </dependentAssembly>
    </dependentAssembly>
    <assemblyIdentity name="TypeLib"
      publicKeyToken="1f2e74e897abbcfe" culture="neutral"/>
    <bindingRedirect oldVersion="3.0.0.0-3.5.0.0"
```

```

        newVersion="4.0.0.0" />
    <publisherPolicy apply="no" />
</dependentAssembly>
</assemblyBinding>
</runtime>
</configuration>

```

XML-файл предоставляет CLR обширную информацию.

- ❑ **Элемент probing.** Определяет поиск в подкаталогах AuxFiles и bin\subdir, расположенных в базовом каталоге приложения, при попытке найти сборку с нестрогим именем. Сборки со строгим именем CLR ищет в GAC или по URL-адресу, указанному элементом codeBase.
- ❑ **Первый набор элементов dependentAssembly, assemblyIdentity и binding-Redirect.** При попытке найти сборки FredTypes с номерами версий с 3.0.0.0 по 3.5.0.0 включительно и нейтральными региональными стандартами, изданные организацией, владеющей открытым ключом с маркером 1f2e74e897abbcfe, система вместо этого будет искать аналогичную сборку, но с номером версии 4.0.0.0.
- ❑ **Элемент codebase.** При попытке найти сборку JeffTypes с номером версии 2.0.0.0 и нейтральными региональными стандартами, изданную организацией, владеющей открытым ключом с маркером 32ab4ba45e0a69a1, система будет пытаться выполнить привязку по адресу, заданному в URL: <http://www.Wintellect.com/JeffTypes.dll>. Хотя я и не говорил об этом в главе 2, элемент codeBase можно применять и к сборкам с нестрогими именами. При этом номер версии сборки игнорируется и его следует опустить при определении элемента codeBase. URL-адрес, заданный элементом codeBase, также должен ссылаться на подкаталог базового каталога приложения.
- ❑ **Второй набор элементов dependentAssembly, assemblyIdentity и binding-Redirect.** Подменяет искомую сборку: при попытке найти сборку JeffTypes с номером версии 1.0.0.0 и нейтральными региональными стандартами, изданную организацией, владеющей открытым ключом с маркером 32ab4ba45e0a69a1, система будет искать аналогичную сборку, но с номером версии 2.0.0.0.
- ❑ **Элемент publisherPolicy.** Если организация, производитель сборки TypeLib, развернула файл политики издателя (описание этого файла см. в следующем разделе), среда CLR должна игнорировать этот файл.

При компиляции метода CLR определяет типы и члены, на которые он ссылается. Используя эти данные, исполняющая среда определяет (путем просмотра таблицы AssemblyRef вызывающей сборки), на какую сборку исходно ссылалась вызывающая сборка во время компоновки. Затем CLR ищет сведения о сборке в конфигурационном файле приложения и следует любым изменениям номера версии, заданным в этом файле.

Если значение атрибута `apply` элемента `publisherPolicy` равно `yes` или отсутствует, CLR проверяет наличие в GAC новой сборки/версии и применяет все перенаправления, которые счел необходимым указать издатель сборки (о политике издателя рассказывается в следующем разделе); далее CLR ищет именно эту сборку/версию. Наконец просматривает сборку/версию в файле `Machine.config` и применяет все указанные в нем перенаправления к другим версиям.

На этом этапе CLR определяет номер версии сборки, которую она должна загрузить, и пытается загрузить соответствующую сборку из GAC. Если сборки в GAC нет, а элемент `codeBase` не определен, CLR пытается найти сборку, как описано в главе 2. Если конфигурационный файл, задающий последнее изменение номера версии, содержит элемент `codeBase`, CLR пытается загрузить сборку с URL-адреса, заданного этим элементом.

Эти конфигурационные файлы обеспечивают администратору настоящий контроль над решением, принимаемым CLR относительно загрузки той или иной сборки. Если в приложении обнаруживается ошибка, администратор может связаться с издателем сборки, содержащей ошибку, после чего издатель пришлет новую сборку. По умолчанию среда CLR не может загрузить новую сборку, потому что уже существующая сборка не ссылается на новую версию. Однако администратор может заставить CLR загрузить новую сборку, модифицировав конфигурационный XML-файл приложения.

Если администратор хочет, чтобы все сборки, установленные на компьютере, использовали новую версию, то вместо конфигурационного файла приложения он может модифицировать файл `Machine.config` для данного компьютера, и CLR будет загружать новую версию сборки при каждой ссылке из приложений на старую версию.

Если в новой версии старая ошибка не исправлена, администратор может удалить из конфигурационного файла строки, определяющие использование этой сборки, и приложение станет работать, как раньше. Важно, что система позволяет использовать сборку, версия которой отличается от версии, описанной в метаданных. Такая дополнительная гибкость очень удобна.

Управление версиями при помощи политики издателя

В ситуации, описанной в предыдущем разделе, издатель сборки просто прислал новую версию сборки администратору, который устанавливал сборку и вручную вносил изменения в конфигурационные XML-файлы машины или приложения. Вообще, после того как издатель исправил ошибку в сборке, ему нужен простой способ упаковки и распространения новой сборки всем пользователям. Кроме того, нужно как-то заставить среду CLR каждого пользователя задействовать новую версию сборки вместо старой. Естественно, каждый пользователь может сам изменить конфигурационные XML-файлы на своих машинах, но это ужасно неудобно, да и чревато ошибками. Издателю нужен

подход, который позволил бы ему создать свою «политику» и установить ее на пользовательский компьютер с новой сборкой. В этом разделе показано, как издатель сборки может создать подобную политику.

Допустим, вы — издатель, только что создавший новую версию своей сборки, в которой исправлено несколько ошибок. Упаковывая новую сборку для рассылки пользователям, надо создать конфигурационный XML-файл. Он очень похож на те, что мы обсуждали раньше. Вот пример конфигурационного файла `JeffTypes.config` для сборки `JeffTypes.dll`:

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>

        <assemblyIdentity name="JeffTypes"
          publicKeyToken="32ab4ba45e0a69a1" culture=" neutral"/>
        <bindingRedirect oldVersion="1.0.0.0" newVersion=" 2.0.0.0" />
        <codeBase version="2.0.0.0"
          href="http://www.Wintellect.com/JeffTypes.dll"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Конечно, издатель может определять политику только для своих сборок. Кроме того, показанные здесь элементы — единственные, которые можно задать в конфигурационном файле политики издателя. Например, в конфигурационном файле политики нельзя задавать элементы `probing` и `publisherPolicy`.

Этот конфигурационный файл заставляет CLR при каждой ссылке на версию 1.0.0.0 сборки `JeffTypes` загружать вместо нее версию 2.0.0.0. Теперь вы, как издатель, можете создать сборку, содержащую конфигурационный файл политики издателя. Для создания сборки с политикой издателя вызывается утилита `AL.exe` со следующими параметрами:

```
AL.exe /out:policy.1.0.JeffTypes.dll
      /version:1.0.0.0
      /keyfile:MyCompany.keys
      /linkresource:JeffTypes.config
```

Смысл параметров командной строки для `AL.exe` следующий.

- ❑ Параметр `/out` приказывает `AL.exe` создать новый PE-файл с именем `Policy.1.0.JeffTypes.dll`, в котором нет ничего, кроме манифеста. Имя этой сборки имеет очень большое значение. Первая часть имени, `Policy`, сообщает CLR, что сборка содержит информацию политики издателя. Вторая и третья части имени, `1.0`, сообщают CLR, что эта политика издателя предназначена для любой версии сборки `JeffTypes`, у которой старший и младший номера

версии равны 1.0. Политики издателя применяются только к старшему и младшему номерам версии сборки; нельзя создать политику издателя для отдельных компоновок или редакций сборки. Четвертая часть имени, `JeffTypes`, указывает имя сборки, которой соответствует политика издателя. Пятая и последняя часть имени, `dll`, — это просто расширение, данное результирующему файлу сборки.

- ❑ Параметр `/version` идентифицирует версию сборки с политикой издателя, которая не имеет ничего общего с версией самой сборки. Как видите, версиями сборок, содержащих политику издателя, тоже можно управлять. Сейчас издателю нужно создать политику, перенаправляющую CLR от версии 1.0.0.0 сборки `JeffTypes` к версии 2.0.0.0, а в будущем может потребоваться политика, перенаправляющая от версии 1.0.0.0 сборки `JeffTypes` к версии 2.5.0.0. CLR использует номер версии, заданный этим параметром, чтобы выбрать самую последнюю версию сборки с политикой издателя.
- ❑ Параметр `/keyfile` заставляет `AL.exe` подписать сборку с политикой издателя при помощи пары ключей, принадлежащей издателю. Эта пара ключей также должна совпадать с парой, использованной для подписания всех версий сборки `JeffTypes`. В конце концов, именно это совпадение позволяет CLR установить, что сборка `JeffTypes` и файл с политикой издателя для этой сборки созданы одним издателем.
- ❑ Параметр `/linkresource` заставляет `AL.exe` считать конфигурационный XML-файл отдельным файлом сборки. При этом в результате компоновки получается сборка из двух файлов. Оба следует упаковывать и развертывать на пользовательских компьютерах с новой версией сборки `JeffTypes`. Между прочим, конфигурационный XML-файл нельзя встраивать в сборку, вызвав `AL.exe` с параметром `/embedresource`, и создавать таким образом сборку, состоящую из одного файла, так как CLR требует, чтобы сведения о конфигурации в формате XML размещались в отдельном файле.

Сборку, скомпонованную с политикой издателя, можно упаковать с файлом новой версии сборки `JeffTypes.dll` и передать пользователям. Сборка с политикой издателя должна устанавливаться в GAC. Саму сборку `JeffTypes` можно установить в GAC, но это не обязательно. Ее можно развернуть в базовом каталоге приложения или в другом каталоге, заданном в URL-адресе из элемента `codeBase`.

ВНИМАНИЕ

Издатель должен создавать сборку со своей политикой лишь для развертывания исправленной версии сборки или пакетов исправлений для нее. Установка нового приложения не должна требовать сборки с политикой издателя.

И последнее о политике издателя. Допустим, издатель распространил сборку с политикой издателя, но в новой сборке почему-то оказалось больше новых

ошибок, чем исправлено старых. В этом случае администратору необходимо, чтобы CLR игнорировала сборку с политикой издателя. Для этого он может отредактировать конфигурационный файл приложения, добавив в него элемент `publisherPolicy`:

```
<publisherPolicy apply="no"/>
```

Этот элемент можно разместить в конфигурационном файле приложения как потомок элемента `<assemblyBinding>` — в этом случае он применяется ко всем его сборкам, а в качестве потомка элемента `<dependantAssembly>` — к отдельной сборке. Обработывая конфигурационный файл приложения, CLR «видит», что не следует искать в GAC сборку с политикой издателя, и продолжает работать с более старой версией сборки. Однако замечу, что CLR все равно проверяет и применяет любую политику, заданную в файле `Machine.config`.

ВНИМАНИЕ

Использование сборки с политикой издателя позволяет издателю заявить о совместимости разных версий сборки. Если новая версия несовместима с более ранней версией, издатель не должен создавать сборку с политикой издателя. Вообще, следует использовать сборки с политикой издателя, если компонуется новая версия с исправлениями ошибок. Новую версию сборки нужно протестировать на обратную совместимость. В то же время, если к сборке добавляются новые функции, следует подумать о том, чтобы отказаться от связи с прежними сборками и от применения сборки с политикой издателя. Кроме того, в этом случае отпадет необходимость тестирования на обратную совместимость.

ЧАСТЬ II

Проектирование типов

Глава 4.	Основы типов	100
Глава 5.	Примитивные, ссылочные и значимые типы	121
Глава 6.	Основные сведения о членах и типах	165
Глава 7.	Константы и поля	191
Глава 8.	Методы	196
Глава 9.	Параметры	230
Глава 10.	Свойства	248
Глава 11.	События	272
Глава 12.	Обобщения	289
Глава 13.	Интерфейсы	322

Глава 4. Основы типов

В этой главе мы знакомимся с основами типов и общезыковой исполняющей среды (Common Language Runtime, CLR). В частности, я представляю минимальную функциональность, присущую всем типам, рассказываю о контроле типов, пространствах имен, сборках и различных способах приведения типов объектов. В конце главы я объясняю, как во время выполнения взаимодействуют друг с другом типы, объекты, стеки потоков и управляемая куча.

Все типы — производные от System.Object

В CLR каждый объект прямо или косвенно является производным от System.Object. Это значит, что следующие определения типов идентичны:

```
// Тип, неявно производный от Object
class Employee {
    ...
}
// Тип, явно производный от Object
class Employee : System.Object {
    ...
}
```

Благодаря тому, что все типы, в конечном счете, являются производными от System.Object, любой объект любого типа гарантированно имеет минимальный набор методов. Открытые экземплярные методы класса System.Object перечислены в табл. 4.1.

Таблица 4.1. Открытые методы System.Object

Открытый метод	Описание
Equals	Возвращает true, если два объекта имеют одинаковые значения. Подробнее об этом методе рассказывается в разделе «Равенство и тождество объектов» главы 5
GetHashCode	Возвращает хэш-код для значения данного объекта. Этот метод следует переопределить, если объекты типа используются в качестве ключа в коллекции хэш-таблиц. Очень плохо, что этот метод определен в классе Object, потому что большинство типов не служат ключами в хэш-таблице, этот метод уместнее было бы определить в интерфейсе. Подробнее об этом методе рассказывается в разделе «Хэш-коды объектов» главы 5

Открытый метод	Описание
ToString	По умолчанию возвращает полное имя типа (this.GetType().FullName). На практике этот метод переопределяют, чтобы он возвращал объект String, содержащий состояние объекта в виде строки. Например, переопределенные методы для таких фундаментальных типов, как Boolean и Int32, возвращают значения объектов в строковом виде. Кроме того, к переопределению метода часто прибегают при отладке: вызов такого метода позволяет получить строку, содержащую значения полей объекта. Считается, что ToString «знает» о методе CultureInfo, связанном с вызывающим потоком. Подробнее о методе ToString рассказывается в главе 14
GetType	Возвращает экземпляр объекта, производного от Type, который идентифицирует тип объекта, вызвавшего GetType. Возвращаемый объект Type может использоваться с классами, реализующими отражение для получения информации о типе в виде метаданных. Об отражении см. главу 23. Метод GetType неvirtуальный, его нельзя переопределить, поэтому классу не удастся исказить сведения о своем типе. Таков механизм обеспечения безопасности типов

Кроме того, типы, производные от System.Object, имеют доступ к защищенным методам (табл. 4.2).

Таблица 4.2. Защищенные методы System.Object

Защищенный метод	Описание
MemberwiseClone	Этот неvirtуальный метод создает новый экземпляр типа и присваивает полям нового объекта соответствующие значения объекта this. Возвращается ссылка на созданный экземпляр
Finalize	Этот virtуальный метод вызывается, когда сборщик мусора определяет, что объект является мусором, но до возвращения занятой объектом памяти в кучу. В типах, требующих очистки при сборке мусора, следует переопределить этот метод. Подробнее о нем см. главу 21

CLR требует, чтобы все объекты создавались с помощью оператора new. Объект Employee создается следующим образом:

```
Employee e = new Employee("ConstructorParam1");
```

Оператор new действует так:

1. Вычисляет число байтов, необходимых всем экземплярным полям типа и всем его базовым типам, включая System.Object (в котором отсутствуют собственные экземплярные поля). Каждый объект кучи требует дополнительных членов, они называются *указателем на объект-тип* (type object pointer) и *индексом блока синхронизации* (sync block index) и служат CLR для управления объектом. Байты этих дополнительных членов добавляются к байтам, необходимым для размещения самого объекта.

2. Выделяет память для объекта, резервируя необходимое для данного типа число байтов в управляемой куче байтов, затем все эти байты устанавливаются в ноль (0).
3. Инициализирует указатель на объект-тип и индекс блока синхронизации.
4. Вызывает конструктор экземпляра типа с параметрами, указанными при вызове `new` (в предыдущем примере это строка `ConstructorParam1`). Большинство компиляторов автоматически создает в конструкторе код вызова конструктора базового класса. Каждый конструктор выполняет инициализацию определенных в соответствующем типе полей. В частности, вызывается конструктор `System.Object`, но он ничего не делает и просто возвращает управление. Это легко проверить, загрузив в утилиту `ILDasm.exe` библиотеку `mscorlib.dll` и изучив метод-конструктор типа `System.Object`.

Выполнив все эти операции, `new` возвращает ссылку (или указатель) на вновь созданный объект. В предыдущем примере кода эта ссылка сохраняется в переменной `e` типа `Employee`.

Кстати, у оператора `new` нет пары — оператора `delete`, то есть нет явного способа освобождения памяти, занятой объектом. Сборкой мусора занимается среда CLR (см. главу 21), которая автоматически находит объекты, ставшие ненужными или недоступными, и освобождает занимаемую ими память.

Приведение типов

Одна из важнейших особенностей CLR — *безопасность типов* (type safety). В время выполнения программы среде CLR всегда известен тип объекта. Точно определить тип объекта позволяет метод `GetType`. Поскольку это не виртуальный метод, никакой тип не сможет сообщить о себе ложные сведения. Например, тип `Employee` не может переопределить метод `GetType`, чтобы тот вернул тип `SuperHero`.

При разработке программ часто прибегают к приведению объекта к другим типам. CLR разрешает привести тип объекта к его собственному типу или любому из его базовых типов. В каждом языке программирования приведение типов реализовано по-своему. Например, в C# нет специального синтаксиса для приведения типа объекта к его базовому типу, поскольку такое приведение считается безопасным неявным преобразованием. Однако для приведения типа к производному от него типу разработчик на C# должен ввести операцию явного приведения типов — неявное преобразование приведет к ошибке. Следующий пример демонстрирует приведение к базовому и производному типам:

```
// Этот тип неявно наследует от типа System.Object
internal class Employee {
    ...
}
```

```
public sealed class Program {  
    public static void Main() {  
        // Приведение типа не требуется, т. к. new возвращает объект Employee.  
        // а Object – это базовый тип для Employee.  
        Object o = new Employee();  
        // Приведение типа обязательно, т. к. Employee – производный от Object  
        // В других языках (таких как Visual Basic) компилятор не потребует  
        // явного приведения  
        Employee e = (Employee) o;  
    }  
}
```

Этот пример демонстрирует то, что необходимо компилятору для компиляции кода. Посмотрим, что произойдет во время выполнения программы. CLR проверит операции приведения, чтобы приведение типов осуществлялось либо к фактическому типу объекта, либо к одному из его базовых типов. Например, следующий код успешно компилируется, но в период выполнения выбрасывает исключение `InvalidCastException`:

```
internal class Employee {  
    ...  
}  
internal class Manager : Employee {  
    ...  
}
```

```
public sealed class Program {  
    public static void Main() {  
        // Создаем объект Manager и передаем его в PromoteEmployee  
        // Manager ЯВЛЯЕТСЯ производным от Object,  
        // поэтому PromoteEmployee работает  
        Manager m = new Manager();  
        PromoteEmployee(m);  
  
        // Создаем объект DateTime и передаем его в PromoteEmployee  
        // DateTime НЕ ЯВЛЯЕТСЯ производным от Employee,  
        // поэтому PromoteEmployee вбрасывает  
        // исключение System.InvalidCastException  
        DateTime newYears = new DateTime(2010, 1, 1);  
        PromoteEmployee(newYears);  
    }  
}
```

```
public static void PromoteEmployee(Object o) {  
    // В этом месте компилятор не знает точно, на какой тип объекта  
    // ссылается o, поэтому скомпилирует этот код  
    // Однако в период выполнения CLR знает, на какой тип  
    // ссылается объект o (приведение типа выполняется каждый раз),
```

продолжение ➤

```
// и проверяет, соответствует ли тип объекта типу Employee
// или другому типу, производному от Employee
Employee e = (Employee) o;
...
}
}
```

Метод Main создает объект Manager и передает его в PromoteEmployee. Этот код компилируется и выполняется, так как тип Manager является производным от Object, на который рассчитан PromoteEmployee. Внутри PromoteEmployee CLR проверяет, на что ссылается o — на объект Employee или на объект типа, производного от Employee. Поскольку Manager — производный от Employee тип, CLR выполняет преобразование, и PromoteEmployee продолжает работу.

После того как PromoteEmployee возвращает управление, Main создает объект DateTime, который передает в PromoteEmployee. Объект DateTime тоже является производным от Object, поэтому код, вызывающий PromoteEmployee, компилируется без проблем. Но при выполнении PromoteEmployee CLR выясняет, что o ссылается на объект DateTime, не являющийся ни типом Employee, ни другим типом, производным от Employee. В этот момент CLR не в состоянии выполнить приведение типов и генерирует исключение System.InvalidCastException.

Если разрешить подобное преобразование, работа с типами станет небезопасной. При этом последствия могут быть непредсказуемы: повысится вероятность краха приложения или возникнет брешь в защите, обусловленная возможностью типов выдавать себя за другие типы. Последнее обстоятельство подвергает серьезному риску устойчивость работы приложений. Поэтому столь пристальное внимание в CLR уделяется безопасности типов.

Кстати, в данном примере было бы правильнее выбрать для метода PromoteEmployee в качестве типа параметра не Object, а Employee. А объект Object я использовал только для того, чтобы показать, как обрабатывают операции приведения типов компилятор C# и среда CLR.

Приведение типов в C# с помощью операторов is и as

В C# есть другие механизмы приведения типов. Например, оператор is проверяет совместимость объекта с данным типом, а в качестве результата выдает значение типа Boolean (true или false). Оператор is никогда не генерирует исключение. Взгляните на следующий код:

```
Object o = new Object();
Boolean b1 = (o is Object); // b1 равно true
Boolean b2 = (o is Employee); // b2 равно false
```

Если ссылка на объект равна null, оператор is всегда возвращает false, так как нет объекта, для которого нужно определить тип.

Обычно оператор `is` используется следующим образом:

```
if (o is Employee) {  
    Employee e = (Employee) o;  
    // Используем e внутри инструкции if  
}
```

В этом коде CLR по сути проверяет тип объекта дважды: сначала в операторе `is` определяется совместимость `o` с типом `Employee`, а затем в теле оператора `if` анализируется, является ли `o` ссылкой на `Employee`. Контроль типов в CLR укрепляет безопасность, но при этом приходится жертвовать производительностью, так как среда CLR должна выяснять фактический тип объекта, на который ссылается переменная (`o`), а затем проверять всю иерархию наследования на предмет наличия среди базовых типов заданного типа (`Employee`). Поскольку такая схема встречается в программировании часто, в C# предложен механизм, повышающий эффективность кода с помощью оператора `as`:

```
Employee e = o as Employee;  
if (e != null) {  
    // Используем e внутри инструкции if  
}
```

В этом коде CLR проверяет совместимость `o` с типом `Employee`, если это так, `as` возвращает ненулевой указатель на этот объект. Если `o` и `Employee` несовместимы, оператор `as` возвращает `null`. Заметьте: оператор `as` заставляет CLR верифицировать тип объекта только один раз, а `if` лишь сравнивает `e` с `null` — такая проверка намного эффективнее, чем определение типа объекта.

По сути, оператор `as` отличается от оператора приведения типа только тем, что никогда не генерирует исключение. Если приведение типа невозможно, результатом является `null`. Если не сравнить полученный оператором результат с `null` и попытаться работать с пустой ссылкой, возникнет исключение `NullReferenceException`. Например:

```
System.Object o = new Object(); // Создание объекта Object  
Employee e = o as Employee;     // Приведение o к типу Employee  
// Преобразование невыполнимо: исключение не возникло, но e равно null  
e.ToString(); // Обращение к e вызывает исключение NullReferenceException
```

Для того чтобы убедиться в том, что вы усвоили материал, выполним простой тест. Допустим, существуют описания следующих классов:

```
internal class B {    // Базовый класс  
}  
internal class D : B { // Производный класс  
}
```

Таблица 4.3 содержит в первом столбце код на C#. Давайте определим, каков будет результат обработки этих строк компилятором и CLR. Если код

компилируется и выполняется без ошибок, укажем это в графе ОК, если вызывает ошибку компиляции — в графе CTE (compile-time error), а если приводит к ошибке в период выполнения — в графе RTE (run-time error).

Таблица 4.3. Тест на знание контроля типов

Оператор	ОК	СТЕ	СТЕ
Object o1 = new Object();	Да		
Object o2 = new B();	Да		
Object o3 = new D();	Да		
Object o4 = o3;	Да		
B b1 = new B();	Да		
B b2 = new D();	Да		
D d1 = new D();	Да		
B b3 = new Object();		Да	
D d2 = new Object();		Да	
B b4 = d1;	Да		
D d3 = b2;		Да	
D d4 = (D) d1;	Да		
D d5 = (D) b2;	Да		
D d6 = (D) b1;			Да
B b5 = (B) o1;			Да
B b6 = (D) b2;	Да		

ПРИМЕЧАНИЕ

В С# разрешено определять методы операторов преобразования при помощи типов, об этом речь идет в разделе «Методы операторов преобразования» главы 8. Эти методы вызываются только в случаях, когда имеет место приведение типов, и никогда не вызываются при использовании операторов `is` и `as` в С#.

Пространства имен и сборки

Пространства имен позволяют объединять родственные типы в логические группы, в которых разработчику проще найти нужный тип. Например, в пространстве имен `System.Text` описаны типы для обработки строк, а в пространстве имен `System.IO` — типы для выполнения операций ввода-вывода.

В следующем коде создаются объекты `System.IO.FileStream` и `System.Text.StringBuilder`:

```
public sealed class Program {  
    public static void Main() {  
        System.IO.FileStream fs = new System.IO.FileStream(...);  
        System.Text.StringBuilder sb = new System.Text.StringBuilder();  
    }  
}
```

Этот код грешит многословием — он станет изящнее, если обращение к типам `FileStream` и `StringBuilder` будет компактнее. К счастью, многие компиляторы предоставляют программистам механизмы, позволяющие сократить объем набираемого текста. Например, в компиляторе C# предусмотрена директива `using`. Следующий код аналогичен предыдущему:

```
using System.IO; // Попробуем избавиться от приставок "System.IO"  
using System.Text; // Попробуем избавиться от приставок "System.Text"  
public sealed class Program {  
    public static void Main() {  
        FileStream fs = new FileStream(...);  
        StringBuilder sb = new StringBuilder();  
    }  
}
```

Для компилятора пространство имен — простое средство, позволяющее расширить имя типа и сделать его уникальным за счет добавления к началу имени групп символов, разделенных точками. Например, в данном примере компилятор интерпретирует `FileStream` как `System.IO.FileStream`, а `StringBuilder` — как `System.Text.StringBuilder`.

Применять директиву `using` в C# не обязательно, достаточно набрать полное имя типа. Директива `using` заставляет компилятор C# добавить к имени указанный префикс и попытаться найти подходящий тип.

ВНИМАНИЕ

CLR ничего не знает о пространствах имен. При обращении к какому-либо типу среде CLR надо предоставить полное имя типа (а это может быть действительно длинная строка с точками) и сборку, содержащую описание типа, чтобы во время выполнения загрузить эту сборку, найти в ней нужный тип и оперировать им.

В предыдущем примере компилятор должен гарантировать, что каждый упомянутый в коде тип существует и корректно обрабатывается: вызываемые методы существуют, число и типы передаваемых аргументов указаны правильно, значения, возвращаемые методами, обрабатываются надлежащим образом и т. д. Не найдя тип с заданным именем в исходных файлах и перечисленных

сборках, компилятор попытается добавить к имени типа приставку `System.IO` и проверит, совпадает ли полученное имя с существующим типом. Если имя типа опять не обнаружится, он попробует повторить поиск уже с приставкой `System.Text`. Благодаря двум директивам `using`, показанным ранее, я смог ограничиться именами `FileStream` и `StringBuilder` — компилятор автоматически расширяет ссылки до `System.IO.FileStream` и `System.Collections.StringBuilder`. Полагаю, вам понятно, что вводить такой код намного проще, чем первоначальный.

Компилятору надо сообщить при помощи параметра `/reference` (см. главы 2 и 3), в каких сборках искать описание типа. В поисках нужного типа компилятор просмотрит все известные ему сборки. Если подходящая сборка обнаруживается, сведения о ней и типе помещаются в метаданные результирующего управляемого модуля. Для того чтобы информация из сборки была доступна компилятору, надо указать ему сборку, в которой описаны упоминаемые типы. По умолчанию компилятор `C#` автоматически просматривает сборку `mscorlib.dll`, даже если она явно не указана. В ней содержатся описания всех фундаментальных FCL-типов, таких как `Object`, `Int32`, `String` и др.

Легко догадаться, что такой способ обработки пространства имен чреват проблемами, если два (и более) типа с одинаковыми именами находятся в разных сборках. Microsoft настоятельно рекомендует при описании типов применять уникальные имена. Но порой это невозможно. В CLR поощряется повторное использование компонентов. Допустим, в приложении имеются компоненты, созданные в `Microsoft` и `Wintellect`, в которых есть типы с одним названием, например `Widget`. В этом случае процесс формирования имен типов становится неуправляемым, и чтобы различать эти типы, придется указывать в коде их полные имена. При обращении к `Widget` от `Microsoft` надо указать `Microsoft.Widget`, а при ссылке на `Widget` от `Wintellect` — `Wintellect.Widget`. В следующем коде ссылка на `Widget` неоднозначна, и компилятор `C#` выдаст сообщение `error CS0104: 'Widget' is an ambiguous reference` (ошибка CS0104: 'Widget' — неоднозначная ссылка):

```
using Microsoft; // Определяем приставку "Microsoft."
using Wintellect; // Определяем приставку "Wintellect."
```

```
public sealed class Program {
    public static void Main() {
        Widget w = new Widget(); // Неоднозначная ссылка
    }
}
```

Для того чтобы избавиться от неоднозначности, надо явно указать компилятору, какой экземпляр `Widget` требуется создать:

```
using Microsoft; // Определяем приставку "Microsoft."
using Wintellect; // Определяем приставку "Wintellect."
```

```
public sealed class Program {  
    public static void Main() {  
        Wintellect.Widget w = new Wintellect.Widget(); // Неоднозначности нет  
    }  
}
```

В C# есть еще одна форма директивы `using`, позволяющая создать псевдоним для отдельного типа или пространства имен. Она удобна, если требуется несколько типов из пространства имен, но не хочется смешивать в глобальном пространстве имен все используемые типы. Альтернативный способ преодоления неоднозначности следующий:

```
using Microsoft; // Определяем приставку "Microsoft."  
using Wintellect; // Определяем приставку "Wintellect."  
// Опишем символ WintellectWidget как псевдоним для Wintellect.Widget  
using WintellectWidget = Wintellect.Widget;  
public sealed class Program {  
    public static void Main() {  
        WintellectWidget w = new WintellectWidget(); // Ошибки нет  
    }  
}
```

Эти методы устранения неоднозначности хороши, но иногда их недостаточно. Представьте, что компании Australian Boomerang Company (ABC) и Alaskan Boat Corporation (ABC) создали каждая свой тип с именем `BuyProduct` и собираются поместить его в соответствующие сборки. Не исключено, что обе компании создадут пространства имен `ABC`, в которые и включают тип `BuyProduct`. Тот, кто намерен разработать приложение, оперирующее обоими типами, не сдвинется с места, если в языке программирования не окажется механизма, позволяющего различать программными средствами не только пространства имен, но и сборки. К счастью, в компиляторе C# поддерживаются *внешние псевдонимы* (*extern aliases*), позволяющие справиться с проблемой. Внешние псевдонимы дают также возможность обращаться к одному типу двух (или более) версий одной сборки. Подробнее о внешних псевдонимах см. спецификацию языка C#.

При проектировании типов, применяемых в библиотеках, которые могут использоваться третьими лицами, старайтесь описывать эти типы в пространстве имен так, чтобы компиляторы могли без труда преодолеть неоднозначность типов. Вероятность конфликта заметно снизится, если в названии пространства имен верхнего уровня указать полное, а не сокращенное имя компании. В документации .NET Framework SDK Microsoft использует для своих типов пространство имен Microsoft (например: `Microsoft.CSharp`, `Microsoft.VisualBasic` и `Microsoft.Win32`).

Для того чтобы создать пространство имен, достаточно ввести в код его объявление (на C#):

```
namespace CompanyName {  
    public sealed class A {          // TypeDef: CompanyName.A  
    }  
}
```

```
namespace X {  
    public sealed class B { ... } // TypeDef: CompanyName.X.B  
}
```

В комментарии справа от объявления класса указано реальное имя типа, которое компилятор поместит в таблицу метаданных определения типов — с точки зрения CLR это «настоящее» имя типа.

Одни компиляторы вовсе не поддерживают пространства имен, а другие под термином *namespace* понимают нечто иное. В C# директива `namespace` заставляет компилятор добавлять к каждому имени типа определенную приставку — это избавляет программиста от необходимости писать массу лишнего кода.

Связь между сборками и пространством имен

Пространство имен и сборка (файл, в котором реализован тип) не обязательно связаны друг с другом. В частности, различные типы, принадлежащие одному пространству имен, могут быть реализованы в разных сборках. Например, тип `System.IO.FileStream` реализован в сборке `mscorlib.dll`, а тип `System.IO.FileSystemWatcher` — в сборке `System.dll`. На самом деле, сборка `System.IO.dll` в .NET Framework даже не поставляется.

Одна сборка может содержать типы из разных пространств имен. Например, типы `System.Int32` и `System.Text.StringBuilder` находятся в сборке `mscorlib.dll`.

Когда вы посмотрите документацию .NET Framework SDK, то обнаружите, что там четко определено пространство имен, к которому принадлежит тип, и сборка, в которой этот тип реализован. На рис. 4.1 показано, что тип

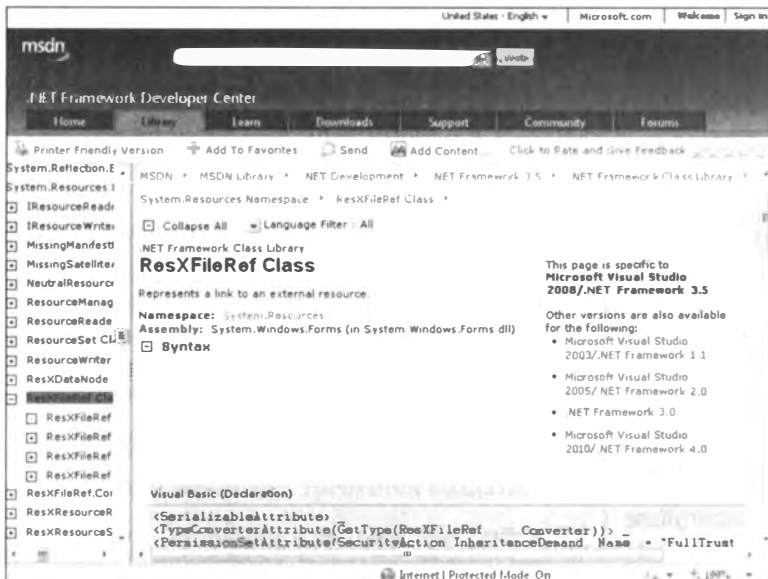


Рис. 4.1. В документации SDK показана связь сборки и пространства имен для типа

ResXFileRef является частью пространства имен System.Resources и реализован в сборке System.Windows.Forms.dll. Для того чтобы скомпилировать код, ссылающийся на тип ResXFileRef, необходимо добавить директиву using System.Resources и использовать параметр компилятора /r:System.Windows.Forms.dll.

Как разные компоненты взаимодействуют во время выполнения

В этом разделе рассказано, как во время выполнения взаимодействуют типы, объекты, стек потока и управляемая куча. Кроме того, объяснено, в чем различие между вызовом статических, экземплярных и виртуальных методов. А начнем мы с некоторых базовых сведений о работе компьютера. То, о чем я собираюсь рассказать, вообще говоря, не является прерогативой CLR, но я начну с общих понятий, а затем перейду к обсуждению информации, относящейся исключительно к CLR.

На рис. 4.2 представлен один процесс Microsoft Windows с загруженной в него исполняющей средой CLR. У процесса может быть много потоков. После создания потоку выделяется стек размером в 1 Мбайт. Выделенная для стека память используется для передачи параметров в методы и хранения определенных в пределах методов локальных переменных. На рис. 4.2 справа показана память стека одного потока. Стеки заполняются от области верхней памяти к области нижней памяти (то есть от старших к младшим адресам). На рисунке поток уже выполняет какой-то код, и в его стеке уже есть какие-то данные (отмечены областью более темного оттенка вверху стека). А теперь представим, что поток выполняет код, вызывающий метод M1.

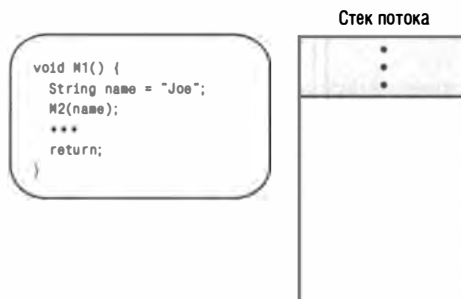


Рис. 4.2. Стек потока перед вызовом метода M1

Все методы, кроме самых простых, содержат некоторый *входной код* (prologue code), инициализирующий метод до начала его работы. Кроме того, эти методы содержат *выходной код* (epilogue code), выполняющий очистку после того, как метод завершит свою основную работу, чтобы вернуть управление вызы-

вающей программе. В начале выполнения метода M1 его входной код выделяет в стеке потока память для локальной переменной name (рис. 4.3).

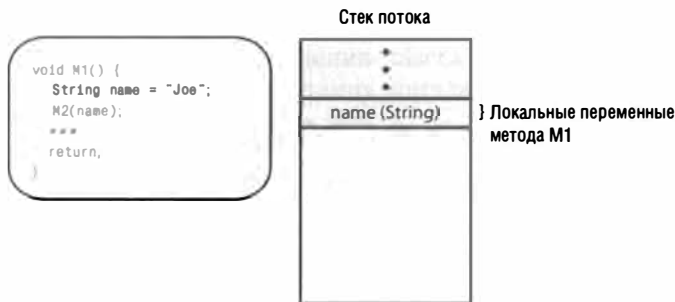


Рис. 4.3. Размещение локальной переменной метода M1 в стеке потока

Далее M1 вызывает метод M2, передавая в качестве аргумента локальную переменную name. При этом адрес локальной переменной name заталкивается в стек (рис. 4.4). Внутри метода M2 местоположение стека хранится в переменной-парамetre s. (Кстати, в некоторых процессорных архитектурах для повышения производительности аргументы передаются через регистры, но это различие для нашего обсуждения несущественно.) При вызове метода адрес возврата в вызывающий метод также заталкивается в стек (показано на рис. 4.4).



Рис. 4.4. При вызове M2 метод M1 заталкивает аргументы и адрес возврата в стек потока

В начале выполнения метода M2 его входной код выделяет в стеке потока память для локальных переменных length и tally (рис. 4.5). Затем выполняется код метода M2. В конце концов, выполнение M2 доходит до команды возврата, которая записывает в указатель команд процессора адрес возврата из стека, и стековый фрейм M2 возвращается в состояние, показанное на рис. 4.3. С этого момента продолжается выполнение кода M1, который следует сразу за вызовом M2, а стековый фрейм метода находится в состоянии, необходимом для работы M1.

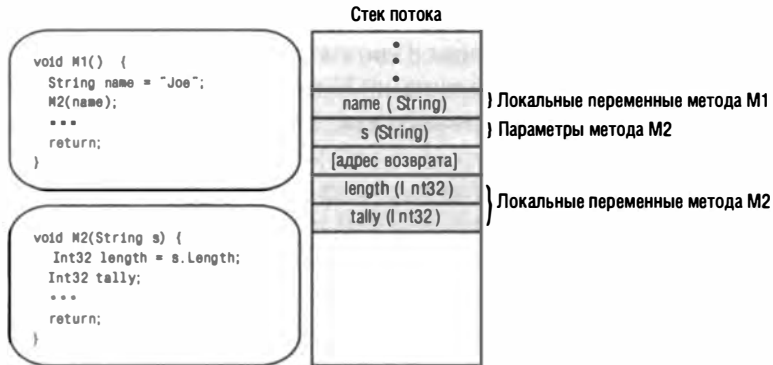


Рис. 4.5. Выделение в стеке потока памяти для локальных переменных метода M2

В конечном счете, метод M1 возвращает управление вызывающей программе, устанавливая указатель команд процессора на адрес возврата (на рисунках не показан, но в стеке он находится прямо над аргументом name), и стековый фрейм M1 возвращается в состояние, показанное на рис. 4.2. С этого момента продолжается выполнение кода вызвавшего метода, причем начинает выполняться код, непосредственно следующий за вызовом M1, а стековый фрейм вызвавшего метода находится в состоянии, необходимом для его работы.

А сейчас давайте переключим внимание на исполняющую среду CLR. Допустим, есть следующие два определения классов:

```

internal class Employee {
    public Int32 GetYearsEmployed () { ... }
    public virtual String GetProgressReport () { ... }
    public static Employee Lookup(String name) { ... }
}
internal sealed class Manager : Employee {
    public override String GenProgressReport() { ... }
}

```

Процесс Windows запустился, в него загружена среда CLR, инициализирована управляемая куча, и создан поток (вместе с его 1 Мбайт памяти в стеке). Поток уже выполняет какой-то код, из которого вызывается метод M3 (рис. 4.6). Метод M3 содержит код, призванный продемонстрировать, как работает CLR; этот код необычный в том смысле, что, в сущности, не делает ничего полезного.

В процессе преобразования IL-кода метода M3 в машинные команды JIT-компилятор выявляет все типы, на которые есть ссылки в M3, — это типы Employee, Int32, Manager и String (из-за наличия строки "Joe"). На данном этапе CLR обеспечивает загрузку в домен приложений всех сборок, в которых определены все эти типы. Затем, используя метаданные сборки, CLR получает информацию о типах и создает структуры данных, собственно и представляю-

шие эти типы. Структуры данных для объектов типа `Employee` и `Manager` показаны на рис. 4.7. Поскольку до вызова `M3` поток уже выполнил какой-то код, для простоты допустим, что объекты типа `Int32` и `String` уже созданы (что вполне возможно, так как это часто используемые типы), поэтому они не показаны на рисунке.

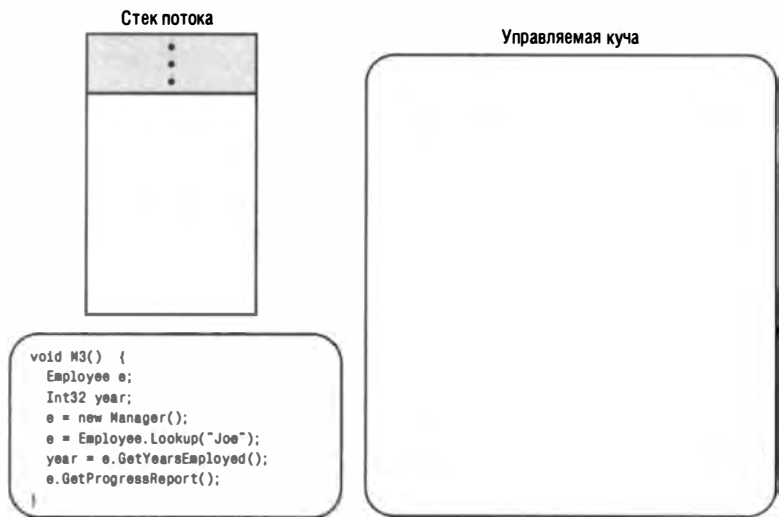


Рис. 4.6. Среда CLR загружена в процесс, куча инициализирована, готовится вызов стека потока, в который загружен метод `M3`

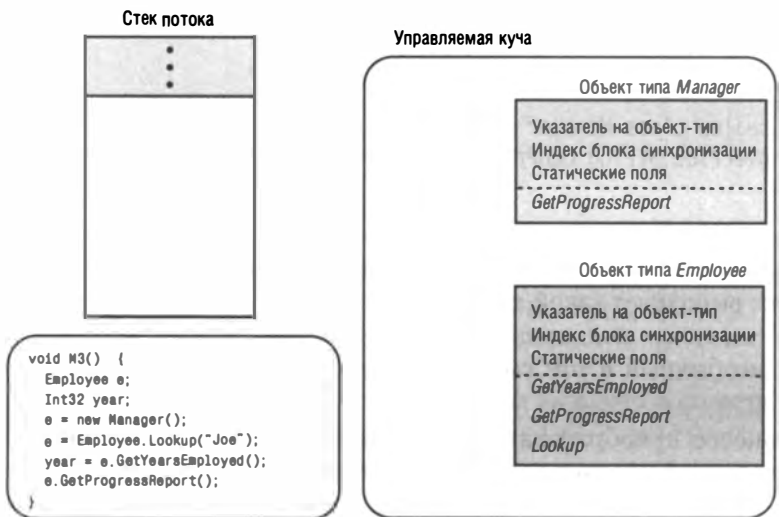


Рис. 4.7. При вызове `M3` создаются объекты типа `Employee` и `Manager`

На минуту отвлечемся на обсуждение этих объектов-типов. Как говорилось ранее в этой главе, все объекты в куче содержат два дополнительных члена: указатель на объект-тип и индекс блока синхронизации. У объектов типа `Employee` и `Manager` оба эти члена есть. Определяя тип, можно создать в них статические поля данных. Байты для этих статических полей выделяются в составе самих объектов-типов. Наконец, у каждого объекта-типа есть таблица методов с входными точками всех методов, определенных в типе. Эта таблица методов уже обсуждалась в главе 1. Так как в типе `Employee` определены три метода (`GetYearsEmployed`, `GenProgressReport` и `Lookup`), в соответствующей таблице методов есть три записи. В типе `Manager` определен один метод (переопределенный метод `GenProgressReport`), который и представлен в таблице методов этого типа.

После того как среда CLR позаботится о создании всех необходимых для метода объектов-типов и компиляции кода метода М3, исполняющая среда приступает к выполнению машинного кода М3. При выполнении входного кода М3 в стеке потока выделяется память для локальных переменных (рис. 4.8). Между прочим, CLR автоматически инициализирует все локальные переменные значением `null` или 0 (нулем) — это делается в рамках выполнения входного кода метода. Однако компилятор C# генерирует сообщение об ошибке `Use of assigned local variable` (использование выделяемой локальной переменной) в случае, если вы попытаетесь обратиться к локальной переменной, неявно инициализированной в вашем коде.

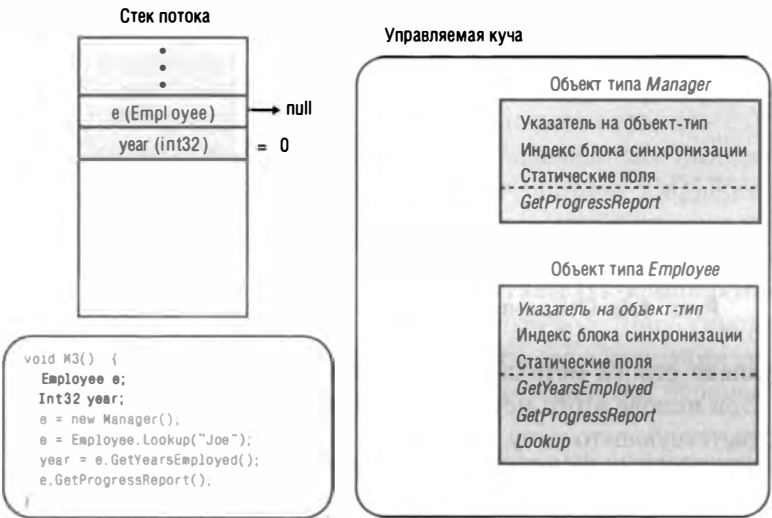


Рис. 4.8. Выделение памяти в стеке потока для локальных переменных метода М3

Далее М3 выполняет код создания объекта Manager. При этом в управляемой куче создается экземпляр типа Manager, объект Manager (рис. 4.9). У объекта Manager — так же как и всех остальных объектов — есть указатель на объект-тип и член SyncBlockIndex. У этого объекта тоже есть байты, необходимые для размещения всех экземплярных полей данных, определенные в типе Manager, а также всех экземплярных полей, определенных во всех базовых классах типа Manager (в данном случае — Employee и Object). Всякий раз при создании нового объекта в куче CLR автоматически инициализирует внутренний член-указатель на объект-тип так, чтобы он указывал на соответствующий объект-тип объекта (в данном случае — на объект-тип Manager). Кроме того, CLR инициализирует индекс блока синхронизации (SyncBlockIndex) и присваивает всем экземплярным полям объекта значение null или 0 (нуль) перед вызовом конструктора типа — метода, который, скорее всего, изменит значения некоторых экземплярных полей. Оператор new возвращает адрес в памяти объекта Manager, который хранится в переменной e (в стеке потока).

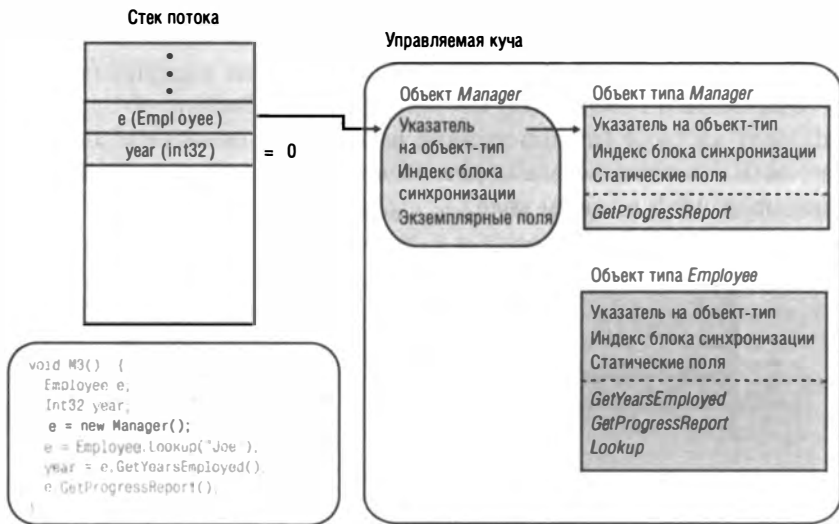


Рис. 4.9. Создание и инициализация объекта Manager

Следующая строка метода М3 вызывает статический метод `Lookup` объекта `Employee`. При вызове этого метода CLR определяет местонахождение объекта-типа, соответствующего типу, в котором определен статический метод. Затем на основании таблицы методов объекта-типа среда CLR находит точку входа в вызываемый метод, обрабатывает код JIT-компилятором (при необходимости) и вызывает полученный машинный код. Для нашего обсуждения достаточно предположить, что метод `Lookup` объекта `Employee` выполняет запрос к базе данных, чтобы найти сведения о Joe. Допустим также, что в базе данных указано, что Joe занимает должность менеджера, поэтому код метода `Lookup` создает в куче новый объект `Manager`, инициализирует его данными Joe и воз-

возвращает адрес готового объекта. Адрес размещается в локальной переменной `e`. Результат этой операции показан на рис. 4.10.

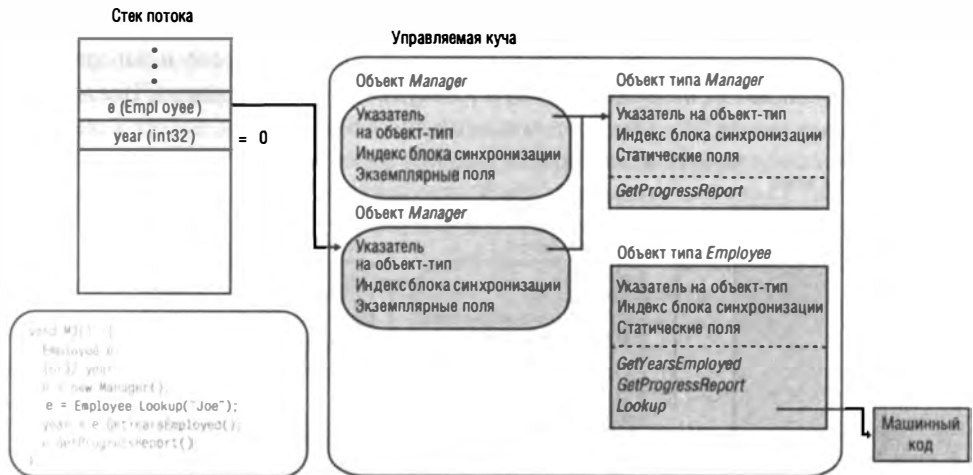


Рис. 4.10. Статический метод `Lookup` в `Employee` выделяет память и инициализирует объект `Manager` для `Joe`

Заметьте, что `e` больше не ссылается на первый созданный объект `Manager`. Поскольку нет переменных, ссылающихся на этот объект, он становится подходящим клиентом для механизма сборки мусора, который на следующем проходе освободит занятую объектом память.

Следующая строка метода `M3` вызывает неvirtуальный экземплярный метод `GetYearsEmployed` объекта `Employee`. При этом CLR определяет местонахождение объекта типа, соответствующего типу переменной, использованной для вызова. В данном случае переменная `e` определена как `Employee`. (Если бы вызываемый метод не был определен в типе `Employee`, в процессе поиска метода среда CLR начала бы последовательно просматривать классы иерархии — вплоть до `Object`.) Далее CLR находит в таблице методов объекта-типа запись о входе в вызываемый метод, обрабатывает код JIT-компилятором (при необходимости) и вызывает полученный машинный код. Допустим, что метод `GetYearsEmployed` возвращает 5, то есть стаж работы `Joe` в компании составляет пять лет. Полученное целое число размещается в локальной переменной `year` (рис. 4.11).

Следующая строка метода `M3` вызывает виртуальный экземплярный метод `GenProgressReport` в `Employee`. При вызове виртуального экземплярного метода CLR приходится выполнять некоторую дополнительную работу. Во-первых, CLR обращается к переменной, используемой для вызова, и затем следует по адресу вызывающего объекта. В данном случае переменная `e` указывает на объект `Joe` типа `Manager`. Во-вторых, CLR проверяет у объекта внутренний член-указатель на объект-тип; этот член ссылается на фактический тип объекта.

Затем CLR находит в таблице методов объекта-типа запись о входе в вызываемый метод, обрабатывает код JIT-компилятором (при необходимости) и вызывает полученный машинный код. Допустим, что метод `GetYearsEmployed` возвращает 5, то есть стаж работы Joe в компании составляет пять лет. В нашем случае вызывается реализация метода `GenProgressReport` в `Manager`, потому что е ссылается на объект `Manager`. Результат этой операции показан на рис. 4.12.

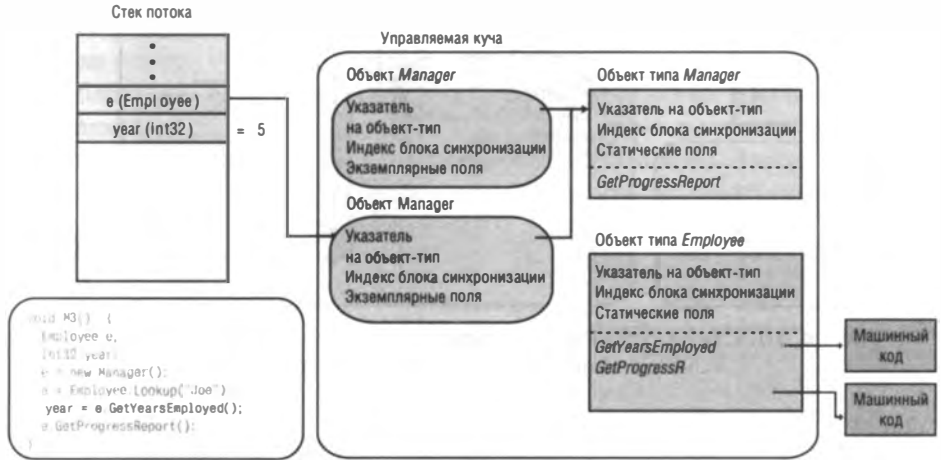


Рис. 4.11. Невиртуальный экземплярный метод `GetYearsEmployed` в `Employee` возвращает значение 5

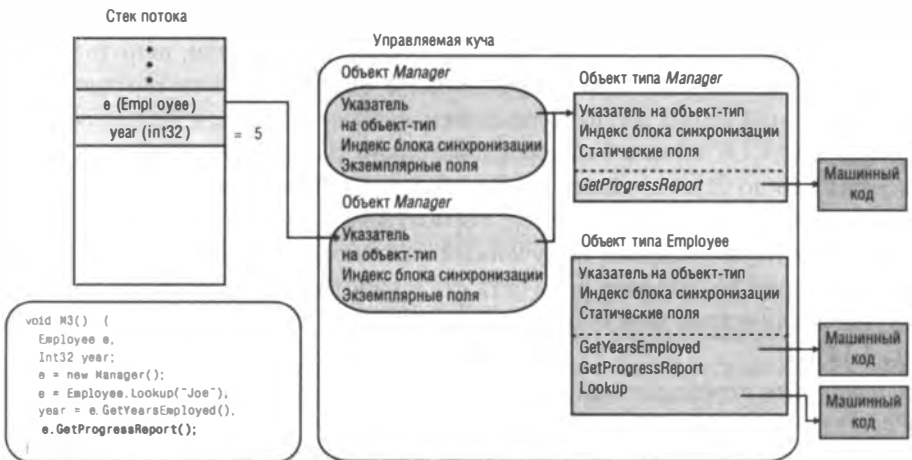


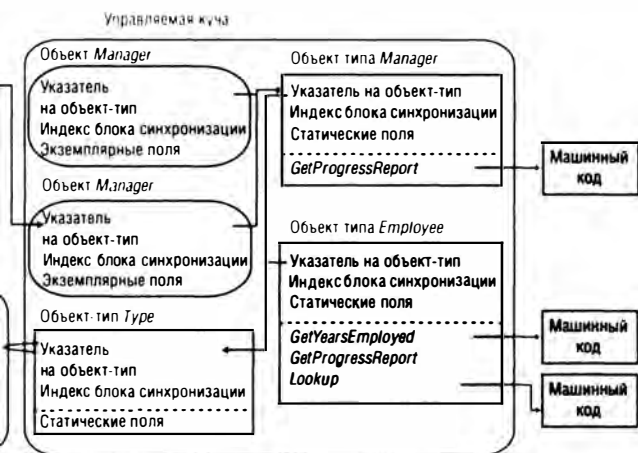
Рис. 4.12. При вызове виртуального метода `GenProgressReport` экземпляра `Employee` будет вызвана переопределенная реализация этого метода в `Manager`

е внимание, что объекты типа Employee и Manager содержат объекты-типы. Причина в том, что объекты-типы — это, объекты. Создавая объекты-типы, среда CLR должна их и спросить: «Какие значения будут присвоены при м, при своем запуске в процессе CLR сразу же создает для типа System.Type (он определен в MSCorLib.dll). Manager являются «экземплярами» этого типа, и по ли на объекты-типы инициализируются так, чтобы System.Type (рис. 4.13).

то Joe — это всего лишь оуе, член-указатель на /ее, это приведет к тому, ее, а не в Manager.

дным текстом, IL и ма-ентах и локальных пере-ле ссылаются на объекты (ранят указатель на свой методов). Мы обсудили, те и виртуальные экзе-млее полную картину ра-тировании и реализации , я хотел бы сказать еще

а Employee и Manager содержат что объекты-типы — это, ы, среда CLR должна их ия будут присвоены при ссе CLR сразу же создает (ределен в MSCorLib.dll). ярами» этого типа, и по ализируются так, чтобы



Объекты типа Manager и Employee как экземпляры типа System.Type

System.Type сам является объектом и поэтому также объект

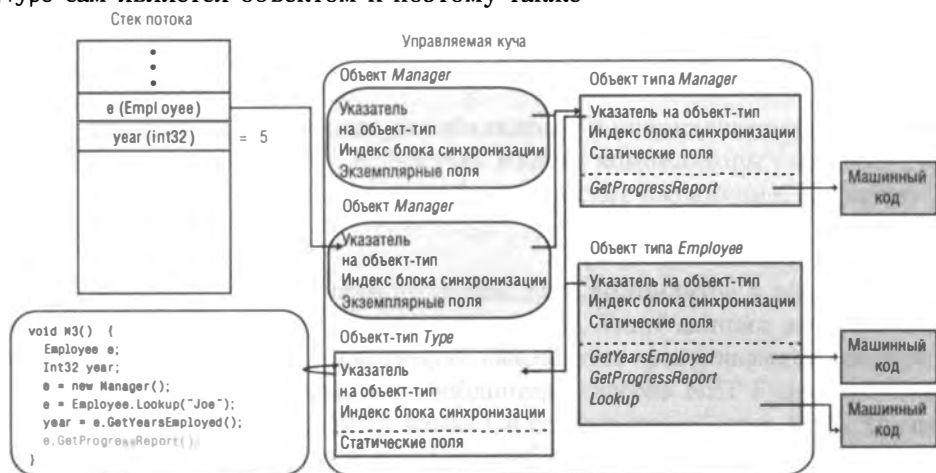


Рис. 4.13. Объекты типа Manager и Employee как экземпляры типа System.Type

Конечно, объект-тип System.Type сам является объектом и поэтому также имеет член-указатель на объект-тип; значит, закономерно поинтересоваться,

на что ссылается этот член. А ссылается он на самого себя, так как объект-тип `System.Type` сам по себе является «экземпляром» объекта-типа. Теперь становится понятно, как устроена и работает вся система типов в CLR. Кстати, метод `GetType` типа `System.Object` просто возвращает адрес, хранящийся в указателе на объект-тип заданного объекта. Иначе говоря, метод `GetType` возвращает указатель на объект-тип объекта, и именно поэтому можно определить истинный тип любого объекта в системе (включая объекты-типы).

Глава 5. Примитивные, ссылочные и значимые типы

В этой главе речь идет о разновидностях типов, с которыми вы будете иметь дело при программировании для платформы Microsoft .NET Framework. Важно, чтобы все разработчики четко осознавали разницу в поведении типов. Приступая к изучению .NET Framework, я толком не понимал, в чем разница между примитивными, ссылочными и значимыми типами, в результате мой код получался не слишком эффективным и изобиловал трудно вылавливаемыми «жучками». Надеюсь, мой опыт и мои объяснения различий между этими типами помогут вам избавиться от лишней головной боли и повысить производительность своей работы.

Примитивные типы в языках программирования

Некоторые типы данных применяются так часто, что для работы с ними во многих компиляторах предусмотрен упрощенный синтаксис. Например, целую переменную можно создать следующим образом:

```
System.Int32 a = new System.Int32();
```

Конечно, подобный синтаксис для объявления и инициализации целой переменной кажется громоздким. К счастью, многие компиляторы (включая C#) позволяют использовать вместо этого более простые выражения, например:

```
int a = 0;
```

Подобный код читается намного лучше, да и компилятор в обоих случаях генерирует идентичный IL-код для System.Int32. Типы данных, которые поддерживаются компилятором напрямую, называются *примитивными* (primitive types) и отображаются им в типы из библиотеки классов .NET Framework Class Library (FCL). Например, типу int языка C# соответствует System.Int32. Значит, весь следующий код компилируется без ошибок и преобразуется в одинаковые IL-команды:

```
int a = 0;           // Самый удобный синтаксис
System.Int32 a = 0;  // Удобный синтаксис
int a = new int();   // Неудобный синтаксис
System.Int32 a = new System.Int32(); // Самый неудобный синтаксис
```

В табл. 5.1 представлены типы FCL и соответствующие им примитивные типы C#. В других языках типам, удовлетворяющим общезыковой спецификации (Common Language Specification, CLS), соответствуют аналогичные примитивные типы. Однако поддержка языком типов, не удовлетворяющих требованиям CLS, не обязательна.

Таблица 5.1. Примитивные типы C# и соответствующие типы FCL

Примитивный тип	FCL тип	Совместимость с CLS	Описание
sbyte	System.Sbyte	Нет	8-разрядное значение со знаком
byte	System.Byte	Да	8-разрядное значение без знака
short	System.Int16	Да	16-разрядное значение со знаком
ushort	System.UInt16	Нет	16-разрядное значение без знака
int	System.Int32	Да	32-разрядное значение со знаком
uint	System.UInt32	Нет	32-разрядное значение без знака
long	System.Int64	Да	64-разрядное значение со знаком
ulong	System.UInt64	Нет	64-разрядное значение без знака
char	System.Char	Да	16-разрядный символ Unicode (char никогда не представляет 8-разрядное значение, как в неуправляемом коде на C++)
float	System.Single	Да	32-разрядное значение с плавающей точкой в стандарте IEEE
double	System.Double	Да	64-разрядное значение с плавающей точкой в стандарте IEEE
bool	System.Boolean	Да	Булево значение (true или false)
decimal	System.Decimal	Да	128-разрядное значение с плавающей точкой повышенной точности, часто используемое для финансовых расчетов, где недопустимы ошибки округления. Один разряд числа — это знак, в следующих 96 разрядах помещается само значение, следующие 8 разрядов — степень числа 10, на которое делится 96-разрядное число (может быть в диапазоне от 0 до 28). Остальные разряды не используются
string	System.String	Да	Массив символов
object	System.Object	Да	Базовый тип для всех типов

Примитив- ный тип	FCL тип	Совмести- мость с CLS	Описание
dynamic	System.Object	Да	Для CLR тип dynamic идентичен типу object. Однако компилятор C# позволяет переменным типа dynamic участвовать в динамической отправке, используя упрощенный синтаксис. Подробнее об этом читайте в разделе «Примитивный тип данных dynamic» в конце этой главы

Иначе говоря, можно считать, что компилятор C# автоматически предполагает, что во всех файлах исходного кода есть следующие директивы using (как говорилось в главе 4):

```
using sbyte = System.SByte;  
using byte = System.Byte;  
using short = System.Int16;  
using ushort = System.UInt16;  
using int = System.Int32;  
using uint = System.UInt32;  
...
```

Я не могу согласиться со следующим утверждением из спецификации языка C#: «С точки зрения стиля программирования предпочтительней использовать ключевое слово, а не полное системное имя типа», поэтому стараюсь задействовать имена FCL-типов и избегать имен примитивных типов. На самом деле, мне бы хотелось, чтобы имен примитивных типов не было совсем, а разработчики употребляли только имена FCL-типов. И вот по каким причинам.

- ❑ Мне попадались разработчики, не знавшие, какое ключевое слово использовать им в коде, `string` или `String`. В C# это не важно, так как ключевое слово `string` в точности преобразуется в FCL-тип `System.String`. Я также слышал, что некоторые разработчики говорили о том, что в 32-разрядных операционных системах тип `int` представлялся 32-разрядным типом, а в 64-разрядных — 64-разрядным типом. Это утверждение совершенно неверно: в C# тип `int` всегда преобразуется в `System.Int32`, поэтому он всегда представляется 32-разрядным типом безотносительно запущенной операционной системы. Использование ключевого слова `Int32` в своем коде тоже позволяет избежать этого недопонимания.
- ❑ В C# `long` отображается на `System.Int64`, но в другом языке это может быть `Int16` или `Int32`. Как известно, в C++/CLI тип `long` трактуется как `Int32`. Если кто-то возьмется читать код, написанный на новом для себя языке, то назначение кода может быть неверно им истолковано. Многим языкам незнакомо ключевое слово `long`, и их компиляторы не пропустят код, где оно встречается.

- ❑ У многих FCL-типов есть методы, в имена которых включены имена типов. Например, у типа `BinaryReader` есть методы `ReadBoolean`, `ReadInt32`, `ReadSingle` и т. д., а у типа `System.Convert` — методы `ToBoolean`, `ToInt32`, `ToSingle` и т. д. Вот вполне приемлемый код, в котором строка, содержащая `float`, кажется неестественной и не очевидно, что код корректный:

```
BinaryReader br = new BinaryReader(...);
float val = br.ReadSingle(); // Код правильный, но выглядит странно
Single val = br.ReadSingle(); // Код правильный и выглядит нормально
```

- ❑ Многие программисты, пишущие исключительно на C#, часто забывают, что в CLR могут применяться и другие языки программирования. Например, среда FCL практически полностью написана на C#, и разработчики из команды FCL сейчас предоставляют такие методы, как метод `GetLongLength` класса `Array`, возвращающий значение `Int64`, которое имеет тип `long` в C#, но не в других языках программирования (например, C++/CLI). Другой пример — метод `LongCount` класса `System.Linq.Enumerable`.

По этим причинам я буду использовать в этой книге только имена FCL-типов.

Скорее всего, следующий код во многих языках благополучно скомпилируется и выполнится:

```
Int32 i = 5; // 32-разрядное число
Int64 l = i; // Неявное приведение типа к 64-разрядному значению
```

Однако если вспомнить, что говорилось о приведении типов в главе 4, можно решить, что он компилироваться не будет. Все-таки `System.Int32` и `System.Int64` — разные типы и не приводятся друг к другу. Могу вас обнадежить: код успешно компилируется и делает все, что ему положено. Объясню, почему.

Дело в том, что компилятор C# неплохо разбирается в примитивных типах и применяет свои правила при компиляции кода. Иначе говоря, он распознает наиболее распространенные эталоны программирования и генерирует такие IL-команды, благодаря которым исходный код работает так, как требуется. В первую очередь, это относится к приведению типов, литералам и операторам, примеры которых мы рассмотрим позже.

Начнем с того, что компилятор выполняет явное и неявное приведение между примитивными типами, например:

```
Int32 i = 5;           // Неявное приведение Int32 к Int32
Int64 l = i;           // Неявное приведение Int32 к Int64
Single s = i;          // Неявное приведение Int32 к Single
Byte b = (Byte) i;     // Явное приведение Int32 к Byte
Int16 v = (Int16) s;   // Явное приведение Single к Int16
```

C# разрешает неявное приведение типа, если это преобразование «безопасно», то есть не сопряжено с потерей данных; пример — преобразование из `Int32`

в Int64. Однако для преобразования с риском потери данных C# требует явного приведения типа. Для числовых типов «небезопасное» преобразование означает «связанное с потерей точности или величины числа». Например, преобразование из Int32 в Byte требует явного приведения к типу, так как при больших величинах Int32 теряется точность; требует приведения и преобразование из Single в Int64, поскольку число Single может оказаться больше, чем допустимо для Int64.

Для реализации приведения разные компиляторы могут порождать разный код. Например, в случае приведения числа 6,8 типа Single к типу Int32 одни компиляторы создадут код, который поместит в Int32 число 6, а другие округлят результат до 7. Между прочим, в C# дробная часть всегда отбрасывается. Точные правила приведения для примитивных типов вы найдете в разделе спецификаций языка C#, посвященном преобразованиям («Conversions»).

Помимо приведения, компилятор «знает» и о другой особенности примитивных типов: к ним применима литеральная форма записи. Литералы сами по себе считаются экземплярами типа, поэтому можно вызывать экземплярные методы, например, следующим образом:

```
Console.WriteLine(123.ToString() + 456.ToString()); // "123456"
```

Кроме того, благодаря тому, что выражения, состоящие из литералов, вычисляются на этапе компиляции, возрастает скорость выполнения приложения.

```
Boolean found = false; // В готовом коде found присваивается 0
Int32 x = 100 + 20 + 3; // В готовом коде x присваивается 123
String s = "a " + "bc"; // В готовом коде s присваивается "a bc"
```

И наконец, компилятор «знает», как и в каком порядке интерпретировать встретившиеся в коде операторы (в том числе +, -, *, /, %, &, ^, |, ==, !=, >, <, >=, <=, <<, >>, ~, !, ++, -- и т. п.):

```
Int32 x = 100; // Оператор присваивания
Int32 y = x + 23; // Операторы суммирования и присваивания
Boolean lessThanFifty = (y < 50); // Операторы "меньше чем" и присваивания
```

Проверяемые и непроверяемые операции для примитивных типов

Программистам должно быть хорошо известно, что многие арифметические операции над примитивными типами могут привести к переполнению:

```
Byte b = 100;
b = (Byte) (b + 200); // После этого b равно 44 (или в шестнадцатеричной системе - 2C)
```

ВНИМАНИЕ

Арифметические операции в CLR выполняются только над 32- и 64-разрядными числами. Поэтому `b` и 200 сначала преобразуются в 32-разрядные (или в 64-разрядные, если хотя бы одному из операндов недостаточно 32 разрядов) значения, а затем уже суммируются. Поэтому 200 и `b` (их размер меньше 32-разрядов) преобразуются в 32-разрядные значения и суммируются. Полученное 32-разрядное число, прежде чем поместить его обратно в переменную `b`, нужно привести к типу `Byte`. Так как в данном случае `C#` не выполняет неявного приведения типа, во вторую строку потребовалось ввести операцию приведения к типу `Byte`.

Такое переполнение «втихую» обычно в программировании не приветствуется, и если его не выявить, приложение будет вести себя непредсказуемо. Изредка, правда (например, при вычислении хэшей или контрольных сумм), такое переполнение не только приемлемо, но и желательно.

В каждом языке — свои способы обработки переполнения. В `C` и `C++` переполнение ошибкой не считается и разрешается усекать значения — приложение не прервет свою работу. А вот в `Visual Basic` переполнение всегда рассматривается как ошибка, и при его обнаружении генерируется исключение.

В CLR есть IL-команды, позволяющие компилятору по-разному реагировать на переполнение. Например, суммирование двух чисел выполняет команда `add`, не реагирующая на переполнение, а также команда `add.ovf`, которая при переполнении генерирует исключение `System.OverflowException`. Кроме того, в CLR есть аналогичные IL-команды для вычитания (`sub/sub.ovf`), умножения (`mul/mul.ovf`) и преобразования данных (`conv/conv.ovf`).

Пишущий на `C#` программист может сам решать, как обрабатывать переполнение; по умолчанию проверка переполнения отключена. Это значит, что компилятор генерирует для операций сложения, вычитания, умножения и преобразования IL-команды без проверки переполнения. В результате код выполняется быстро, но разработчик должен быть либо уверен в отсутствии переполнения, либо специально предусмотреть возможность его возникновения.

Чтобы включить механизм управления процессом обработки переполнения на этапе компиляции, добавьте в командную строку компилятора параметр `/checked+`. Он сообщает компилятору, что для выполнения сложения, вычитания, умножения и преобразования должны быть сгенерированы IL-команды с проверкой переполнения. Такой код медленнее, так как CLR тратит время на проверку этих операций, ожидая переполнения. Когда оно возникает, CLR генерирует исключение `OverflowException`. Код приложения должен предусматривать корректную обработку этого исключения.

Однако программистам вряд ли понравится необходимость включения или отключения режима проверки переполнения во всем коде. Им лучше самим решать, как реагировать на переполнение в каждом конкретном случае. И `C#` предлагает такой механизм гибкого управления проверкой в виде операторов

checked и unchecked. Например (предполагается, что компилятор по умолчанию создает код без проверки):

```
UInt32 invalid = unchecked((UInt32) -1); // OK
```

А вот пример с использованием оператора checked (предполагается, что компилятор по умолчанию создает код без проверки):

```
Byte b = 100; b = checked((Byte) (b + 200)); // Вбрасывается  
// исключение OverflowException
```

Здесь b и 200 преобразуются в 32-разрядные числа и суммируются; результат равен 300. Затем при преобразовании 300 в Byte генерируется исключение OverflowException. Если приведение к типу Byte вывести из оператора checked, исключения не будет:

```
b = (Byte) checked(b + 200); // b содержит 44; нет OverflowException
```

Наряду с операторами checked и unchecked в C# есть одноименные инструкции, позволяющие включить проверяемые или непроверяемые выражения внутри блока:

```
checked { // Начало проверяемого блока  
    Byte b = 100;  
    b = (Byte) (b + 200); // Это выражение проверяется на переполнение  
} // Конец проверяемого блока
```

Кстати, внутри такого блока можно задействовать оператор +=, который немного упростит код:

```
checked { // Начало проверяемого блока  
    Byte b = 100;  
    b += 200; // Это выражение проверяется на переполнение  
} // Конец проверяемого блока
```

ВНИМАНИЕ

Установка режима контроля переполнения не влияет на работу метода, вызываемого внутри оператора или инструкции checked, так как действие оператора (и инструкции) checked распространяется только на выбор IL-команд сложения, вычитания, умножения и преобразования данных. Например:

```
checked {  
    // Предположим, SomeMethod пытается поместить 400 в Byte  
    SomeMethod(400);  
    // Вбрасывание OverflowException в SomeMethod  
    // зависит от наличия в нем операторов проверки  
}
```

Я наблюдал большое количество вычислений, генерирующих непредсказуемые результаты. Обычно это случается из-за неправильного ввода данных поль-

зователем или же из-за возвращения неожиданных значений переменных. Итак, при использовании операторов `checked` и `unchecked` учитывайте следующее.

- ❑ Используйте типы со знаком (`Int32` и `Int64`) вместо числовых типов без знака (`UInt32` и `UInt64`) везде, где это возможно. Это позволит компилятору выявлять ошибку переполнения. К тому же числовые типы без знака несовместимы с CLS.
- ❑ Включайте в блок `checked` ту часть кода, в которой возможно переполнение из-за неверных входных данных, например при обработке запросов, содержащих данные, предоставленные конечным пользователем или клиентской машиной.
- ❑ Включайте в блок `unchecked` ту часть кода, в которой переполнение не является проблемой, например при вычислении контрольной суммы.
- ❑ Для кода, где нет операторов и блоков `checked` и `unchecked`, делают предположение, что генерация исключения *необходима* при переполнении, например при вычислении (скажем, простых чисел), когда входные данные известны и переполнение считается ошибкой.

При отладке кода установите параметр компилятора `/checked+`. Выполнение приложения замедлится, так как система будет контролировать переполнение во всем коде, не помеченном ключевыми словами `checked` или `unchecked`. Обнаружив исключение, вы сможете исправить ошибку. При окончательной сборке приложения установите параметр `/checked-`, что ускорит выполнение приложения, а исключения генерироваться не будут. Для того чтобы изменить значение параметра `checked` в Microsoft Visual Studio, откройте окно свойств вашего проекта, перейдите на вкладку `Build`, щелкните на кнопке `Advanced` и установите флажок `Check For Arithmetic Overflow/underflow`, как это показано на рис. 5.1.

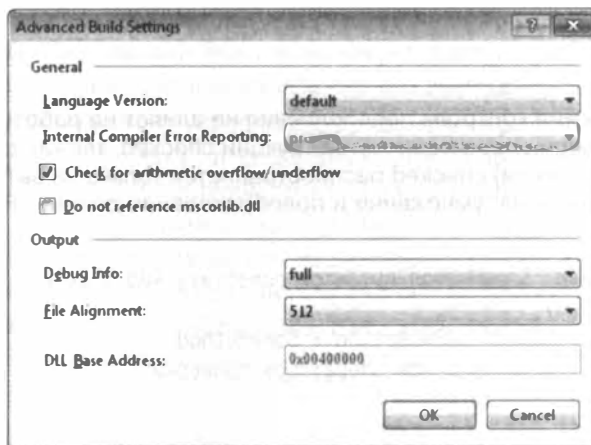


Рис. 5.1. Изменение применяемых по умолчанию параметров компилятора Visual Studio в окне Advanced Build Settings

В случае если для вашего приложения производительность не играет главной роли, я рекомендую всегда оставлять параметр `/checked` включенным. Это позволит защитить приложение от некорректных данных и брешей в системе безопасности. Например, при вычислении значения индекса в некотором массиве данных вам лучше всего использовать исключение `OverflowException`.

ВНИМАНИЕ

Тип `System.Decimal` стоит особняком. В отличие от многих языков программирования (включая `C#` и `Visual Basic`), в CLR тип `Decimal` не относится к примитивным типам. В CLR нет IL-команд для работы со значениями типа `Decimal`. В документации по `.NET Framework` сказано, что тип `Decimal` имеет открытые статические методы-члены `Add`, `Subtract`, `Multiply`, `Divide` и прочие, а также перегруженные операторы `+`, `-`, `*`, `/` и т. д.

При компиляции кода с типом `Decimal` компилятор генерирует вызовы членов `Decimal`, которые и выполняют реальную работу. Поэтому значения типа `Decimal` обрабатываются медленнее примитивных CLR-типов. Кроме того, раз нет IL-команд для манипуляции числами типа `Decimal`, то не будут иметь эффекта ни операторы `checked` и `unchecked`, ни соответствующие параметры командной строки компилятора. И любая «небезопасная» операция над типом `Decimal` обязательно вызовет исключение `OverflowException`.

Аналогично, тип `System.Numeric.BigInteger` используется в массивах `UInt32` для того, чтобы представить произвольные большие целочисленные переменные не больше и не меньше граничных. Следовательно, операции с типом `BigInteger` никогда не вызовут исключения `OverflowException`. Однако они могут выбросить исключение `OutOfMemoryException`, если значение переменной окажется слишком большим.

Ссылочные и значимые типы

CLR поддерживает две разновидности типов: *ссылочные* (reference types) и *значимые* (value types). Большинство типов в FCL — ссылочные, но программисты чаще всего используют значимые. Память для ссылочных типов всегда выделяется из управляемой кучи, а оператор `C# new` возвращает адрес в памяти, где размещается сам объект. При работе со ссылочными типами имейте в виду следующие обстоятельства, относящиеся к производительности приложения:

- ❑ память для ссылочных типов всегда выделяется из управляемой кучи;
- ❑ каждый объект, размещаемый в куче, имеет некоторые дополнительные члены, подлежащие инициализации;
- ❑ незанятые полезной информацией байты объекта обнуляются (это касается полей);
- ❑ размещение объекта в управляемой куче со временем инициирует сборку мусора.

Если бы все типы были ссылочными, эффективность приложения резко упала бы. Представьте, насколько замедлилось бы выполнение приложения, если бы при каждом обращении к значению типа `Int32` выделялась память! Поэтому, чтобы ускорить обработку простых, часто используемых типов, CLR предлагает «облегченные» типы — *значимые*. Экземпляры этих типов обычно размещаются в стеке потока (хотя они могут быть встроены и в объект ссылочного типа). В представляющей экземпляр переменной нет указателя на экземпляр; поля экземпляра размещаются в самой переменной. Поскольку переменная содержит поля экземпляра, то для работы с экземпляром не нужно выполнять разыменовывание (*dereference*) экземпляра. Благодаря тому, что экземпляры значимых типов не обрабатываются сборщиком мусора, уменьшается интенсивность работы с управляемой кучей и сокращается количество коллекций (*collections*), требуемых приложению на протяжении его существования.

В документации на .NET Framework можно сразу увидеть, какие типы относят к ссылочным, а какие — к значимым. Если тип называют *классом* (*class*), речь идет о ссылочном типе. Например, классы `System.Object`, `System.Exception`, `System.IO.FileStream` и `System.Random` — это ссылочные типы. В свою очередь, значимые типы в документации называют *структурами* (*structure*) и *перечислениями* (*enumeration*). Например, структуры `System.Int32`, `System.Boolean`, `System.Decimal`, `System.TimeSpan` и перечисления `System.DayOfWeek`, `System.IO.FileAttributes` и `System.Drawing.FontStyle` являются значимыми типами.

При внимательном знакомстве с документацией можно заметить, что все структуры являются прямыми потомками абстрактного типа `System.ValueType`, который, в свою очередь, является производным от типа `System.Object`. По умолчанию все значимые типы должны быть производными от `System.ValueType`. Все перечисления являются производными от типа `System.Enum`, производного от `System.ValueType`. CLR и языки программирования по-разному интерпретируют перечисления. О перечислимых типах см. главу 15.

При определении собственного значимого типа нельзя выбрать произвольный базовый тип, однако значимый тип может реализовать один или несколько выбранных вами интерфейсов. Кроме того, в CLR значимый тип является изолированным, то есть он не может служить базовым типом для какого-либо другого ссылочного или значимого типа. Поэтому, например, нельзя в описании нового типа указывать в качестве базовых типы `Boolean`, `Char`, `Int32`, `UInt64`, `Single`, `Double`, `Decimal` и т. д.

ВНИМАНИЕ

Многим разработчикам (в частности, тем, кто пишет неуправляемый код на C/C++) деление на ссылочные и значимые типы сначала будет казаться странным. В неуправляемом коде C/C++ вы объявляете тип, и уже код решает, куда поместить экземпляр типа: в стек потока или в кучу приложения. В управляемом коде иначе: разработчик, описывающий тип, указывает, где должны размещаться экземпляры данного типа, а разработчик, использующий тип в своем коде, управлять этим не может.

В следующем коде (и на рис. 5.2) показано различие между ссылочными и значимыми типами:

```
// Ссылочный тип (поскольку 'class')
class SomeRef { public Int32 x; }

// Значимый тип (поскольку 'struct')
struct SomeVal { public Int32 x; }

static void ValueTypeDemo() {
    SomeRef r1 = new SomeRef(); // Размещается в куче
    SomeVal v1 = new SomeVal(); // Размещается в стеке
    r1.x = 5; // Разыменовывание указателя
    v1.x = 5; // Изменение в стеке
    Console.WriteLine(r1.x); // Отображается "5"
    Console.WriteLine(v1.x); // Также отображается "5"
    // В левой части рис. 5.2 показан результат
    // выполнения предыдущих строк

    SomeRef r2 = r1; // Копируется только ссылка (указатель)
    SomeVal v2 = v1; // Помещаем в стек и копируем члены
    r1.x = 8; // Изменяются r1.x и r2.x
    v1.x = 9; // Изменяется v1.x, но не v2.x
    Console.WriteLine(r1.x); // Отображается "8"
    Console.WriteLine(r2.x); // Отображается "8"
    Console.WriteLine(v1.x); // Отображается "9"
    Console.WriteLine(v2.x); // Отображается "5"
    // В правой части рис. 5.2 показан результат
    // выполнения ВСЕХ предыдущих строк
}
```

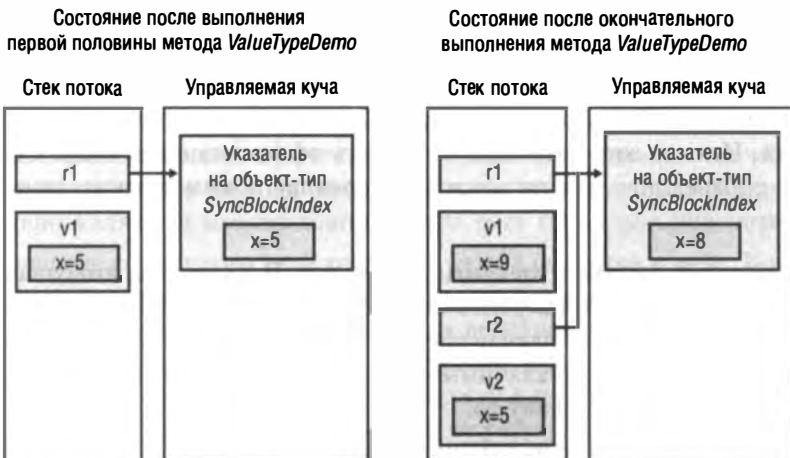


Рис. 5.2. Разница между размещением в памяти значимых и ссылочных типов

В этом примере тип `SomeVal` объявлен с ключевым словом `struct`, а не более популярным `class`. В C# типы, объявленные как `struct`, являются значимыми, а объявленные как `class`, — ссылочными. Разницы в поведении ссылочных и значимых типов практически не видно. Поэтому так важно представлять, к какому семейству относится тот или иной тип — к ссылочному или значимому: ведь это может существенно повлиять на эффективность кода.

В предыдущем примере есть следующая строка:

```
SomeVal v1 = new SomeVal(); // Размещается в стеке
```

Может показаться, что экземпляр `SomeVal` будет помещен в управляемую кучу. Однако поскольку компилятор C# «знает», что `SomeVal` является значимым типом, в сформированном им коде экземпляр `SomeVal` будет помещен в стек потока. C# также обеспечивает обнуление всех полей экземпляра значимого типа.

Ту же строку можно записать иначе:

```
SomeVal v1; // Размещается в стеке
```

Здесь тоже создается IL-код, который помещает экземпляр `SomeVal` в стек потока и обнуляет все его поля. Единственное отличие в том, что экземпляр, созданный оператором `new`, C# «считает» инициализированным. Поясню эту мысль на следующем примере:

```
// Две следующие строки компилируются, так как C# считает,  
// что поля в v1 инициализируются нулем  
SomeVal v1 = new SomeVal();  
Int32 a = v1.x;
```

```
// Следующие строки вызовут ошибку компиляции, поскольку C# не считает,  
// что поля в v1 инициализируются нулем  
SomeVal v1;  
Int32 a = v1.x;  
// error CS0170: Use of possibly unassigned field 'x'  
// (ошибка CS0170: Используется поле 'x', которому не присвоено значение)
```

Проектируя свой тип, проверьте, не использовать ли вместо ссылочного типа значимый. Иногда это позволяет повысить эффективность кода. Сказанное особенно справедливо для типа, удовлетворяющего *всем* перечисленным далее условиям.

- ❑ Тип ведет себя подобно примитивному типу. В частности, это означает, что тип достаточно простой и у него нет членов, способных изменить экземплярные поля типа, в этом случае говорят, что тип *неизменяемый* (immutable). На самом деле, многие значимые типы рекомендуется пометить спецификатором `readonly` (см. главу 7).
- ❑ Типу не нужен любой другой тип в качестве базового.
- ❑ Тип не будет иметь производных от него типов.

Размер экземпляров типа также нужно учитывать, потому что по умолчанию аргументы передаются по значению, при этом поля экземпляров значимого типа копируются, что отрицательно сказывается на производительности. Повторюсь: метод, возвращающий значимый тип, приводит к копированию полей экземпляра в память, выделенную вызывающим кодом в месте возврата из метода, что снижает эффективность работы программы. Поэтому в дополнение к перечисленным условиям следует объявлять тип как значимый, если любое из следующих условий верно.

- ❑ Размер экземпляров типа мал (примерно 16 байт или меньше).
- ❑ Размер экземпляров типа велик (более 16 байт), но экземпляры не передаются в качестве параметров метода или не являются возвращаемыми из метода значениями.

Основное достоинство значимых типов в том, что они не размещаются в управляемой куче. Конечно, в сравнении со ссылочными типами у значимых типов есть недостатки. Вот некоторые особенности, отличающие значимые и ссылочные типы.

- ❑ Объекты значимого типа существуют в двух формах (см. следующий раздел): *неупакованной* (unboxed) и *упакованной* (boxed). Ссылочные типы бывают только в упакованной форме.
- ❑ Значимые типы являются производными от `System.ValueType`. Этот тип имеет те же методы, что и `System.Object`. Однако `System.ValueType` переопределяет метод `Equals`, который возвращает `true`, если значения полей в обоих объектах совпадают. Кроме того, в `System.ValueType` переопределен метод `GetHashCode`, который создает значение хэш-кода с помощью алгоритма, учитывающего значения полей экземпляра объекта. Из-за проблем с производительностью в реализации по умолчанию, определяя собственные значимые типы значений, надо переопределить и написать свою реализацию методов `Equals` и `GetHashCode`. О методах `Equals` и `GetHashCode` рассказано в конце этой главы.
- ❑ Поскольку в объявлении нового значимого или ссылочного типа нельзя указывать значимый тип в качестве базового класса, создавать в значимом типе новые виртуальные методы нельзя. Методы не могут быть абстрактными и неявно являются изолированными (то есть их нельзя переопределить).
- ❑ Переменные ссылочного типа содержат адреса объектов в куче. Когда переменная ссылочного типа создается, ей по умолчанию присваивается `null`, то есть в этот момент она не указывает на действительный объект. Попытка задействовать переменную с таким значением приведет к генерации исключения `NullReferenceException`. В то же время в переменной значимого типа всегда содержится некое значение соответствующего типа, а при инициализации всем членам этого типа присваивается 0. Поскольку переменная значимого типа не является указателем, при обращении к значимому типу

исключение `NullReferenceException` возникнуть не может. CLR поддерживает понятие значимого типа особого вида, допускающего присваивание `null` (nullable types). Этот тип обсуждается в главе 19.

- Когда переменной значимого типа присваивается другая переменная значимого типа, выполняется копирование всех ее полей. Когда переменной ссылочного типа присваивается переменная ссылочного типа, копируется только ее адрес.
- Вследствие сказанного в предыдущем абзаце, несколько переменных ссылочного типа могут ссылаться на один объект в куче, благодаря чему, работая с одной переменной, можно изменить объект, на который ссылается другая переменная. В то же время каждая переменная значимого типа имеет собственную копию данных «объекта», поэтому операции с одной переменной значимого типа не влияют на другую переменную.
- Так как неупакованные значимые типы не размещаются в куче, отведенная для них память освобождается сразу при возвращении управления методом, в котором описан экземпляр этого типа. Это значит, что экземпляр значимого типа не получает уведомления (через метод `Finalize`), когда его память освобождается.

ПРИМЕЧАНИЕ

Действительно, было бы довольно странно включать в описание значимого типа метод `Finalize`, так как он вызывается только для упакованных экземпляров. Поэтому многие компиляторы (включая C#, C++/CLI и Visual Basic) не допускают в описании значимых типов методы `Finalize`. Правда, CLR допускает включение метода `Finalize` в описание значимого типа, однако при сборке мусора для упакованных экземпляров значимого типа этот метод не вызывается.

Как CLR управляет размещением полей для типа

Для повышения производительности CLR дано право устанавливать порядок размещения полей типа. Например, CLR может выстроить поля таким образом, что ссылки на объекты окажутся в одной группе, а поля данных и свойства — выровненные и упакованные — в другой. Однако при описании типа можно указать, сохранить ли порядок полей данного типа, определенный программистом, или разрешить CLR выполнить эту работу.

Для того чтобы сообщить CLR способ управления полями, укажите в описании класса или структуры атрибут `System.Runtime.InteropServices.StructLayoutAttribute`. Чтобы порядок полей устанавливался CLR, нужно передать конструктору атрибута параметр `LayoutKind.Auto`, чтобы сохранить установленный программистом порядок — параметр `LayoutKind.Sequential`, а параметр `LayoutKind.Explicit` позволяет разместить поля в памяти, явно задав смещения.

Если в описании типа не применен атрибут `StructLayoutAttribute`, порядок полей выберет компилятор.

Для ссылочных типов (классов) компилятор C# выбирает вариант `LayoutKind.Auto`, а для значимых типов (структур) — `LayoutKind.Sequential`. Очевидно, разработчики компилятора считают, что структуры обычно используются для взаимодействия с неуправляемым кодом, а значит, поля нужно расположить так, как определено разработчиком. Однако при создании значимого типа, не работающего совместно с неуправляемым кодом, скорее всего, поведение компилятора, предлагаемое по умолчанию, потребуется изменить, например:

```
using System; using System.Runtime.InteropServices;
```

```
// Для повышения производительности разрешим CLR
// установить порядок полей для этого типа
internal struct SomeValType {
    Byte b;
    Int16 x;
}
```

Атрибут `StructLayoutAttribute` также позволяет явно задать смещение для всех полей, передав в конструктор `LayoutKind.Explicit`. Затем можно применить атрибут `System.Runtime.InteropServices.FieldOffsetAttribute` ко всем полям путем передачи конструктору этого атрибута значения типа `Int32`, указывающего на смещение (в байтах) первого байта поля, начиная с начала экземпляра. Явное расположение обычно служит имитацией того, что в неуправляемом коде на C/C++ называлось *объединением* (*union*), потому что несколько полей могут начинаться с одного смещения в памяти, например:

```
using System;
using System.Runtime.InteropServices;

// Разработчик явно задает порядок полей в значимом типе
[StructLayout(LayoutKind.Explicit)]
internal struct SomeValType {
    [FieldOffset(0)]
    Byte b; // Поля b и x перекрываются
    [FieldOffset(0)]
    Int16 x; // в экземплярах этого класса
}
```

Стоит заметить, что считается недопустимым определять тип, в котором перекрываются ссылочный и значимый типы. Можно определить тип, в котором перекрываются несколько значимых типов, однако все перекрывающиеся байты должны быть доступны через открытые поля, чтобы обеспечить верификацию типа. Если какое-то поле одного значимого типа является закрытым и одновременно открытым в другом перекрывающем значимом типе, такой тип не поддается верификации.

Упаковка и распаковка значимых типов

Значимые типы «легче» ссылочных: для них не нужно выделять память в управляемой куче, их не затрагивает сборка мусора, к ним нельзя обратиться через указатель. Однако часто требуется получать ссылку на экземпляр значимого типа, например если вы хотите сохранить структуры `Point` в объекте типа `ArrayList` (определен в пространстве имен `System.Collections`). В коде это выглядит примерно следующим образом:

```
// Объявляем значимый тип
struct Point {
    public Int32 x, y;
}

public sealed class Program {
    public static void Main() {
        ArrayList a = new ArrayList();
        Point p;          // Выделяется память для Point (не в куче)
        for (Int32 i = 0; i < 10; i++) {
            p.x = p.y = i; // Инициализация членов в нашем значимом типе
            a.Add(p);       // Упаковка значимого типа и добавление
                           // ссылки в ArrayList
        }
        ...
    }
}
```

В каждой итерации цикла инициализируются поля значимого типа `Point`. Затем `Point` помещается в `ArrayList`. Задумаемся, что же помещается в `ArrayList`: структура `Point`, адрес структуры `Point` или что-то иное? За ответом обратимся к методу `Add` типа `ArrayList` и посмотрим описание его параметра. В данном случае прототип метода `Add` выглядит следующим образом:

```
public virtual Int32 Add(Object value);
```

Отсюда видно, что в качестве параметра `Add` нужен тип `Object`, то есть ссылка (или указатель) на объект в управляемой куче. Однако в примере я передаю переменную `p`, имеющую значимый тип `Point`. Чтобы код работал, нужно преобразовать значимый тип `Point` в объект из управляемой кучи и получить на него ссылку.

Для преобразования значимого типа в ссылочный служит *упаковка* (*boxing*). При упаковке экземпляра значимого типа происходит следующее.

1. В управляемой куче выделяется память. Ее объем определяется длиной значимого типа и двумя дополнительными членами — указателем на объект-тип и индексом `SyncBlockIndex`. Эти члены необходимы для всех объектов в управляемой куче.

2. Поля значимого типа копируются в память, только что выделенную в куче.
3. Возвращается адрес объекта. Этот адрес является ссылкой на объект, то есть значимый тип превращается в ссылочный.

Компилятор C# создает IL-код, необходимый для упаковки экземпляра значимого типа, автоматически, но вы должны понимать, что происходит «за кулисами» и помнить об опасности «распухания» кода и снижения производительности.

В предыдущем примере компилятор C# обнаружил, что методу, требующему ссылочный тип, я передаю как параметр значимый тип, и автоматически создал код для упаковки объекта. Вследствие этого поля экземпляра `p` значимого типа `Point` в период выполнения копируются во вновь созданный в куче объект `Point`. Полученный адрес упакованного объекта `Point` (теперь это ссылочный тип) передается методу `Add`. Объект `Point` остается в куче до очередной сборки мусора. Переменную `p` значимого типа `Point` можно повторно использовать или удалить из памяти, так как `ArrayList` ничего о ней не знает. Заметьте: время жизни упакованного значимого типа превышает время жизни неупакованного значимого типа.

ПРИМЕЧАНИЕ

Следует заметить, что в состав FCL входит новое множество обобщенных классов наборов, из-за чего устарели необобщенные классы коллекций. Так, вместо класса `System.Collections.ArrayList` следует использовать класс `System.Collections.Generic.List<T>`. Обобщенные классы коллекций во многих отношениях совершеннее своих необобщенных аналогов. В частности, API-интерфейс стал яснее и совершеннее, кроме того, повышена производительность классов коллекций. Но одно из самых ценных улучшений заключается в предоставляемой обобщенными классами коллекций возможности работать с коллекциями значимых типов, не прибегая к их упаковке/распаковке. Одна эта особенность позволяет значительно повысить производительность, так как радикально сокращается число создаваемых в управляемой куче объектов, что, в свою очередь, сокращает число проходов сборщика мусора в приложении. Более того, в результате обеспечивается безопасность типов на этапе компиляции, а код становится понятнее за счет сокращения числа приведений типов. Все эти вопросы обсуждаются в главе 12.

Познакомившись с упаковкой, перейдем к распаковке. Допустим, в другом месте кода нужно извлечь первый элемент массива `ArrayList`:

```
Point p = (Point) a[0];
```

Здесь ссылку (или указатель), содержащуюся в элементе с номером 0 массива `ArrayList`, вы пытаетесь поместить в переменную `p` значимого типа `Point`. Для этого все поля, содержащиеся в упакованном объекте `Point`, надо скопировать

в переменную `p` значимого типа, находящуюся в стеке потока. CLR выполняет эту процедуру в два этапа. Сначала извлекается адрес полей `Point` из упакованного объекта `Point`. Этот процесс называют *распаковкой* (unboxing). Затем значения полей копируются из кучи в экземпляр значимого типа, находящийся в стеке.

Распаковка *не* является точной противоположностью упаковки. Она гораздо менее ресурсозатратна, чем упаковка, и состоит только в получении указателя на исходный значимый тип (поля данных), содержащийся в объекте. В сущности, указатель ссылается на неупакованную часть упакованного экземпляра. И никакого копирования при распаковке (в отличие от упаковки) не требуется. Однако обычно вслед за распаковкой выполняется копирование полей, поэтому в сумме обе эти операции являются отражением операции упаковки.

Понятно, что упаковка и распаковка/копирование снижают производительность приложения (в плане как замедления, так и расходования дополнительной памяти), поэтому нужно знать, когда компилятор сам создает код для выполнения этих операций, и стараться свести их к минимуму.

При распаковке ссылочного типа происходит следующее.

1. Если переменная, содержащая ссылку на упакованный значимый тип, равна `null`, генерируется исключение `NullReferenceException`.
2. Если ссылка указывает на объект, не являющийся упакованным значением требуемого значимого типа, генерируется исключение `InvalidCastException`¹.

Иллюстрацией второго пункта может быть код, который *не* работает так, как хотелось бы:

```
public static void Main() {  
    Int32 x = 5;  
    Object o = x; // Упаковка x: o указывает на упакованный объект  
    Int16 y = (Int16) o; // Генерируется InvalidCastException  
}
```

Казалось бы, можно взять упакованный экземпляр `Int32`, на который указывает `o`, и привести к типу `Int16`. Однако когда выполняется распаковка объекта, должно быть сделано приведение к неупакованному типу (в нашем случае — к `Int32`). Вот как выглядит исправленный вариант:

```
public static void Main() {  
    Int32 x = 5;  
    Object o = x; // Упаковка x: o указывает на упакованный объект  
    Int16 y = (Int16)(Int32) o; // Распаковка, а затем приведение типа  
}
```

¹ CLR также позволяет распаковывать значимые типы в версию этого же типа, поддерживающую присвоение значений `null` (см. главу 19).

Как я уже отмечал, распаковка часто сопровождается копированием полей. Код на С# демонстрирует, что операции распаковки и копирования всегда работают совместно:

```
public static void Main() {  
    Point p;  
    p.x = p.y = 1;  
    Object o = p; // Упаковка p; o указывает на упакованный объект  
    p = (Point) o; // Распаковка o и копирование полей из экземпляра в стек  
}
```

В последней строке компилятор С# генерирует IL-команду для распаковки o (получение адреса полей в упакованном экземпляре) и еще одну IL-команду для копирования полей из кучи в переменную p, располагающуюся в стеке.

Теперь посмотрите на следующий пример:

```
public static void Main() {  
    Point p;  
    p.x = p.y = 1;  
    Object o = p; // Упаковка p; o указывает на упакованный экземпляр  
    // Изменение поля x структуры Point (присвоение числа 2). p = (Point) o;  
    // Распаковка o и копирование полей из экземпляра в переменную в стеке  
  
    p.x = 2; // Изменение состояния переменной в стеке  
  
    o = p; // Упаковка p; o ссылается на новый упакованный экземпляр  
}
```

Во второй части примера нужно лишь изменить поле x структуры Point с 1 на 2. Для этого выполняют распаковку, копирование полей, изменение поля (в стеке) и упаковку (создающую новый объект в управляемой куче). Надеюсь, вы понимаете, что все эти операции обязательно сказываются на производительности приложения.

В некоторых языках, например в С++/CLI, разрешается распаковать упакованный значимый тип, не копируя поля. После распаковки возвращается адрес неупакованной части упакованного объекта (дополнительные члены — указатель объект-тип объекта и SyncBlockIndex — игнорируются). Затем, используя полученный указатель, можно манипулировать полями неупакованного экземпляра (который находится в упакованном объекте в куче). Например, чтобы повысить производительность предыдущего кода, нужно переписать его на С++/CLI, где изменить значение поля x структуры Point внутри упакованного экземпляра Point. Это позволит избежать как выделения памяти для нового объекта, так и повторного копирования всех полей!

ВНИМАНИЕ

Если вы заботитесь о производительности своего приложения, вам необходимо знать, когда компилятор создает код, выполняющий эти операции. К сожалению, многие компиляторы создают код упаковки неявно, поэтому иногда сложно отследить, предусматривает ли созданный код упаковку. Если меня действительно волнует производительность приложения, я прибегаю к такому инструменту, как ILDasm.exe, позволяющему просматривать IL-код готовых методов на предмет наличия команд упаковки.

Рассмотрим еще несколько примеров, демонстрирующих упаковку и распаковку:

```
public static void Main() {
    Int32 v = 5; // Создаем неупакованную переменную значимого типа
    Object o = v; // o указывает на упакованное Int32, содержащее 5
    v = 123; // Изменяем неупакованное значение на 123
    Console.WriteLine(v + ", " + (Int32) o); // Отображается "123, 5"
}
```

Сколько в этом коде операций упаковки и распаковки? Вы не поверите — целых три! Разобраться в том, что здесь происходит, нам поможет IL-код метода Main. Чтобы быстрее найти отдельные операции, я снабдил распечатку комментариями.

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Размер кода 45 (0x2e)
    .maxstack 3
    .locals init (int32 V_0,
                 object V_1)
    // Загружаем 5 в v
    IL_0000: ldc.i4.5
    IL_0001: stloc.0

    // Упакуем v и сохраняем указатель в o
    IL_0002: ldloc.0
    IL_0003: box          [mscorlib]System.Int32
    IL_0008: stloc.1

    // Загружаем 123 в v
    IL_0009: ldc.i4.s 123
    IL_000b: stloc.0

    // Упакуем v и сохраняем в стеке указатель для Concat
    IL_000c: ldloc.0
    IL_000d: box          [mscorlib]System.Int32
```

```
// Загружаем строку в стек для Concat
IL_0012: ldstr ". "

// Распакуем o: берем указатель в поле Int32 в стеке
IL_0017: ldloc.1
IL_0018: unbox.any [mscorlib]System.Int32

// Упакуем Int32 и сохраняем в стеке указатель для Concat
IL_001d: box [mscorlib]System.Int32

// Вызываем Concat
IL_0022: call string [mscorlib]System.String::Concat(object,
                                                    object,
                                                    object)

// Строку, возвращенную из Concat, передаем в WriteLine
IL_0027: call
void [mscorlib]System.Console::WriteLine(string)

// Метод Main возвращает управление, и приложение завершается
IL_002c: ret
} // Конец метода App::Main
```

Вначале создается экземпляр *v* неупакованного значимого типа *Int32*, которому присваивается число 5. Затем создается переменная *o* ссылочного типа *Object*, которая указывает на *v*. Однако поскольку ссылочные типы всегда должны указывать на объекты в куче, C# генерирует соответствующий IL-код для упаковки *v* и заносит адрес упакованной «копии» *v* в *o*. Теперь величина 123 помещается в неупакованный значимый тип *v*, но это не влияет на упакованное значение типа *Int32*, которое остается равным 5.

Дальше вызывается метод *WriteLine*, в который нужно передать объект *String*, но такого объекта нет. Вместо строкового объекта мы имеем неупакованный экземпляр значимого типа *Int32* (*v*), объект *String* (ссылочного типа) и ссылку на упакованный экземпляр значимого типа *Int32* (*o*), который приводится к неупакованному типу *Int32*. Эти элементы нужно как-то объединить, чтобы получился объект *String*.

Чтобы создать *String*, компилятор C# формирует код, в котором вызывается статический метод *Concat* объекта *String*. Есть несколько перегруженных версий этого метода, различающихся лишь количеством параметров. Поскольку строка формируется путем конкатенации трех элементов, компилятор выбирает следующую версию метода *Concat*:

```
public static String Concat(Object arg0, Object arg1, Object arg2);
```

В качестве первого параметра, *arg0*, передается *v*. Но *v* — это неупакованное значение, а *arg0* — это значение *Object*, поэтому экземпляр *v* нужно упаковать, а его адрес передать в качестве *arg0*. Параметром *arg1* является строка ". " в виде

ссылки на объект `String`. И наконец, чтобы передать параметр `arg2`, `o` (ссылка на `Object`) приводится к типу `Int32`. Для этого нужна распаковка (но без копирования), при которой извлекается адрес неупакованного экземпляра `Int32` внутри упакованного экземпляра `Int32`. Этот неупакованный экземпляр `Int32` надо опять упаковать, а его адрес в памяти передать в качестве параметра `arg2` методу `Concat`.

Метод `Concat` вызывает методы `ToString` для каждого указанного объекта и выполняет конкатенацию строковых представлений этих объектов. Возвращаемый из `Concat` объект `String` передается затем методу `WriteLine`, который отображает окончательный результат.

Полученный IL-код станет эффективнее, если обращение к `WriteLine` переписать:

```
Console.WriteLine(v + ", " + o); // Отображается "123, 5"
```

Этот вариант строки отличается от предыдущего только отсутствием для переменной `o` операции приведения типа `Int32`. Этот код выполняется быстрее, так как `o` уже является ссылочным типом `Object` и его адрес можно сразу передать методу `Concat`. Отказавшись от приведения типа, я избавился от двух операций: распаковки и упаковки. В этом легко убедиться, если заново собрать приложение и посмотреть на сгенерированный IL-код:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Размер кода 35 (0x23)
    .maxstack 3
    .locals init (int32 V_0,
                 object V_1)

    // Загружаем 5 в v
    IL_0000: ldc.i4.5
    IL_0001: stloc.0

    // Упакуем v и сохраняем указатель в o
    IL_0002: ldloc.0
    IL_0003: box [mscorlib]System.Int32
    IL_0008: stloc.1

    // Загружаем 123 в v
    IL_0009: ldc.i4.s 123
    IL_000b: stloc.0

    // Упакуем v и сохраняем в стеке указатель для Concat
    IL_000c: ldloc.0
    IL_000d: box [mscorlib]System.Int32
```

```
// Загружаем строку в стек для Concat
IL_0012: ldstr ".", "

// Загружаем в стек адрес упакованного Int32 для Concat
IL_0017: ldloc.1

// Вызываем Concat
IL_0018: call string [mscorlib]System.String::Concat(object,
                                                    object,
                                                    object)

// Строку, возвращенную из Concat, передаем в WriteLine
IL_001d: call void [mscorlib]System.Console::WriteLine(string)

// Main возвращает управление, чем завершается работа приложения
IL_0022: ret
} // Конец метода App::Main
```

Беглое сравнение двух версий IL-кода метода Main показывает, что вариант без приведения типа Int32 на 10 байт меньше, чем вариант с приведением типа. Дополнительные операции распаковки/упаковки, безусловно, приводят к разрастанию кода. Если мы пойдем дальше, то увидим, что эти операции потребуют выделения памяти в управляемой куче для дополнительного объекта, которую в будущем должен освободить сборщик мусора. Конечно, обе версии приводят к одному результату и разница в скорости незаметна, однако лишние операции упаковки, выполняемые многократно (например, в цикле), могут заметно повлиять на производительность приложения и расходование памяти.

Предыдущий код можно улучшить, изменив вызов метода WriteLine:

```
Console.WriteLine(v.ToString() + ".", " + o); // Отображается "123, 5"
```

Для неупакованного значимого типа `v` теперь вызывается метод `ToString`, возвращающий `String`. Строковые объекты являются ссылочными типами и могут легко передаваться в метод `Concat` без упаковки.

Вот еще один пример, демонстрирующий упаковку и распаковку:

```
public static void Main() {
    Int32 v = 5;           // Создаем неупакованную переменную значимого типа
    Object o = v;          // o указывает на упакованную версию v

    v = 123;               // Изменяет неупакованный значимый тип на 123
    Console.WriteLine(v);  // Отображает "123"

    v = (Int32) o;          // Распаковывает и копирует o в v
    Console.WriteLine(v);  // Отображает "5"
}
```

Сколько операций упаковки вы насчитали в этом коде? Правильно — одну. Дело в том, что в классе `System.Console` описан метод `WriteLine`, принимающий в качестве параметра тип `Int32`:

```
public static void WriteLine(Int32 value);
```

В показанных ранее вызовах `WriteLine` переменная `v`, имеющая неупакованный значимый тип `Int32`, передается по значению. Возможно, где-то у себя `WriteLine` упакует это значение `Int32`, но тут уж ничего не поделаешь. Главное — мы сделали то, что от нас зависело: убрали упаковку из своего кода.

Пристально взглянув на FCL, можно заметить, что многие перегруженные методы используют в качестве параметров значимые типы. Так, тип `System.Console` предлагает несколько перегруженных вариантов метода `WriteLine`:

```
public static void WriteLine(Boolean);  
public static void WriteLine(Char);  
public static void WriteLine(Char[]);  
public static void WriteLine(Int32);  
public static void WriteLine(UInt32);  
public static void WriteLine(Int64);  
public static void WriteLine(UInt64);  
public static void WriteLine(Single);  
public static void WriteLine(Double);  
public static void WriteLine(Decimal);  
public static void WriteLine(Object);  
public static void WriteLine(String);
```

Аналогичный набор перегруженных версий есть у метода `Write` типа `System.Console`, у метода `Write` типа `System.IO.BinaryWriter`, у методов `Write` и `WriteLine` типа `System.IO.TextWriter`, у метода `AddValue` типа `System.Runtime.Serialization.SerializationInfo`, у методов `Append` и `Insert` типа `System.Text.StringBuilder` и т. д. Большинство этих методов имеет перегруженные версии только затем, чтобы уменьшить количество операций упаковки для наиболее часто используемых значимых типов.

Если определить собственный значимый тип, у этих FCL-классов не будет соответствующей перегруженной версии для этого типа. Более того, для ряда значимых типов, уже существующих в FCL, нет перегруженных версий указанных методов. Если вызывать метод, у которого нет перегруженной версии для передаваемого значимого типа, результат в конечном итоге будет один — вызов перегруженного метода, принимающего `Object`. Передача значимого типа как `Object` приведет к упаковке, что отрицательно скажется на производительности. Определяя собственный класс, можно задать в нем обобщенные методы (возможно, содержащие параметры типа, которые являются значимыми типами). Обобщения позволяют определить метод, принимающий любой значимый тип, не требуя при этом упаковки (см. главу 12).

И последнее, что касается упаковки: если вы знаете, что ваш код будет периодически заставлять компилятор упаковывать какой-то значимый тип, можно уменьшить объем и повысить быстродействие своего кода, выполнив упаковку этого типа вручную. Взгляните на следующий пример.

```
using System;
```

```
public sealed class Program {  
    public static void Main() {  
        Int32 v = 5; // Создаем переменную упакованного значимого типа  
  
        #if INEFFICIENT  
            // При компиляции следующей строки v упакуется  
            // три раза, расходуя и время, и память  
            Console.WriteLine("{0}. {1}. {2}". v. v. v);  
        #else  
            // Следующие строки дают тот же результат,  
            // но выполняются намного быстрее и расходуют меньше памяти  
            Object o = v; // Упакуем вручную v (только единожды)  
  
            // При компиляции следующей строки код для упаковки не создается  
            Console.WriteLine("{0}. {1}. {2}". o. o. o);  
        #endif  
    }  
}
```

Если компилировать этот код с определенным символом `INEFFICIENT`, компилятор создаст код, трижды выполняющий упаковку `v` и выделяющий память в куче для трех объектов! Это особенно расточительно, так как каждый объект будет содержать одно значение — 5. Если компилировать код без определения символа `INEFFICIENT`, значение `v` будет упаковано только раз и только один объект будет размещен в куче. Затем при обращении к `Console.WriteLine` трижды передается ссылка на один и тот же упакованный объект. Второй вариант выполняется *намного* быстрее и расходует меньше памяти в куче.

В этих примерах довольно легко определить, где нужно упаковать экземпляры значимого типа. Простое правило: если нужна ссылка на экземпляр значимого типа, этот экземпляр должен быть упакован. Обычно упаковка выполняется, когда надо передать значимый тип методу, требующему ссылочный тип. Однако могут быть и другие ситуации, когда требуется упаковать экземпляр значимого типа.

Помните, мы говорили, что неупакованные значимые типы «легче», чем ссылочные, поскольку:

- ❑ память в управляемой куче им не выделяется;
- ❑ у них нет дополнительных членов, присущих каждому объекту в куче: указателя на объект-тип и индекса `SyncBlockIndex`.

Поскольку неупакованные значимые типы не имеют индекса `SyncBlockIndex`, то не может быть и нескольких потоков, синхронизирующих свой доступ к экземпляру через методы типа `System.Threading.Monitor` (или инструкция `lock` языка C#).

Раз неупакованные значимые типы не имеют указателя на объект-тип, нельзя вызвать унаследованные или переопределенные в типе реализации виртуальных методов (таких как `Equals`, `GetHashCode` или `ToString`). Причина в том, что CLR может вызывать эти методы не виртуально, а `System.ValueType` переопределяет их и ожидает, что значение в аргументе `this` ссылается на неупакованный экземпляр значимого типа. Вспомните, что значимый тип всегда неявно изолирован и поэтому не может выступать базовым классом другого типа. Это также значит, что CLR может не виртуально вызывать виртуальные методы значимого типа.

Вместе с тем вызов не виртуального унаследованного метода (такого, как `GetType` или `MemberwiseClone`) требует упаковки значимого типа, так как эти методы определены в `System.Object`, поэтому методы ожидают, что в аргументе `this` передается указатель на объект в куче.

Кроме того, приведение неупакованного экземпляра значимого типа к одному из интерфейсов этого типа требует, чтобы экземпляр был упакован, так как интерфейсы всегда являются ссылочными типами. (Об интерфейсах см. главу 13.) Сказанное иллюстрирует следующий код:

```
using System;
```

```
internal struct Point : IComparable {
    private Int32 m_x, m_y;

    // Конструктор, просто инициализирующий поля
    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }

    // Переопределяем метод ToString, унаследованный от System.ValueType
    public override String ToString() {
        // Возвращаем Point как строку
        return String.Format("{0}. {1}", m_x, m_y);
    }

    // Безопасная в отношении типов реализация метода CompareTo
    public Int32 CompareTo(Point other) {
        // Используем теорему Пифагора для определения точки,
        // наиболее удаленной от начала координат (0, 0)
        return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
            - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
    }
}
```

```
// Реализация метода CompareTo интерфейса IComparable
public Int32 CompareTo(Object o) {
    if (GetType() != o.GetType()) {
        throw new ArgumentException("o is not a Point");
    }

    // Вызов безопасного в отношении типов метода CompareTo
    return CompareTo((Point) o);
}

}

public static class Program {
    public static void Main() {
        // Создаем в стеке два экземпляра Point
        Point p1 = new Point(10, 10);
        Point p2 = new Point(20, 20);

        // p1 НЕ пакуется для вызова ToString (виртуальный метод)
        Console.WriteLine(p1.ToString()); // "(10, 10)"

        // p1 ПАКУЕТСЯ для вызова GetType (невиртуальный метод)
        Console.WriteLine(p1.GetType()); // "Point"

        // p1 НЕ пакуется для вызова CompareTo
        // p2 НЕ пакуется, потому что вызван CompareTo(Point)
        Console.WriteLine(p1.CompareTo(p2)); // "-1"

        // p1 НЕ пакуется, а ссылка размещается в c
        IComparable c = p1;
        Console.WriteLine(c.GetType()); // "Point"

        // p1 НЕ пакуется для вызова CompareTo
        // Поскольку в CompareTo не передается переменная Point,
        // вызывается CompareTo(Object), которому нужна ссылка
        // на упакованный Point
        // c НЕ пакуется, потому что уже ссылается на упакованный Point
        Console.WriteLine(p1.CompareTo(c)); // "0"

        // c НЕ пакуется, потому что уже ссылается на упакованный Point
        // p2 ПАКУЕТСЯ, потому что вызывается CompareTo(Object)
        Console.WriteLine(c.CompareTo(p2)); // "-1"

        // c пакуется, а поля копируются в p2
        p2 = (Point) c;
        // Убеждаемся, что поля скопированы в p2
        Console.WriteLine(p2.ToString()); // "(10, 10)"
    }
}
```


В этом примере демонстрируется сразу несколько сценариев поведения кода, связанного с упаковкой/распаковкой.

- ❑ **Вызов ToString.** При вызове ToString упаковка `p1` не требуется. Сначала я бы решил, что тип `p1` должен быть упакован, так как ToString — метод, унаследованный от базового типа, `System.ValueType`. Обычно, чтобы вызвать наследуемый метод, нужен указатель на объект-тип, а поскольку `p1` является неупакованным значимым типом, то нет ссылки на объект-тип `Point`. Однако компилятор C# видит, что метод ToString переопределен в `Point`, и создает код, который напрямую (невиртуально) вызывает ToString. Компилятор знает, что полиморфизм здесь невозможен, коль скоро `Point` является значимым типом, а значимые типы не могут применяться для другого типа в качестве базового и по-другому реализовывать виртуальный метод.
- ❑ **Вызов GetType.** При вызове неvirtуального метода GetType упаковка `p1` необходима, поскольку тип `Point` не реализует GetType, а наследует его от `System.Object`. Поэтому для вызова GetType нужен указатель на объект-тип `Point`, который можно получить только путем упаковки `p1`.
- ❑ **Первый вызов CompareTo.** При первом вызове CompareTo упаковка `p1` не нужна, так как `Point` реализует метод CompareTo, и компилятор может просто обратиться к нему напрямую. Заметьте: в CompareTo передается переменная `p2` типа `Point`, поэтому компилятор вызывает перегруженную версию CompareTo, которая принимает параметр типа `Point`. Это означает, что `p2` передается в CompareTo по значению, и никакой упаковки не требуется.
- ❑ **Приведение типа к IComparable.** Когда выполняется приведение типа `p1` к переменной интерфейсного типа (`c`), упаковка `p1` необходима, так как интерфейсы по определению имеют ссылочный тип. Поэтому выполняется упаковка `p1`, а указатель на этот упакованный объект сохраняется в переменной `c`. Следующий вызов GetType подтверждает, что `c` действительно ссылается на упакованный объект `Point` в куче.
- ❑ **Второй вызов CompareTo.** При втором вызове CompareTo упаковка `p1` не производится, потому что `Point` реализует метод CompareTo, и компилятор может вызывать его напрямую. Заметьте, что в CompareTo передается переменная `c` интерфейса `IComparable`, поэтому компилятор вызывает перегруженную версию CompareTo, которая принимает параметр типа `Object`. Это означает, что передаваемый параметр должен являться указателем, ссылающимся на объект в куче. К счастью, `c` уже ссылается на упакованный объект `Point`, по этой причине адрес памяти из `c` может передаваться в CompareTo и никакой дополнительной упаковки не требуется.
- ❑ **Третий вызов CompareTo.** При третьем вызове CompareTo переменная `c` уже ссылается на упакованный объект `Point` в куче. Поскольку переменная `c` сама по себе имеет интерфейсный тип `IComparable`, можно вызывать только метод CompareTo интерфейса, а ему требуется параметр `Object`. Это означает, что передаваемый аргумент должен быть указателем, ссылающимся на объект

в куче. Поэтому выполняется упаковка `p2` и указатель на этот упакованный объект передается в `CompareTo`.

- ❑ **Приведение типа к `Point`.** Когда выполняется приведение `s` к типу `Point`, объект в куче, на который указывает `s`, распаковывается, и его поля копируются из кучи в `p2`, экземпляр типа `Point`, находящийся в стеке.

Понимаю, что вся эта информация о ссылочных и значимых типах, упаковке и распаковке поначалу кажется устрашающей. И все же любой разработчик, стремящийся к долгосрочному успеху на ниве .NET Framework, должен хорошо усвоить эти понятия — только так можно научиться быстро и легко создавать эффективные приложения.

Изменение полей в упакованных значимых типах посредством интерфейсов

Посмотрим, насколько хорошо вы усвоили тему значимых типов, упаковки и распаковки. Взгляните на следующий пример: можете ли вы сказать, что будет выведено на консоль в следующем случае.

```
using System;
```

```
// Point – значимый тип
internal struct Point {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }

    public void Change(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }

    public override String ToString() {
        return String.Format("{0}, {1}", m_x, m_y);
    }
}

public sealed class Program {
    public static void Main() {
        Point p = new Point(1, 1);

        Console.WriteLine(p);
    }
}
```

```

    p.Change(2, 2);
    Console.WriteLine(p);

    Object o = p;
    Console.WriteLine(o);

    ((Point) o).Change(3, 3);
    Console.WriteLine(o);
}
}

```

Все просто: Main создает в стеке экземпляр `p` типа `Point` и устанавливает его поля `m_x` и `m_y` равными 1. Затем `p` упаковывается до первого обращения к методу `WriteLine`, который вызывает `ToString` для упакованного типа `Point`, в результате выводится, как и ожидалось, (1, 1). Затем `p` применяется для вызова метода `Change`, который изменяет значения полей `m_x` и `m_y` объекта `p` в стеке на 2. При втором обращении к `WriteLine` выводится, как и предполагалось, (2, 2).

Далее `p` упаковывается в третий раз — `o` ссылается на упакованный объект типа `Point`. При третьем обращении к `WriteLine` снова выводится (2, 2), что опять вполне ожидаемо. И наконец, я обращаюсь к методу `Change` для изменения полей в упакованном объекте типа `Point`. Между тем `Object` (тип переменной `o`) ничего не «знает» о методе `Change`, так что сначала нужно привести `o` к `Point`. При таком приведении типа `o` распаковывается, и поля упакованного объекта типа `Point` копируются во временный объект типа `Point` в стеке потока. Поля `m_x` и `m_y` этого временного объекта устанавливаются равными 3, но это обращение к `Change` не влияет на упакованный объект `Point`. При обращении к `WriteLine` снова выводится (2, 2). Для многих разработчиков это оказывается *неожиданностью*.

Некоторые языки, например C++/CLI, позволяют изменять поля в упакованном значимом типе, но только не C#. Однако и C# можно обмануть, применив интерфейс. Вот модифицированная версия предыдущего кода:

```

using System;

// Интерфейс, определяющий метод Change
internal interface IChangeBoxedPoint {
    void Change(Int32 x, Int32 y);
}

// Point - значимый тип
internal struct Point : IChangeBoxedPoint {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }
}

```

```
public void Change(Int32 x, Int32 y) {
    m_x = x;
    m_y = y;
}

public override String ToString() {
    return String.Format("{0}, {1}", m_x, m_y);
}

}

public sealed class Program {
    public static void Main() {
        Point p = new Point(1, 1);

        Console.WriteLine(p);
        p.Change(2, 2);
        Console.WriteLine(p);

        Object o = p;
        Console.WriteLine(o);

        ((Point) o).Change(3, 3);
        Console.WriteLine(o);

        // p упаковывается, упакованный объект изменяется и освобождается
        ((IChangeBoxedPoint) p).Change(4, 4);
        Console.WriteLine(p);

        // Упакованный объект изменяется и выводится
        ((IChangeBoxedPoint) o).Change(5, 5);
        Console.WriteLine(o);
    }
}
```

Этот код практически совпадает с предыдущим. Основное отличие заключается в том, что метод `Change` определяется интерфейсом `IChangeBoxedPoint` и теперь тип `Point` реализует этот интерфейс. Внутри `Main` первые четыре вызова `WriteLine` те же самые и выводят те же результаты (что и следовало ожидать). Однако в конце `Main` я добавил пару примеров.

В первом примере `p` — неупакованный объект типа `Point` — приводится к типу `IChangeBoxedPoint`. Такое приведение типа вызывает упаковку `p`. Метод `Change` вызывается для упакованного значения, и его поля `m_x` и `m_y` становятся равными 4, но при возврате из `Change` упакованный объект немедленно становится доступным для утилизации сборщиком мусора. Так что при пятом обращении к `WriteLine` на экран выводится (2, 2), что для многих неожиданно.

В последнем примере упакованный тип `Point`, на который ссылается `o`, приводится к типу `IChangeBoxedPoint`. Упаковка здесь не производится, поскольку

тип `o` уже упакован. Затем вызывается метод `Change`, который изменяет поля `m_x` и `m_y` упакованного типа `Point`. Интерфейсный метод `Change` позволил мне изменить поля упакованного объекта типа `Point`! Теперь при обращении к `Writeline` выводится (5, 5).

Назначение этих примеров — продемонстрировать, как метод интерфейса может изменить поля в упакованном значимом типе. В C# сделать это без интерфейсов нельзя.

ВНИМАНИЕ

Ранее в этой главе я отмечал, что значимые типы должны быть неизменяемыми, то есть в значимых типах нельзя определять члены, которые изменяют какие-либо поля экземпляра. Фактически я рекомендовал, чтобы такие поля в значимых типах помечались спецификатором `readonly`, чтобы компилятор сообщил об ошибке, если вы вдруг случайно напишите метод, пытающийся модифицировать такое поле. Предыдущий пример как нельзя лучше иллюстрирует это. Показанное в примере неожиданное поведение программы проявляется при попытке вызвать методы, изменяющие поля экземпляра значимого типа. Если после создания значимого типа не вызывать методы, изменяющие его состояние, не возникнет недоразумений при копировании поля в процессе упаковки и распаковки. Если значимый тип неизменяемый, результатом будет простое многократное копирование одного и того же состояния, поэтому не возникнет непонимания наблюдаемого поведения.

Некоторые главы этой книги я показал разработчикам. Познакомившись с примерами программ (например, из этого раздела), они сказали, что разочарованы в значимых типах. Должен сказать, что эти незначительные нюансы значимых типов стоили мне многодневной отладки, поэтому я и описываю их в этой книге. Надеюсь, вы не забудете об этих нюансах, тогда они не застигнут вас врасплох. Не бойтесь значимых типов — они полезны и занимают свою нишу. Просто не забывайте, что ссылочные и значимые типы ведут себя по-разному в зависимости от того, как применяются. Возьмите предыдущий код и объявите `Point` как `class`, а не `struct` — увидите, что все получится. И наконец, радостная новость заключается в том, что значимые типы, содержащиеся в библиотеке FCL — `Byte`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Decimal`, `BigInteger`, `Complex` и все перечисляемые типы, — ведут себя неизменно и не преподносят никаких сюрпризов.

Равенство и тождество объектов

Часто разработчикам приходится создавать код сравнения объектов. В частности, это необходимо, когда объекты размещаются в коллекциях и требуется писать код для сортировки, поиска и сравнения отдельных элементов в коллекции. В этом разделе рассказывается о равенстве и тождестве объектов, а также о том, как определять тип, который правильно реализует равенство объектов.

У типа `System.Object` есть виртуальный метод `Equals`, который возвращает `true` для двух «равных» объектов. Вот реализация метода `Equals` для `Object`:

```
public class Object {  
    public virtual Boolean Equals(Object obj) {  
        // Если обе ссылки указывают на один и тот же объект.  
        // значит, эти объекты равны  
        if (this == obj)  
            return true;  
        // Предполагаем, что объекты не равны  
        return false;  
    }  
}
```

Как видите, в этом методе реализован простейший подход: сравниваются две ссылки, переданные в аргументах `this` и `obj`, и если они указывают на один объект, возвращается `true`, в противном случае возвращается `false`. Это кажется разумным, так как `Equals` «понимает», что объект равен самому себе. Однако если аргументы ссылаются на разные объекты, `Equals` сложнее определить, содержат ли объекты одинаковые значения, поэтому возвращается `false`. Иначе говоря, оказывается, что стандартная реализация метода `Equals` типа `Object` реализует проверку на тождество, а не на равенство.

Как видите, оказалось, что приведенная здесь стандартная реализация не годится: нужно переопределить метод `Equals` и написать собственную реализацию. Вот как нужно создавать свою версию метода `Equals`.

1. Если аргумент `obj` равен `null`, вернуть `false`, так как ясно, что текущий объект, указанный в `this`, не равен `null` при вызове нестатического метода `Equals`.
2. Если аргументы `obj` и `this` ссылаются на объекты одного типа, вернуть `true`. Этот шаг поможет повысить производительность в случае сравнения объектов с многочисленными полями.
3. Если аргументы `obj` и `this` ссылаются на объекты разного типа, вернуть `false`. Не нужно быть семи пядей во лбу, чтобы понять, что объект `String` не равен объекту `FileStream`.
4. Сравнить все определенные в типе экземплярные поля объектов `obj` и `this`. Если хотя бы одна пара полей не равна, вернуть `false`.
5. Вызвать метод `Equals` базового класса, чтобы сравнить определенные в нем поля. Если метод `Equals` базового класса вернул `false`, тоже вернуть `false`, в противном случае вернуть `true`.

Учитывая это, компания Microsoft должна была бы реализовать метод `Equals` типа `Object` примерно так:

```
public class Object {  
    public virtual Boolean Equals(Object obj) {  
        // Сравнимый объект не может быть равным null  
        if (obj == null) return false;
```

продолжение ➤

```
// Объекты разных типов не могут быть равны
if (this.GetType() != obj.GetType()) return false;

// Если типы объектов совпадают, возвращаем true при условии,
// что все их поля попарно равны
// Так как в System.Object не определены поля,
// следует считать, что поля равны
return true;
}
}
```

Однако, поскольку в Microsoft метод Equals реализован иначе, правила собственной реализации Equals намного сложнее, чем кажется. Если ваш тип переопределяет Equals, переопределенная версия метода должна вызывать реализацию Equals в базовом классе, если только не планируется вызывать реализацию в типе Object. Это означает еще и то, что поскольку тип может переопределять метод Equals типа Object, этот метод больше не может использоваться для проверки на тождественность. Для исправления ситуации в Object предусмотрен статический метод ReferenceEquals со следующим прототипом:

```
public class Object {
    public static Boolean ReferenceEquals(Object objA, Object objB) {
        return (objA == objB);
    }
}
```

Для проверки на тождественность нужно всегда вызывать ReferenceEquals (то есть проверять на предмет того, относятся ли две ссылки к одному объекту). Не нужно использовать оператор == языка C# (если только перед этим оба операнда не приводятся к типу Object), так как тип одного из операндов может перегружать этот оператор, в результате чего его семантика перестает соответствовать понятию «тождественность».

Как видите, в области равенства и тождественности в .NET Framework дела обстоят довольно сложно. Кстати, в System.ValueType (базовом классе всех значимых типов) метод Equals типа Object переопределен и корректно реализован для проверки на равенство (но не тождественность). Логика работы переопределенного метода такова:

1. Если аргумент obj равен null, вернуть false.
2. Если аргументы obj и this ссылаются на объекты разного типа, вернуть false.
3. Сравнить попарно все определенные в типе экземплярные поля объектов obj и this. Если хотя бы одна пара полей не равна, вернуть false.
4. Вернуть true. Метод Equals типа ValueType не вызывает одноименный метод типа Object.

Для выполнения шага 3 в методе Equals типа ValueType используется отражение (см. главу 23). Так как отражение в CLR работает медленно, при создании

собственного значимого типа нужно переопределить `Equals` и создать свою реализацию, чтобы повысить производительность сравнения значений на предмет равенства экземпляров созданного типа. И, конечно же, не стоит вызывать из этой реализации метод `Equals` базового класса.

Определяя собственный тип и приняв решение переопределить `Equals`, следите за тем, чтобы поддерживались свойства, присущие равенствам:

- ❑ **Рефлексивность:** `x.Equals(x)` должно возвращать `true`.
- ❑ **Симметричность:** `x.Equals(y)` и `y.Equals(x)` должны возвращать одно и то же значение.
- ❑ **Транзитивность:** если `x.Equals(y)` возвращает `true` и `y.Equals(z)` возвращает `true`, то `x.Equals(z)` также должно возвращать `true`.
- ❑ **Постоянство:** если в двух сравниваемых значениях не произошло изменений, результат сравнения тоже не должен измениться.

Отступление от этих правил при создании своего варианта `Equals` грозит непредсказуемым поведением приложения.

При переопределении метода `Equals` может потребоваться выполнить несколько дополнительных операций.

- ❑ **Реализовать в типе метод `Equals` интерфейса `System.IEquatable<T>`.** Этот обобщенный интерфейс позволяет определить безопасный в отношении типов метод `Equals`. Обычно `Equals` реализуют так, что, принимая параметр типа `Object`, код метода вызывает безопасный в отношении типов метод `Equals`.
- ❑ **Перегрузить методы операторов `==` и `!=`.** Обычно код реализации этих операторных методов вызывает безопасный в отношении типов метод `Equals`.

Более того, если предполагается сравнивать экземпляры собственного типа для целей сортировки, рекомендуется также реализовать метод `CompareTo` типа `System.IComparable` и безопасный в отношении типов метод `CompareTo` типа `System.IComparable<T>`. Реализовав эти методы, можно реализовать метод `Equals` так, чтобы он вызывал `CompareTo` типа `System.IComparable<T>` и возвращал `true`, если `CompareTo` возвратит 0. После реализации методов `CompareTo` также часто требуется перегрузить методы различных операторов сравнения (`<`, `<=`, `>`, `>=`) и реализовать код этих методов так, чтобы он вызывал безопасный в отношении типов метод `CompareTo`.

Хэш-коды объектов

Разработчики FCL решили, что при формировании коллекции хэш-таблиц полезно применять любые экземпляры любых типов. С этой целью в `System.Object` включен виртуальный метод `GetHashCode`, позволяющий получить для любого объекта целочисленный (`Int32`) хэш-код.

Если вы определяете тип и переопределяете метод `Equals`, вы должны переопределить и метод `GetHashCode`. Если при определении типа переопределить только один из этих методов, компилятор C# выдаст предупреждение. Например, при компиляции представленного далее кода появится предупреждение: `warning CS0659: 'Program' overrides Object.Equals(Object o) but does not override Object.GetHashCode() ('Program' переопределяет Object.Equals(Object o), но не переопределяет Object.GetHashCode())`.

```
public sealed class Program {
    public override Boolean Equals(Object obj) { ... }
}
```

Причина, по которой в типе должны быть описаны оба метода — `Equals` и `GetHashCode`, — состоит в том, что реализация типов `System.Collections.Hashtable`, `System.Collections.Generic.Dictionary` и любых других коллекций требует, чтобы два равных объекта имели одинаковые значения хэш-кодов. Поэтому, переопределяя `Equals`, нужно переопределить `GetHashCode` и гарантировать тем самым соответствие алгоритма, применяемого для вычисления равенства, алгоритму, используемому для вычисления хэш-кода объекта.

По сути, когда вы добавляете пару «ключ-значение» в коллекцию, первым вычисляется хэш-код для ключа. Этот хэш-код указывает, в каком «сегменте» будет храниться пара «ключ-значение». Когда коллекции нужно найти некий ключ, она вычисляет для него хэш-код. Хэш-код определяет «сегмент» поиска имеющегося в таблице ключа, равного заданному. Применение этого алгоритма хранения и поиска ключей означает, что если вы измените хранящийся в коллекции ключ объекта, коллекция больше не сможет найти этот объект. Если вы намерены изменить ключ объекта в хэш-таблице, то сначала удалите имеющуюся пару «ключ-значение», модифицируйте ключ, а затем добавьте в хэш-таблицу новую пару «ключ-значение».

В описании метода `GetHashCode` нет особых хитростей. Однако для некоторых типов данных и их распределения в памяти бывает непросто подобрать алгоритм хэширования, который выдавал бы хорошо распределенный диапазон значений. Вот простой алгоритм, неплохо подходящий для объектов `Point`:

```
internal sealed class Point {
    private Int32 m_x, m_y;
    public override Int32 GetHashCode() {
        return m_x ^ m_y; // Исключающее ИЛИ для m_x и m_y
    }
    ...
}
```

Выбирая алгоритм вычисления хэш-кодов для экземпляров своего типа, старайтесь следовать определенным правилам:

- Используйте алгоритм, который дает случайное распределение, повышающее производительность хэш-таблицы.

- ❑ Алгоритм может вызывать метод `GetHashCode` базового типа и использовать возвращаемое им значение, однако в общем случае лучше отказаться от вызова встроенного метода `GetHashCode` для типа `Object` или `ValueType`, так как эти реализации не годятся в силу низкой производительности их алгоритмов хэширования.
- ❑ В алгоритме должно использоваться как минимум одно экземплярное поле.
- ❑ Поля, используемые в алгоритме, в идеале не должны изменяться, то есть их нужно инициализировать при создании объекта и не изменять в течение всей его жизни.
- ❑ Алгоритм должен быть максимально быстрым.
- ❑ Объекты с одинаковым значением должны возвращать одинаковые коды, например два объекта `String`, содержащие одинаковый текст, должны возвращать одно значение хэш-кода.

Реализация `GetHashCode` в `System.Object` ничего «не знает» о производных типах и их полях. Поэтому этот метод возвращает число, однозначно идентифицирующее объект в пределах домена приложений; при этом гарантируется, что это число не изменится на протяжении всей жизни объекта. Однако когда объект прекратит свое существование после сборки мусора, это число может стать хэш-кодом для другого объекта.

ПРИМЕЧАНИЕ

Если тип переопределен в методе `GetHashCode` типа `Object`, то вы не можете его больше вызывать для получения уникального идентификатора объекта. Если нужно получить уникальный (в рамках домена приложений) идентификатор объекта, вызовите соответствующий метод FCL. В пространстве имен `System.Runtime.CompilerServices` найдите внутри класса `RuntimeHelpers` статический метод `GetHashCode`, который ссылается на `Object` в качестве аргумента. Этот метод возвращает уникальный идентификатор объекта, даже если тип объекта переопределен в методе `GetHashCode` типа `Object`. Название этого метода обусловлено историческими причинами, но было бы лучше, если бы компания Microsoft назвала его как-нибудь по-другому, например `GetUniqueObjectID`.

В реализации `GetHashCode` для `System.ValueType` используется механизм отражения (который отличается медленной работой) и некоторые поля экземпляра типа обрабатываются операцией исключающего «или» (XOR). Такой простой способ может быть полезен для некоторых значимых типов, но я бы посоветовал вам написать свою реализацию `GetHashCode`, поскольку в своем методе вы будете уверены, к тому же ваша версия окажется быстрее базовой.

ВНИМАНИЕ

Если вы взялись за реализацию собственной коллекции хэш-таблиц или пишете код, в котором будет вызываться метод `GetHashCode`, никогда не храните значения хэш-кодов. Они подвержены изменениям в силу своей природы. Например, при переходе к следующей версии типа алгоритм вычисления хэш-кода объекта может просто измениться.

Я знаю компанию, которая проигнорировала это предупреждение. Посетители ее веб-сайта создавали новые учетные записи, выбирая имя пользователя и пароль. Строка (`String`) пароля передавалась методу `GetHashCode`, а полученный хэш-код сохранялся в базе данных. В дальнейшем при входе на веб-сайт посетители указывали свой пароль, который снова обрабатывался методом `GetHashCode`, и полученный хэш-код сравнивался с сохраненным в базе данных. При совпадении пользователю предоставлялся доступ. К несчастью, после обновления версии CLR метод `GetHashCode` типа `String` изменился и стал возвращать другой хэш-код. Результат оказался плачевным — все пользователи потеряли доступ к веб-сайту!

Примитивный тип данных `dynamic`

Язык C# обеспечивает безопасность типов данных. Это означает, что все выражения разрешаются в экземпляр типа, и компилятор генерирует только тот код, который старается представить операции, правомерные для данного типа данных.

Выгода от использования языка, обеспечивающего безопасность типов данных, по сравнению с языками, не обеспечивающими таковую, заключается в том, что еще на этапе компиляции обнаруживается множество ошибок программирования, что помогает программисту корректировать код перед его выполнением. К тому же при помощи подобных языков программирования можно получать более быстрые приложения, потому что они разрешают больше допущений еще на этапе компиляции и затем переводят эти допущения в язык IL или метаданные.

Однако возможны неприятные ситуации, возникающие из-за того, что программа должна выполняться на основе информации, недоступной до ее выполнения. Если вы используете языки программирования, обеспечивающие безопасность данных (например, C#) для взаимодействия с этой информацией, синтаксис становится громоздким, особенно в случае, если вы работаете с множеством строк, в результате производительность приложения падает. Если вы пишете приложение на «чистом» языке C#, неприятная ситуация может подстерегать вас только во время работы с информацией, определяемой на этапе выполнения, когда вы используете отражения (см. главу 23). Однако многие разработчики используют также C# для связи с компонентами, не реализован-

ными на С#. Некоторые из этих компонентов могут быть написаны на динамических языках, например Python или Ruby, или быть СОМ-объектами, которые поддерживают интерфейс IDispatch (возможно, реализованный на С или С++), или объектами модели DOM (Document Object Model), реализованными при помощи разных языков и технологий.

Связь с DOM-объектами отчасти полезна для построения Silverlight-приложений. Для того чтобы облегчить разработку при помощи отражений или коммуникаций с другими компонентами, компилятор С+ предлагает помечать типы как *динамические* (dynamic). Вы также можете записывать результаты вычисления выражений в переменную и указывать ее тип как динамический. Затем динамическое выражение (переменная) может быть использовано для вызовов членов класса, например поля, свойства/индексатора, метода, делегата, или унарных/бинарных операторов. Когда ваш код вызывает член класса при помощи динамического выражения (переменной), компилятор создает специальный IL-код, который описывает желаемую операцию. Этот код представляется нам *полезной нагрузкой* (payload).

Во время выполнения программы такой код определяет существующую операцию для выполнения на основе действительного типа объекта, на который ссылается динамическое выражение (переменная).

Следующий код поясняет эту мысль:

```
Private static class DynamicDemo {
    public static void Main() {
        for (Int32 demo = 0; demo < 2; demo++) {
            dynamic arg = (demo == 0) ? (dynamic) 5 : (dynamic) "A";
            dynamic result = Plus(arg);
            M(result);
        }
    }

    private static dynamic Plus(dynamic arg) { return arg + arg; }

    private static void M(Int32 n) { Console.WriteLine("M(Int32): " + n); }
    private static void M(String s) { Console.WriteLine("M(String): " + s); }
}
```

После выполнения метода Main получается следующий результат:

```
M(Int32): 10
M(String): AA
```

Для того чтобы понять, что произошло, давайте обратимся к методу Plus. У этого метода параметр помечен как динамический, и внутри метода этот аргумент используется в двух операндах для бинарного оператора +. Из-за того что переменная arg является динамической, компилятор С# составляет полезную нагрузку, которая проверяет действительный тип переменной arg во время выполнения и определяет действия оператора +.

Во время первого вызова метода `Plus` значение его аргумента равно 5 (тип `Int32`), поэтому `Plus` возвращает значение 10 (тоже тип `Int32`). Результат присваивается переменной `result` (динамического типа). Затем вызывается метод `M` с переменной `result`. Для вызова метода `M` компилятор создает код полезной нагрузки, который на этапе выполнения будет проверять действительный тип значения переменной, переданной в метод `M`. Когда `result` содержит тип `Int32`, перегрузка метода `M` делает параметр вызываемого метода типом `Int32`.

Во время второго вызова метода `Plus` значение его аргумента равно `A` (тип `String`), и `Plus` возвращает строку `AA` (результат конкатенации `A` с собой), которая в виде переменной записывается в `result`. В это время код полезной нагрузки определяет, что действительный тип, переданный в `M`, является строковым, и перегружает `M` со строковым параметром.

Когда тип поля, параметр метода, возвращаемый тип метода или локальная переменная обозначаются как динамические, компилятор конвертирует этот тип в тип `System.Object` и применяет экземпляр `System.Runtime.CompilerServices.DynamicAttribute` к полю, параметру или возвращаемому типу в метаданных. Если локальная переменная определена как динамическая, то тип переменной также будет типом `Object`, но атрибут `DynamicAttribute` неприменим к локальным переменным из-за того, что они используются только внутри метода. Из-за того, что типы `dynamic` и `Object` одинаковы, вы не сможете создавать методы с сигнатурами, отличающимися только типами `dynamic` и `Object`.

Можно использовать тип `dynamic` для определения обобщенного (generic) типа аргументов в обобщенном классе (ссылочного типа), структуре (значимый тип), интерфейсе, делегате или методе. Когда вы это делаете, компилятор конвертирует тип `dynamic` в `Object` и применяет `DynamicAttribute` к различным частям метаданных, где это необходимо. Обратите внимание, что обобщенный код, который вы используете, уже скомпилирован в соответствии с типом `Object`, и динамическая отправка не осуществляется, поскольку компилятор не производит код полезной нагрузки в обобщенном коде.

Любое выражение может быть явно приведено к динамическому, поскольку все выражения дают в результате тип, унаследованный от `Object`¹. Это нормально, компилятор не позволит вам написать код с неявным приведением выражения от типа `Object` к другому типу, вы должны использовать явное приведение типов. Однако компилятор разрешит выполнить явное приведение типа `dynamic` к другому типу.

```
Object o1 = 123;      // OK: Неявное приведение Int32 к Object (упаковка)
Int32 n1 = o;         // Ошибка: Нет неявного приведения Object к Int32
Int32 n2 = (Int32) o; // OK: Явное приведение Object к Int32 (распаковка)
dynamic d1 = 123;     // OK: Неявное приведение Int32 к dynamic (упаковка)
Int32 n3 = d;         // OK: Неявное приведение dynamic к Int32 (распаковка)
```

Пока компилятор позволяет пренебрегать явным приведением динамического типа к другому типу данных, среда CLR на этапе выполнения проверяет

¹ И как обычно, значимый тип будет упакован.

правильность приведения типов с целью обеспечения безопасности типов данных. Если тип объекта несовместим с приведением, CLR вбрасывает исключение `InvalidCastException`.

Обратите внимание на следующий код:

```
dynamic d = 123;  
var result = M(d); // 'var result' одинаков с 'dynamic result'
```

Здесь компилятор позволяет коду компилироваться, потому что на этапе компиляции он не знает, какой из методов `M` будет вызван. Следовательно, он также не знает, какой тип будет возвращен методом `M`. Компилятор предполагает, что переменная `result` имеет динамический тип. Вы можете убедиться в этом, когда наведете указатель мыши на переменную `var` в редакторе Visual Studio — во всплывающем IntelliSense-окне вы увидите следующее.

dynamic: Represents an object whose operations will be resolved at runtime.

Если метод `M`, вызванный на этапе выполнения, возвращает `void`, то исключение не вбрасывается, вместо этого переменной `result` присваивается значение `null`.

ВНИМАНИЕ

Не путайте типы `dynamic` и `var`. Объявление локальной переменной как `var` является синтаксическим указанием компилятору подставлять специальные данные из соответствующего выражения. Ключевое слово `var` может использоваться только для объявления локальных переменных внутри метода, тогда как ключевое слово `dynamic` может указываться с локальными переменными, полями и аргументами. Вы не можете привести выражение к типу `var`, но вы можете привести его к типу `dynamic`. Вы должны явно инициализировать переменную, объявленную как `var`, тогда как переменную, объявленную как `dynamic`, инициализировать нельзя. Больше подробно о типе `var` рассказывается в главе 9.

При конвертации типа `dynamic` в другой статический тип результатом будет, очевидно, тоже статический тип. При создании типа с одним и более аргументом в его конструкторе строится объект того типа, который вы создаете:

```
dynamic d = 123;  
var x = (Int32) d;           // Конвертация: 'var x' одинаково с 'Int32 x'  
var dt = new DateTime(d);    // Создание: 'var dt' одинаково с 'DateTime dt'
```

Если динамическое выражение определено как коллекция в инструкции `foreach` или при помощи ключевого слова `using`, то компилятор генерирует код, который попытается привести выражение к необобщенному интерфейсу `System.IEnumerable` или интерфейсу `System.IDisposable`. Если приведение типов состоится, то выражение будет использоваться, и код выполнится. В противном случае будет выброшено исключение `Microsoft.CSharp.RuntimeBinder.RuntimeBinderException`.

ВНИМАНИЕ

Динамическое выражение реально имеет тот же тип, что и `System.Object`. Компилятор принимает операции с выражением как допустимые и не генерирует ни предупреждений, ни ошибок. Однако исключения могут быть выброшены на этапе выполнения программы, если программа попытается выполнить недопустимую операцию. К тому же Visual Studio не может предложить какую-либо IntelliSense-поддержку для написания кода с динамическими выражениями. Вы не можете определить метод расширения для `dynamic` (об этом рассказывается в главе 8), хотя можете его определить для `Object`. И вы можете использовать лямбда-выражение или анонимный метод (они оба обсуждаются в главе 17) в качестве аргумента при вызове динамического метода, пока компилятор ничего не предполагает о его применении.

Рассмотрим пример кода на C# с использованием COM-объекта `IDispatch` для создания тетради Microsoft Office Excel и размещения строки в ячейке A1.

```
using Microsoft.Office.Interop.Excel;
...
public static void Main() {
    Application excel = new Application();
    excel.Visible = true;
    excel.Workbooks.Add(Type.Missing);
    ((Range)excel.Cells[1, 1]).Value = "Text in cell A1";
    // Помещаем эту строку в ячейку A1.
}
```

Здесь не используется динамический тип, и значение, возвращаемое `excel.Cells[1, 1]`, имеет тип `Object`, который должен быть приведен к типу `Range` перед тем, как будет доступно его свойство `Value`. Однако во время генерации выполняемой сборки для COM-объекта любое использование типа `VARIANT` в COM-методе будет конвертироваться в динамический тип, этот механизм называется *динамофикацией* (dynamification). Следовательно, после того как вызов `excel.Cells[1, 1]` становится динамическим, вы не можете явно привести его к типу `Range`, прежде чем его свойство `Value` будет доступно.

Динамофикация значительно упрощает код, взаимодействующий с COM-объектами. Рассмотрим пример с более простым кодом.

```
using Microsoft.Office.Interop.Excel;
...
public static void Main() {
    Application excel = new Application();
    excel.Visible = true;
    excel.Workbooks.Add(Type.Missing);
    excel.Cells[1, 1].Value = "Text in cell A1";
    // Помещаем эту строку в ячейку A1
}
```

Код, представленный далее, показывает, как использовать отражение для вызова метода с аргументом типа String ("ff") на строку ("Jeffrey Richter") и поместить результат с типом Int32 в локальную переменную result.

```
Object target = "Jeffrey Richter";
Object arg = "ff";

// Находим метод, который согласует желаемые типы аргумента
Type[] argTypes = new Type[] { arg.GetType() };
MethodInfo method = target.GetType().GetMethod("Contains", argTypes);

// Вызываем метод с желаемым аргументом
Object[] arguments = new Object[] { arg };
Boolean result = Convert.ToBoolean(method.Invoke(target, arguments));
```

Если использовать динамический тип из языка C#, этот код можно значительно улучшить синтаксически.

```
dynamic target = "Jeffrey Richter";
dynamic arg = "ff";
Boolean result = target.Contains(arg);
```

Ранее я уже говорил о том, что компилятор C# на этапе выполнения программы генерирует код полезной нагрузки, основываясь на действительных типах объекта. Этот код полезной нагрузки использует класс, известный как *компоновщик* (runtime binder). Различные языки программирования инкапсулируют в свои компоновщики правила языка. Код для компоновщика C# находится в сборке Microsoft.CSharp.dll, и вы должны сослаться на эту сборку при компоновке проекта с использованием ключевого слова dynamic. Эта сборка ссылается на файл ответа, определенный по умолчанию, CSC.rsp. Коду из этой сборки известно, как представлять сложение в случае, когда оператор + применяется к двум объектам типа Int32 и как представлять конкатенацию применительно к двум объектам типа String.

Во время выполнения сборка Microsoft.CSharp.dll должна загрузиться в домен приложений, что снизит производительность приложения и повысит расход памяти. Кроме того, сборка Microsoft.SSharp.dll загружает библиотеки System.dll и System.Core.dll. А если вы используете динамический тип для связи с COM-объектами, загружается и библиотека System.Dynamic.dll. И когда будет выполнен код полезной нагрузки, генерирующий динамический код во время выполнения, этот код окажется в сборке, названной *сборкой анонимно хостируемых динамических методов* (Anonymously Hosted Dynamic Methods Assembly). Назначение этого кода заключается в повышении производительности динамических ссылок в сценариях с частично вызываемыми сайтами при помощи динамических аргументов, имеющих одинаковый тип на этапе выполнения.

В соответствии со всеми издержками, связанными с особенностями встроенных динамических вычислений в C#, вы должны осознанно решить, что именно

вы желаете добиться от динамического кода: превосходной производительности приложения при загрузке всех этих сборок или оптимального расходования памяти. Если вы используете динамический код только в паре мест вашего программного кода, разумнее придерживаться старого подхода: либо вызывать методы отражений (для управляемых объектов), либо «вручную» приводить типы (для COM-объектов).

Во время выполнения компоновщик C# разрешает динамические операции в соответствии с типом объекта. Сначала компоновщик проверяет, реализован ли тип в интерфейсе `IDynamicMetaObjectProvider`. И если объект там реализован, вызывается метод `GetMetaObject`, который возвращает порожденный тип `DynamicMetaObject`. Этот тип может обработать все члены, методы и операторы, связанные с объектом. Интерфейс `IDynamicMetaObjectProvider` и основной класс `DynamicMetaObject` определены в пространстве имен `System.Dynamic` и находятся в сборке `System.Core.dll`.

Динамические языки, такие как Python и Ruby, поддерживают свои типы при помощи порожденного типа `DynamicMetaObject`, таким образом, они могут быть доступны для взаимодействия из других языков (например, C#). Аналогичным образом компоновщик C# при связи с COM-компонентами будет использовать порожденный тип `DynamicMetaObject`, умеющий взаимодействовать с COM-компонентами. Порожденный тип `DynamicMetaObject` определен в сборке `System.Dynamic.dll`.

Если тип объекта, используемый в динамическом выражении, не реализует интерфейс `IDynamicMetaObjectProvider`, тогда компилятор C# воспринимает его как обычный объект типа языка C# и все связанные с ним действия осуществляет через отражение.

Глава 6. Основные сведения о членах и типах

В главах 4 и 5 были рассмотрены типы и операции, применимые ко всем экземплярам любого типа. Кроме того, объяснялось, почему все типы делятся на две категории — ссылочные и значимые. В этой и последующих главах я показываю, как проектировать типы, используя различные члены, которые можно определить в типах. В главах с 7 по 11 эти члены рассматриваются подробнее.

Члены типа

В типе можно определить следующие члены.

- ❑ **Константа** — идентификатор, определяющий некую постоянную величину. Эти идентификаторы обычно используют, чтобы сделать код более читабельным, а также для удобства сопровождения и поддержки. Константы всегда связаны с типом, а не с экземпляром типа. Константы всегда статичны. Подробнее о константах см. главу 7.
- ❑ **Поле** представляет собой неизменяемое или изменяемое значение. Поле может быть статическим — тогда оно является частью типа. Поле может быть экземплярным (нестатическим) — тогда оно является частью объекта. Я настоятельно рекомендую делать поля закрытыми, чтобы внешний код не мог нарушить состояние типа или объекта. Подробнее о полях см. главу 7.
- ❑ **Конструктор экземпляров** — метод, служащий для инициализации полей экземпляра при его создании. Подробнее о конструкторах экземпляров см. главу 8.
- ❑ **Конструктор типа** — метод, используемый для инициализации статических полей типа. Подробнее о конструкторах типа см. главу 8.
- ❑ **Метод** представляет собой функцию, выполняющую операции, которые изменяют или запрашивают состояние типа (статический метод) или объекта (экземплярный метод). Методы обычно осуществляют чтение и запись полей типов или объектов. Подробнее о методах см. главу 8.
- ❑ **Перегруженный оператор** определяет, что нужно проделать с объектом при применении к нему оператора. Перегрузка операторов не упоминается общезыковой спецификации CLS, поскольку не все языки программирования ее поддерживают. Подробнее о перегруженных операторах см. главу 8.

- ❑ **Оператор преобразования** — метод, задающий порядок явного или неявного преобразования объекта из одного типа в другой. Операторы преобразования не входят в спецификацию CLS по той же причине, что и перегруженные операторы. Подробнее об операторах преобразования см. главу 8.
- ❑ **Свойство** представляет собой механизм, позволяющий применить простой синтаксис (напоминающий обращение к полям) для установки или получения части логического состояния типа или объекта, не нарушая это состояние. Свойства чаще всего бывают непараметризованными. Параметризованные свойства обычно используются в классах коллекций. Подробнее о свойствах см. главу 10.
- ❑ **Событие**. Статическое событие служит механизмом, позволяющим типу посылать уведомление статическому или экземпляльному методу. Экземплярное (нестатическое) событие служит механизмом, позволяющим объекту посылать уведомление статическому или экземпляльному методу. События обычно инициируются в ответ на изменение состояния типа или объекта, порождающего событие. Событие состоит из двух методов, позволяющих статическим или экземплярным методам регистрировать и отменять регистрацию (подписку) на событие. Помимо этих двух методов, в событиях обычно используется поле-делегат для управления набором зарегистрированных методов. Подробнее о событиях см. главу 11.
- ❑ **Тип** позволяет определять другие вложенные в него типы. Обычно этот подход применяется для разбиения большого, сложного типа на небольшие блоки с целью упростить его реализацию.

Итак, цель данной главы состоит не в подробном описании различных членов, а в изложении основных положений и объяснении, что общего между этими членами.

Независимо от используемого языка программирования, компилятор должен обработать исходный код и создать метаданные и IL-код для всех членов типа. Формат метаданных един и не зависит от выбранного языка программирования — именно поэтому CLR называют *общезыковой* исполняющей средой. Метаданные — это стандартная информация, которую предоставляют и используют все языки, позволяя коду на одном языке программирования без проблем обращаться к коду на совершенно другом языке.

Стандартный формат метаданных также используется средой CLR для определения порядка поведения констант, полей, конструкторов, методов, свойств и событий во время выполнения. Короче говоря, метаданные — это ключ ко всей платформе разработки Microsoft .NET Framework; они обеспечивают интеграцию языков, типов и объектов.

В следующем примере на C# показано определение типа со всеми возможными членами. Этот код успешно компилируется (не без предупреждений), но пользы от него немного — всего лишь демонстрация, как компилятор транс-

лирует такой тип и его члены в метаданные. Еще раз напомним, что каждый из членов в отдельности детально рассмотрен в следующих главах.

```
using System;
```

```
public sealed class SomeType {                                     // 1

    // Вложенный класс
    private class SomeNestedType { }                             // 2

    // Константа, неизменяемое и статическое изменяемое поле
    private const Int32 SomeConstant = 1;                         // 3
    private readonly Int32 SomeReadOnlyField = 2;                 // 4
    private static Int32 SomeReadWriteField = 3;                   // 5

    // Конструктор типа
    static SomeType() { }                                         // 6

    // Конструкторы экземпляров
    public SomeType() { }                                         // 7
    public SomeType(Int32 x) { }                                  // 8

    // Экземплярный и статический методы
    private String InstanceMethod() { return null; }              // 9
    public static void Main() { }                                  // 10

    // Непараметризованное экземплярное свойство
    public Int32 SomeProp {                                       // 11
        get { return 0; }                                         // 12
        set { }                                                    // 13
    }

    // Параметризованное экземплярное свойство
    public Int32 this[String s] {                                  // 14
        get { return 0; }                                         // 15
        set { }                                                    // 16
    }

    // Экземплярное событие
    public event EventHandler SomeEvent;                           // 17
}
```

После компиляции типа можно просмотреть метаданные с помощью утилиты `ILDasm.exe` (рис. 6.1).

Заметьте, что компилятор генерирует метаданные для всех членов типа. На самом деле, для некоторых членов, например для события (член 17), компилятор создает дополнительные члены (поле и два метода) и метаданные. На данном этапе не требуется точно понимать, что изображено на рисунке, но по

мере чтения следующих глав я рекомендую возвращаться к этому примеру и смотреть, как задается тот или иной член и как это влияет на метаданные, генерируемые компилятором.

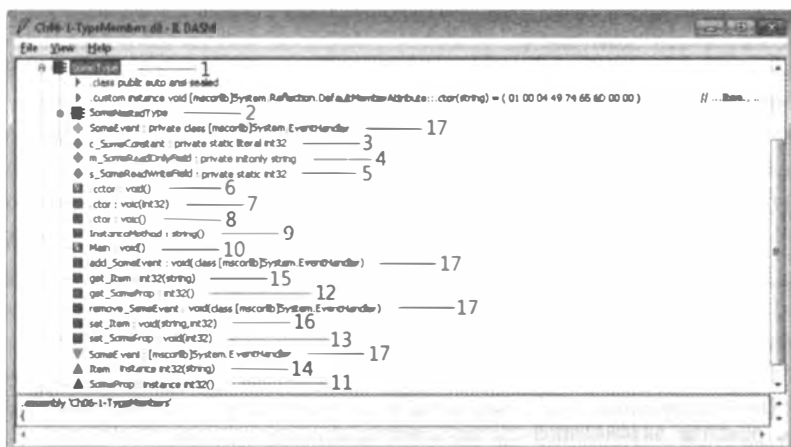


Рис. 6.1. Метаданные, полученные с помощью утилиты ILDasm.exe на основе кода примера

Видимость типа

При определении типа с видимостью в рамках файла, а не другого типа его можно сделать *открытым* (public) или *внутренним* (internal). Открытый тип доступен любому коду любой сборки. Внутренний тип доступен только из сборки, где он определен. По умолчанию компилятор C# делает тип внутренним (с более ограниченной видимостью). Вот несколько примеров.

using System;

```
// Это открытый тип; он доступен из любой сборки
public class ThisIsAPublicType { ... }
```

```
// Это внутренний тип; он доступен только из собственной сборки
internal class ThisIsAnInternalType { ... }
```

```
// Это внутренний тип, так как модификатор доступа не указан явно
class ThisIsAlsoAnInternalType { ... }
```

Дружественные сборки

Представьте себе следующую ситуацию: в компании есть группа *А*, определяющая набор полезных типов в одной сборке, и группа *Б*, использующая эти типы.

По разным причинам, таким как индивидуальные графики работы, географическая разобщенность, различные источники финансирования или структуры подотчетности, эти группы не могут разместить все свои типы в единой сборке; вместо этого в каждой группе создается собственный файл сборки.

Чтобы сборка группы *Б* могла использовать типы группы *А*, группа *А* должна определить все нужные второй группе типы как открытые. Однако это означает, что эти типы будут доступны абсолютно всем сборкам. В результате разработчики другой компании смогут написать код, использующий общедоступные типы, а это нежелательно. Вполне возможно, в полезных типах существуют определенные предположения, которым должна следовать группа *Б* при написании кода, использующего типы группы *А*. То есть нам необходим способ, позволяющий группе *А* определять свои типы как внутренние, но в то же время разрешающий группе *Б* получать доступ к этим типам. Для таких ситуаций в CLR и C# предусмотрен механизм *дружественных сборок* (friend assemblies).

В процессе создания сборки можно указать другие сборки, которые она будет считать «друзьями», — для этого служит атрибут `InternalsVisibleTo`, определенный в пространстве имен `System.Runtime.CompilerServices`. У атрибута есть строковый параметр, определяющий имя дружественной сборки, и открытый ключ (передаваемая атрибуту строка не должна содержать информацию о версии, региональных стандартах или архитектуре процессора). Заметьте, что дружественные сборки получают доступ *ко всем* внутренним типам сборки, а также к внутренним членам этих типов. Приведем пример сборки, которая объявляет своими друзьями две другие сборки со строгими именами `Wintellect` и `Microsoft`:

```
using System;
using System.Runtime.CompilerServices; // Для атрибута InternalsVisibleTo

// Внутренние типы этой сборки доступны из кода двух следующих сборок
// (независимо от версии или региональных стандартов)
[assembly: InternalsVisibleTo("Wintellect, PublicKey=12345678...90abcdef")]
[assembly: InternalsVisibleTo("Microsoft, PublicKey=b77a5c56...1934e089")]
```

```
internal sealed class SomeInternalType { ... } internal sealed class
AnotherInternalType { ... }
```

Обращаться из дружественной сборки просто к внутренним типам представленной здесь сборки. Например, дружественная сборка `Wintellect` с открытым ключом `!12345678...90abcdef` может обратиться к внутреннему типу `SomeInternalType` представленной сборки следующим образом:

```
using System;

internal sealed class Foo {
    private static Object SomeMethod() {
```

продолжение ➤

```
// Эта сборка Wintellect получает доступ к внутреннему типу
// другой сборки, как если бы он был открытым
SomeInternalType sit = new SomeInternalType();
return sit;
}
}
```

Поскольку внутренние члены принадлежащих сборке типов становятся доступными для дружественных сборок, следует очень осторожно подходить к определению уровня доступа, предоставляемого членам своего типа, и объявлению дружественных сборок. Заметьте, что при компиляции дружественной (то есть не содержащей атрибута `InternalsVisibleTo`) сборки компилятору C# требуется задавать параметр `/out: файл`. Он нужен компилятору, чтобы узнать имя компилируемой сборки и определить, должна ли результирующая сборка быть дружественной. Можно подумать, что компилятор C# в состоянии самостоятельно выяснить это имя, так как он обычно самостоятельно определяет имя выходного файла; однако компилятор «не знает» имя выходного файла, пока не завершится компиляция. Поэтому требование указывать этот параметр позволяет значительно повысить производительность компиляции.

Аналогично, при компиляции модуля (в противоположность сборке) с параметром `/t:module`, который должен стать частью дружественной сборки, необходимо также использовать параметр `/moduleassemblyname:строка` компилятора C#. Последний параметр говорит компилятору, к какой сборке будет относиться модуль, чтобы тот разрешил коду этого модуля обращаться к внутренним типам другой сборки.

ВНИМАНИЕ

Дружественные сборки следует использовать только в сборках, поставляемых в одно время или, лучше всего, вместе. Взаимозависимость дружественных сборок настолько высока, что разнесение поставки по времени практически наверняка вызовет проблемы совместимости. Если сборки все же приходится поставлять в разное время, следует решить проблему созданием открытых классов, доступных из любой сборки, но ограничить доступ посредством проверок `LinkDemand`, запрашивая разрешение `StrongNameIdentityPermission`.

Доступ к членам

При определении члена типа (в том числе вложенного) можно указать модификатор доступа к члену. Модификаторы определяют, на какие члены можно ссылаться из кода. В CLR имеется собственный набор возможных модификаторов доступа, но в каждом языке программирования существуют свои синтаксис и термины. Например, термин `Assembly` в CLR указывает, что член доступен изнутри сборки, тогда как в C# для этого используется ключевое слово `internal`.

В табл. 6.1 представлено шесть модификаторов доступа, определяющих уровень ограничения — от максимального (Private) до минимального (Public).

Таблица 6.1. Модификаторы доступа к членам

CLR	C#	Описание
Private (закрытый)	private	Доступен только методам в определяющем типе и вложенных в него типах
Family (родовой)	protected	Доступен только методам в определяющем типе (и вложенных в него типах) или одном из его производных типов независимо от сборки
Family and Assembly (родовой и сборочный)	(не поддерживается)	Доступен только методам в определяющем типе (и вложенных в него типах) и производных типах в определяющей сборке
Assembly (сборочный)	internal	Доступен только методам в определяющей сборке
Assembly or Family (сборочный или родовой)	protected internal	Доступен только методам вложенного типа, производного типа (независимо от сборки) и любым методам определяющей сборки
Public (открытый)	public	Доступен всем методам во всех сборках

Разумеется, доступ к члену можно получить, только если он определен в видимом типе. Например, если в сборке A определен внутренний тип, имеющий открытый метод, то код сборки B не сможет вызвать открытый метод, поскольку внутренний тип сборки A недоступен из B.

В процессе компиляции кода компилятор языка проверяет корректность обращения кода к типам и членам. Обнаружив некорректную ссылку на какие-либо типы или члены, компилятор информирует об ошибке. Помимо этого, во время выполнения JIT-компилятор тоже проверяет корректность обращения к полям и методам при компиляции IL-кода в процессорные команды. Например, обнаружив код, неправильно пытающийся обратиться к закрытому полю или методу, JIT-компилятор генерирует исключение `FieldAccessException` или `MethodAccessException` соответственно.

Верификация IL-кода гарантирует правильность обработки модификаторов доступа к членам в период выполнения, даже если компилятор языка проигнорировал проверку доступа. Другая, более вероятная возможность заключается в компиляции кода, обращающегося к открытому члену другого типа (другой сборки); если в период выполнения загрузится другая версия сборки, где модификатор доступа открытого члена заменен *защищенным* (protected) или *закрытым* (private), верификация обеспечит корректное управление доступом.

Если не указать явно модификатор доступа, компилятор C# обычно (но не всегда) выберет по умолчанию закрытый — наиболее строгий из всех. CLR требует, чтобы все члены интерфейсного типа были открытыми. Поэтому компилятор C# запрещает программисту явно указывать модификаторы доступа к членам интерфейса, просто делая все члены открытыми.

ПРИМЕЧАНИЕ

Подробнее о правилах применения в C# модификаторов доступа к типам и членам, а также о том, какие модификаторы C# выбирает по умолчанию в зависимости от контекста объявления, рассказывается в разделе «Declared Accessibility» спецификации языка C#.

Более того, как видно из таблицы, в CLR есть модификатор доступа *родовой и сборочный*. Но разработчики C# сочли этот атрибут лишним и не включили в язык C#.

Если в производном типе переопределяется член базового типа, компилятор C# требует, чтобы у членов базового и производного типов были одинаковые модификаторы доступа. То есть если член базового класса является *защищенным*, то и член производного класса должен быть *защищенным*. Однако это ограничение языка C#, а не CLR. При наследовании базовому классу CLR позволяет понижать, но не повышать уровень доступа к члену. Например, *защищенный* метод базового класса можно переопределить в производном классе в *открытый*, но не в *закрытый*. Это необходимо, чтобы пользователь производного типа всегда мог получить доступ к методу базового класса путем приведения к базовому типу. Если бы среда CLR разрешала накладывать более жесткие ограничения на доступ к методу в производном типе, она выдавала бы желаемое за действительное.

Статические классы

Существуют классы, не предназначенные для создания экземпляров, например Console, Math, Environment и ThreadPool. У этих классов есть только статические методы. В сущности, такие классы существуют лишь для группировки логически связанных членов. Например, класс Math объединяет методы, выполняющие математические операции. В C# такие классы определяются с ключевым словом static. Его разрешается применять только к классам, но не к структурам (значимым типам), поскольку CLR всегда разрешает создавать экземпляры значимых типов, и нет способа обойти это ограничение.

Компилятор налагает на *статический* класс ряд ограничений.

- ❑ Класс должен быть прямым потомком System.Object — наследование любому другому базовому классу лишено смысла, поскольку наследование применимо только к объектам, а создать экземпляр *статического* класса невозможно.

- ❑ Класс не должен реализовывать никаких интерфейсов, поскольку методы интерфейса можно вызывать только через экземпляры класса.
- ❑ В классе можно определять только *статические* члены (поля, методы, свойства и события). Любые экземплярные члены вызовут ошибку компиляции.
- ❑ Класс нельзя использовать в качестве поля, параметра метода или локальной переменной, поскольку это подразумевает существование переменной, ссылающейся на экземпляр, что запрещено. Обнаружив подобное обращение со статическим классом, компилятор вернет сообщение об ошибке.

Приведем пример *статического* класса, в котором определены *статические* члены; сам по себе класс не представляет практического интереса.

using System;

```
public static class AStaticClass {  
    public static void AStaticMethodQ { }  
  
    public static String AStaticProperty {  
        get { return s_AStaticField; }  
        set { s_AStaticField = value; }  
    }  
  
    private static String s_AStaticField;  
  
    public static event EventHandler AStaticEvent;  
}
```

На рис. 6.2 приведен результат дизассемблирования с помощью утилиты ILDasm.exe библиотечной (DLL) сборки, полученной при компиляции приведенного фрагмента кода. Как видите, определение класса с ключевым словом

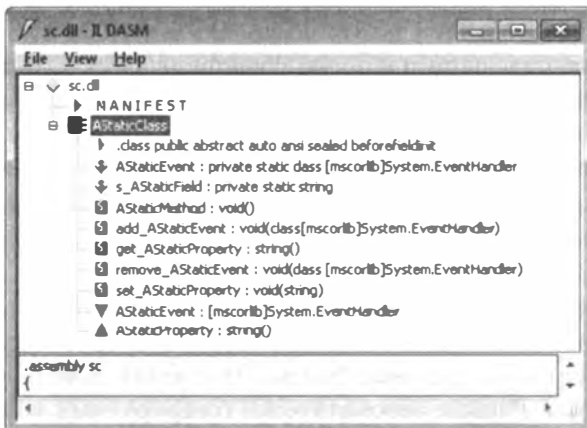


Рис. 6.2. Судя по метаданным, выводимым с помощью утилиты в ILDasm.exe, класс является абстрактным и изолированным

static заставляет компилятор C# сделать этот класс абстрактным (abstract) и изолированным (sealed). Более того, компилятор не создает в классе метод конструктора экземпляров (.ctor).

Частичные классы, структуры и интерфейсы

Частичные классы, структуры и интерфейсы поддерживаются исключительно компиляторами C# и некоторых других языков, а CLR ничего о них не знает. На самом деле, этот раздел добавлен скорее для полноты изложения материала, ведь книга рассказывает о возможностях CLR при использовании C#.

Ключевое слово `partial` говорит компилятору C#, что исходный код класса, структуры или интерфейса может располагаться в нескольких файлах. Есть три основные причины, по которым исходный код разбивается на несколько файлов.

- ❑ **Управление версиями.** Представьте, что определение типа содержит большой объем исходного кода. Если этот тип будут одновременно редактировать два программиста, по завершении работы им придется каким-то образом объединять свои результаты, что весьма неудобно. Ключевое слово `partial` позволяет разбить исходный код типа на несколько файлов, чтобы один и тот же тип могли одновременно редактировать несколько программистов.
- ❑ **Разделение файла или структуры на конечные логические модули внутри файла.** Иногда требуется создать один тип для решения разных задач. Для упрощения реализации я иногда объявляю одинаковые типы повторно внутри одного файла. Затем в каждой части такого типа я реализую по одной функциональной особенности типа со всеми его полями, методами, свойствами, событиями и т. д. Это позволяет мне упростить наблюдение за членами, обеспечивающими единую функциональность и объединенными в группу. Я также могу легко закомментировать частичный тип с целью удаления всей функциональности из класса или замены ее другой (путем использования новой части частичного типа).
- ❑ **Разделители кода.** При создании в Microsoft Visual Studio нового проекта Windows Forms или Web Forms некоторые файлы с исходным кодом создаются автоматически. Они называются шаблонными. При использовании конструкторов форм Visual Studio в процессе создания и редактирования элементов управления формы автоматически генерирует весь необходимый код и помещает его в отдельные файлы. Это значительно повышает продуктивность работы. Раньше автоматически генерируемый код попадал в тот же файл, где программист писал свой исходный код. Проблема была в том, что при случайном изменении сгенерированного кода конструктор форм

переставал корректно работать. Начиная с Visual Studio 2005, при создании нового проекта Visual Studio создает два исходных файла: один предназначен для программиста, а в другой помещается код, создаваемый редактором форм. В результате вероятность случайного изменения генерируемого кода существенно снижается.

Ключевое слово `partial` применяется к типам во всех файлах с определением типа. При компиляции компилятор объединяет эти файлы, и готовый тип помещается в результирующий файл сборки с расширением `exe` или `dll` (или в файл модуля с расширением `netmodule`). Как уже отмечалось, частичные типы реализуются только компилятором C#, поэтому все файлы с исходным кодом типа необходимо писать на одном языке и компилировать их в единый модуль.

Компоненты, полиморфизм и версии

Объектно-ориентированное программирование (ООП) на протяжении многих лет неизменно пользуется популярностью. В поздние 70-е и ранние 80-е годы объектно-ориентированные приложения были намного меньше по размеру, и весь код приложения разрабатывался в одной компании. Разумеется, в то время уже были операционные системы, позволяющие приложениям по максимуму использовать их возможности, но современные ОС предлагают намного больше функций.

Сложность программного обеспечения существенно возросла, к тому же пользователи требуют от приложений богатых функциональных возможностей — графического интерфейса, меню, различных устройств ввода-вывода (мышь, принтер, планшет), сетевых функций и т. п. Все это привело к существенному расширению функциональности операционных систем и платформ разработки в последние годы. Более того, сейчас уже не представляется возможным или эффективным писать приложение «с нуля» и разрабатывать все необходимые компоненты самостоятельно. Современные приложения состоят из компонентов, разработанных многими компаниями. Эти компоненты объединяются в единое приложение в рамках парадигмы ООП.

При компонентной разработке приложений (Component Software Programming, CSP) идеи ООП используются на уровне компонентов. Вот некоторые свойства компонента.

- ❑ Компонент (сборка в .NET) можно публиковать.
- ❑ Компоненты уникальны и идентифицируются по имени, версии, региональным стандартам и открытому ключу.
- ❑ Компонент сохраняет свою уникальность (код одной сборки никогда статически не связывается с другой сборкой — в .NET применяется только динамическое связывание).

- ❑ В компоненте всегда четко указана зависимость от других компонентов (ссылочные таблицы метаданных).
- ❑ В компоненте задокументированы его классы и члены. В С# даже разрешается помещать в код компонента XML-документацию — для этого служит параметр `/doc` командной строки компилятора.
- ❑ В компоненте определены требуемые разрешения на доступ. Для этого в CLR существует механизм защиты доступа к коду (Code Access Security, CAS).
- ❑ Опубликованный компонентом интерфейс (объектная модель) не изменяется во всех его служебных версиях. *Служебной версией* (servicing) называют новую версию компонента, обратно совместимую с оригинальной. Обычно служебная версия содержит исправления ошибок, исправления системы безопасности и небольшие корректировки функциональности. Однако в нее нельзя добавлять новые зависимости или разрешения безопасности.

Как видите, в компонентном программировании большое внимание уделяют управлению версиями. Компоненты постоянно изменяют, к тому же они поставляются в разное время. Необходимость управления версиями существенно усложняет компонентное программирование по сравнению с ООП, где все приложение пишет, тестирует и предоставляет одна компания.

В .NET номер версии состоит из четырех частей: *старшего* (major) и *младшего* (minor) номеров версии, номера *компоновки* (build) и номера *редакции* (revision). Например, у сборки с номером 1.2.3.4 старший номер версии — 1, младший номер версии — 2, номер компоновки — 3 и номер редакции — 4. Старший и младший номера обычно определяют уникальность сборки, а номера компоновки и редакции указывают на служебную версию.

Допустим, компания поставила сборку версии 2.7.0.0. Если впоследствии нужно предоставить сборку с исправленными ошибками, выпускают новую сборку, в которой изменяют только номера компоновки и редакции, например 2.7.1.34. То есть сборка является служебной версией и обратно совместима с оригинальной (2.7.0.0).

В то же время, если компания выпустит новую версию сборки, в которую внесены значительные изменения, а обратная совместимость не гарантируется, нужно изменить старший и/или младший номер версии (например, 3.0.0.0).

ПРИМЕЧАНИЕ

Все сказанное является лишь рекомендацией. К сожалению, CLR никак не анализирует номер версии, и если сборка зависит от версии 1.2.3.4 другой сборки, CLR будет пытаться загрузить только версию 1.2.3.4 (если только не задействовать механизм перенаправления связывания).

После ознакомления с порядком присвоения номера версии новому компоненту самое время узнать о возможностях CLR и языков программирования (таких как C#), позволяющих разработчикам писать код, устойчивый к изменениям компонентов.

Проблемы управления версиями возникают, когда тип, определенный в одном компоненте (сборке), используется в качестве базового класса для типа другого компонента (сборки). Ясно, что изменения в базовом классе могут влиять на поведение производного класса. Эти проблемы особенно характерны для полиморфизма, когда в производном типе переопределяются виртуальные методы базового типа.

В C# для типов и/или их членов есть пять ключевых слов, влияющих на управление версиями, причем они напрямую связаны с соответствующими возможностями CLR. В табл. 6.2 показано, как ключевые слова влияют на определение типа или члена типа.

Таблица 6.2. Ключевые слова C# и их влияние на управление версиями компонентов

Ключевое слово C#	Тип	Метод/Свойство/Событие	Константа/Поле
abstract	Экземпляры такого типа создавать нельзя	Член необходимо переопределить и реализовать в производном типе — только после этого можно создавать экземпляры производного типа	(запрещено)
virtual	(запрещено)	Член можно переопределить в производном типе	(запрещено)
override	(запрещено)	Член переопределяется в производном типе	(запрещено)
sealed	Тип нельзя использовать	Член нельзя переопределить в производном типе. Это ключевое слово может применяться только к методу, переопределяющему виртуальный метод	(запрещено)
new	Применительно к вложенному типу, методу, свойству, событию, константе или полю означает, что член никак не связан с похожим членом, который может существовать в базовом классе		

О назначении и использовании этих ключевых слов рассказывается в разделе «Работа с виртуальными методами при управлении версиями типов», но прежде необходимо рассмотреть механизм вызова виртуальных методов в CLR.

Вызов виртуальных методов, свойств и событий в CLR

В этом разделе речь идет только о методах, но все сказанное относится и к виртуальным свойствам и событиям, поскольку они, как показано далее, на самом деле реализованы как методы.

Методы содержат код, выполняющий некоторые действия над типом (статические методы) или экземпляром типа (нестатические). У каждого метода есть имя, сигнатура и возвращаемое значение, которое может быть пустым (void). У типа может быть несколько методов с одним именем, но с разным числом параметров или разными возвращаемыми значениями. Можно также определить два метода с одним и тем же именем и параметрами, но с разными типами возвращаемого значения. Однако эта «возможность» большинством языков не используется (за исключением IL) — все они требуют, чтобы методы с одинаковым именем различались параметрами, а возвращаемое значение при определении уникальности метода игнорируется. Впрочем, благодаря операторам преобразования язык C# смягчает это ограничение (см. главу 8).

Определим класс `Employee` с тремя различными вариантами методов.

```
internal class Employee {  
    // Невиртуальный экземплярный метод  
    public Int32 GetYearsEmployedQ { ... }  
    // Виртуальный метод (виртуальный, значит, экземплярный)  
    public virtual String GenProgressReportQ { ... }  
    // Статический метод  
    public static Employee Lookup(String name) { ... }  
}
```

При компиляции этого кода компилятор помещает три записи в таблицу определений методов сборки. Каждая запись содержит флаги, указывающие, является ли метод экземплярным, виртуальным или статическим.

При компиляции кода, ссылающегося на эти методы, компилятор проверяет флаги в определении методов, чтобы выяснить, какой IL-код нужно вставить для корректного вызова методов. В CLR есть две инструкции для вызова метода:

- Инструкция `call` используется для вызова статических, экземплярных и виртуальных методов. Если с помощью этой инструкции вызывается статический метод, необходимо указать тип, в котором определяется метод. При вызове экземплярного или виртуального метода необходимо указать переменную, ссылающуюся на объект, причем в `call` подразумевается, что эта переменная не равна `null`. Иначе говоря, сам тип переменной указывает, в каком типе определен необходимый метод. Если в типе переменной метод не определен, проверяются базовые типы. Инструкция `call` часто служит для неvirtуального вызова виртуального метода.

- ❑ Инструкция `callvirt` используется только для вызова экземплярных и виртуальных методов. При вызове необходимо указать переменную, ссылающуюся на объект. Если с помощью этой инструкции вызывается неvirtуальный экземплярный метод, тип переменной показывает, где определен необходимый метод. При использовании `callvirt` для вызова виртуального экземплярного метода CLR определяет настоящий тип объекта, на который ссылается переменная, и вызывает метод полиморфно. При компиляции такого вызова JIT-компилятор генерирует код для проверки значения переменной — если оно равно `null`, CLR сгенерирует исключение `NullReferenceException`. Из-за этой дополнительной проверки инструкция `callvirt` выполняется немного медленнее, чем `call`. Проверка на `null` выполняется даже при вызове неvirtуального экземплярного метода.

Давайте посмотрим, как эти инструкции используются в C#.

```
using System;
```

```
public sealed class Program { public static void Main() {
    Console.WriteLine(); // Вызов статического метода

    Object o = new Object();
    o.GetHashCode();      // Вызов виртуального экземплярного метода
    o.GetType();          // Вызов неvirtуального экземплярного метода
}
}
```

После компиляции результирующий IL-код выглядит следующим образом.

```
.method public hidebysig static void Main() cil managed {
    .entrypoint
    // Code size 26 (0x1a)
    .maxstack 1
    .locals init (object V_0)
    IL_0000: call void System.Console::WriteLine()
    IL_0005: newobj instance void System.Object::ctor()
    IL_000a: stloc.0
    IL_000b: ldloc.0
    IL_000c: callvirt instance int32 System.Object::GetHashCode()
    IL_0011: pop
    IL_0012: ldloc.0
    IL_0013: callvirt instance class System.Type System.Object::GetType()
    IL_0018: pop
    IL_0019: ret
} // end of method Program::Main
```

Поскольку метод `WriteLine` является статическим, компилятор C# использует для его вызова инструкцию `call`. Для вызова виртуального метода `GetHashCode` применяется инструкция `callvirt`. Наконец, метод `GetType` также вызывается

с помощью инструкции `callvirt`. Это выглядит странным, поскольку метод `GetType` не виртуальный. Тем не менее код работает, потому что во время JIT-компиляции CLR выясняет, что `GetType` — это не виртуальный метод, и вызывает его не виртуально.

Разумеется, возникает вопрос: почему компилятор C# не использует инструкцию `call`? Так решили разработчики C# — JIT-компилятор должен создавать код проверки, не равен ли `null` вызывающий объект. Поэтому вызовы не виртуальных экземплярных методов выполняются чуть медленнее, чем могли бы. Следующий код в C# вызовет исключение `NullReferenceException`, хотя в некоторых языках все работает отлично.

```
using System;
```

```
public sealed class Program {  
    public Int32 GetFive() { return 5; }  
    public static void Main() {  
        Program p = null;  
        Int32 x = p.GetFive(); // В C# генерируется NullReferenceException  
    }  
}
```

Теоретически с этим кодом все в порядке. Хотя переменная `p` равна `null`, для вызова не виртуального метода `GetFive` среде CLR необходимо узнать только тип `p`, а это `Program`. При вызове `GetFive` аргумент `this` равен `null`, но в методе `GetFive` он не используется, поэтому исключения нет. Однако компилятор C# вместо инструкции `call` вставляет `callvirt`, поэтому код вызовет исключение `NullReferenceException`.

ВНИМАНИЕ

Если метод определен как не виртуальный, не рекомендуется в дальнейшем делать его виртуальным. Причина в том, что некоторые компиляторы для вызова не виртуального метода используют инструкцию `call` вместо `callvirt`. Если метод сделать виртуальным и не перекомпилировать ссылающийся на него код, виртуальный метод будет вызван не виртуально, в результате приложение может повести себя непредсказуемо. Если ссылающийся код написан на C#, проблем не будет, поскольку в C# все экземплярные методы вызываются с помощью инструкции `callvirt`. Проблемы возможны, когда ссылающийся код написан на другом языке.

Иногда компилятор вместо `callvirt` использует для вызова виртуального метода команду `call`. Следующий пример показывает, почему это действительно бывает необходимо.

```
internal class SomeClass {  
    // ToString - виртуальный метод базового класса Object  
    public override String ToString() {  
  
        // Компилятор использует команду call для не виртуального вызова
```

```
// метода ToString класса Object

// Если бы компилятор вместо call использовал callvirt, этот
// метод продолжал бы рекурсивно вызывать сам себя до переполнения стека
return base.ToString();
}
}
```

При вызове виртуального метода `base.ToString` компилятор C# вставляет команду `call`, чтобы метод `ToString` базового типа вызывался неvirtуально. Это необходимо, ведь если `ToString` вызвать виртуально, вызов будет выполняться рекурсивно до переполнения стека потока, что совершенно нежелательно.

Компиляторы стремятся использовать команду `call` при вызове методов, определенных значимыми типами, поскольку они изолированные. В этом случае полиморфизм невозможен даже для виртуальных методов, и вызов выполняется быстрее. Кроме того, сама природа экземпляра значимого типа гарантирует, что он никогда не будет равен `null`, поэтому исключение `NullReferenceException` не возникнет. Наконец, для виртуального вызова виртуального метода значимого типа CLR необходимо получить ссылку на объект значимого типа, чтобы воспользоваться его таблицей методов, а это требует упаковки значимого типа. Упаковка создает большую нагрузку на кучу, увеличивая частоту сборки мусора и снижая производительность.

Независимо от используемой для вызова экземплярного или виртуального метода инструкции — `call` или `callvirt` — эти методы всегда в качестве первого параметра получают скрытый аргумент `this`, ссылающийся на объект, над которым производятся действия.

При проектировании типа следует стремиться минимизировать количество виртуальных методов. Во-первых, виртуальный метод вызывается медленнее неvirtуального. Во-вторых, JIT-компилятор не может подставлять (`inline`) виртуальные методы, что также бьет по производительности. В-третьих, как показано далее, виртуальные методы затрудняют управление версиями компонентов. В-четвертых, при определении базового типа часто создается набор перегруженных методов. Чтобы сделать их полиморфными, лучше всего сделать наиболее сложный метод виртуальным, оставив другие методы неvirtуальными. Кстати, следование этому правилу поможет управлять версиями компонентов, не нарушая работу производных типов. Приведем пример:

```
public class Set {
    private Int32 m_length = 0;

    // Этот перегруженный метод – неvirtуальный
    public Int32 Find(Object value) {
        return Find(value, 0, m_length); }
    // Этот перегруженный метод – неvirtуальный
    public Int32 Find(Object value, Int32 startIndex) {
```

продолжение ➤

```
return Find(value, 0, m_length); }

// Наиболее функциональный метод сделан виртуальным
// и может быть переопределен
public virtual Int32 Find(Object value, Int32 startIndex, Int32 endIndex) {
    // Здесь находится настоящая реализация, которую можно переопределить...
}
// Другие методы
}
```

Разумное использование видимости типов и модификаторов доступа к членам

В .NET Framework приложения состоят из типов, определенных в многочисленных сборках, созданных различными компаниями. Это означает практически полное отсутствие контроля над используемыми компонентами и типами. У разработчика нет доступа к исходным кодам компонентов (он может даже не знать, на каком языке они написаны), к тому же версии компонентов обновляются в разное время. Более того, из-за полиморфизма и наличия защищенных членов разработчик базового класса должен доверять коду разработчика производного класса. В свою очередь, разработчик производного класса должен доверять коду, наследуемому от базового класса. Это лишь часть ограничений, с которыми приходится сталкиваться при разработке компонентов и типов.

Далее я расскажу о том, как правильно задавать видимость типов и модификаторы доступа к членам.

В первую очередь, при определении нового типа компиляторам следовало бы по умолчанию делать его изолированным. Вместо этого большинство компиляторов (в том числе C#) поступают как раз наоборот, считая, что программист при необходимости сам может изолировать класс с помощью ключевого слова `sealed`. Было бы неплохо, если бы неправильное, на мой взгляд, поведение, предлагаемое по умолчанию, в следующих версиях компиляторов изменилось. Есть три веские причины в пользу использования изолированных классов.

- ❑ **Управление версиями.** Если класс изначально изолирован, его впоследствии можно сделать неизолированным, не нарушая совместимости. Однако обратное невозможно, поскольку это нарушило бы работу всех производных классов. Кроме того, если в неизолированном классе определены неизолированные виртуальные методы, необходимо сохранять порядок вызова виртуальных методов в новых версиях, иначе в будущем возникнут проблемы с производными типами.
- ❑ **Производительность.** Как уже отмечалось, неvirtуальные методы вызываются быстрее виртуальных, поскольку для последних CLR во время выполнения проверяет тип объекта, чтобы выяснить, где находится метод. Однако, встретив вызов виртуального метода в изолированном типе, JIT-компилятор

может сгенерировать более эффективный код, задействовав неvirtуальный вызов. Это возможно потому, что у изолированного класса не может быть производных классов. Например, в следующем коде JIT-компилятор может вызвать виртуальный метод ToString неvirtуально:

```
using System;
public sealed class Point {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) { m_x = x; m_y = y; }

    public override String ToString0 {
        return String.Format("{0}, {1}", m_x, m_y);
    }
}

public static void Main() {
    Point p = new Point(3, 4);

    // Компилятор C# вставит здесь инструкцию callvirt,
    // но JIT-компилятор оптимизирует этот вызов и сгенерирует код
    // для неvirtуального вызова ToString.
    // поскольку p имеет тип Point, являющийся изолированным
    Console.WriteLine(p.ToString());
}
```

- **Безопасность и предсказуемость.** Состояние класса должно быть надежно защищено. Если класс не изолирован, производный класс может изменить его состояние, воспользовавшись незащищенными полями или методами базового класса, изменяющими его доступные незакрытые поля. Кроме того, в производном классе можно переопределить виртуальные методы и не вызывать реализацию соответствующих методов базового класса. Виртуальные методы, свойства и события базового класса позволяют контролировать его поведение и состояние в производном классе, что при неумелом обращении может вызвать непредсказуемое поведение и проблемы с безопасностью.

Беда в том, что изолированные классы не всегда удобны в использовании. Разработчику приложения может понадобиться производный тип, в котором будут добавлены дополнительные поля или другая информация о состоянии. Кроме того, в производном типе могут потребоваться дополнительные методы для работы с этими полями. Однако в изолированных классах все это запрещено, поэтому я предложил разработчикам CLR ввести новый модификатор доступа к классу — *замкнутый* (closed).

Замкнутый класс можно использовать в качестве базового, но его поведение нельзя изменить в производном классе. Кроме того, производному классу будут доступны только открытые члены замкнутого класса. Это позволит из-

менять базовый класс, не опасаясь за работоспособность производного класса. Идеальным будет, если разработчики компиляторов сделают `closed` модификатором доступа, предлагаемым по умолчанию, поскольку это оптимальный компромисс между безопасностью и свободой выбора. Будем надеяться, что эта идея реализуется в будущих версиях CLR и языков программирования.

Кстати, есть способ (правда, очень неудобный), позволяющий имитировать модификатор `closed`. Для этого при реализации класса необходимо изолировать все наследуемые им виртуальные методы (включая методы, определенные в `System.Object`). Кроме того, необходимо отказаться от защищенных и виртуальных методов, затрудняющих управление версиями. Например:

```
public class SimulatedClosedClass : Object {
    public sealed override Boolean Equals(Object obj) {
        return base.Equals(obj);
    }

    public sealed override Int32 GetHashCode() {
        return base.GetHashCode();
    }
    public sealed override String ToString() {
        return base.ToString();
    }
    // К сожалению, С# не разрешает изолировать метод Finalize
    // Определите дополнительные открытые или закрытые члены...

    // Не определяйте защищенные или виртуальные методы
}
```

К сожалению, CLR и компиляторы пока не поддерживают замкнутые типы. Вот несколько правил, которым я следую при проектировании классов:

- ❑ Если класс не предназначен для наследования, его следует явно объявить изолированным. Как уже отмечалось, С# и другие компиляторы по умолчанию делают класс неизолрированным. Если нет необходимости в предоставлении другим сборкам доступа к классу, его следует сделать внутренним. К счастью, именно так ведет себя по умолчанию компилятор С#. Чтобы определить класс, предназначенный для создания производных классов, одновременно запретив его специализацию, следует имитировать замкнутый класс указанным ранее способом.
- ❑ Поля данных класса всегда следует объявлять закрытыми. По умолчанию С# поступает именно так. Вообще говоря, я бы предпочел, чтобы в С# остались только закрытые поля. Открытый доступ к состоянию объекта — верный путь к непредсказуемому поведению и проблемам с безопасностью. При объявлении полей внутренними (`internal`) также могут возникнуть проблемы, поскольку даже внутри одной сборки очень трудно отследить

все обращения к полям, особенно когда над ней работает несколько разработчиков.

- ❑ Методы, свойства и события класса всегда следует делать закрытыми и не-виртуальными. К счастью, C# по умолчанию делает именно так. Разумеется, чтобы типом можно было воспользоваться, некоторые методы, свойства и события должны быть открытыми, но лучше не делать их защищенными или внутренними, поскольку это может сделать тип уязвимым. Впрочем, защищенный или внутренний член все-таки лучше виртуального, поскольку последний предоставляет производному классу большие возможности и всецело зависит от корректности его поведения.
- ❑ В ООП есть поговорка: «лучший метод борьбы со сложностью — добавление новых типов». Если реализация алгоритма чрезмерно усложняется, следует определить вспомогательные типы, инкапсулирующие часть функциональности. Если вспомогательные типы используются в единственном супертипе, следует сделать их вложенными. Это позволит ссылаться на них через супертип и позволит им обращаться к защищенным членам супертипа. Однако существует правило проектирования, примененное в утилите FxCopCmd.exe Visual Studio и рекомендуемое определять общедоступные вложенные типы в области видимости файла или сборки (за пределами супертипа), поскольку некоторые разработчики считают синтаксис обращения к вложенным типам громоздким.

Работа с виртуальными методами при управлении версиями типов

Как уже отмечалось, управление версиями — важный аспект компонентного программирования. Некоторых проблем я коснулся в главе 3 (там речь шла о сборках со строгими именами и обсуждались меры, позволяющие администраторам гарантировать привязку приложения именно к тем сборкам, с которыми оно было скомпоновано и протестировано). Однако при управлении версиями возникают и другие сложности с совместимостью на уровне исходного кода. В частности, следует быть очень осторожными при добавлении и изменении членов базового типа. Рассмотрим несколько примеров.

Пусть разработчиками компании CompanyA спроектирован тип Phone:

```
namespace CompanyA {  
    public class Phone {  
        public void Dial() {  
            Console.WriteLine("Phone.Dial");  
            // Выполнить действия по набору телефонного номера  
        }  
    }  
}
```

А теперь представьте, что в компании CompanyB спроектировали другой тип, BetterPhone, использующий тип Phone в качестве базового:

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {
        public void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection(); base.Dial();
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // Выполнить действия по набору телефонного номера
        }
    }
}
```

При попытке скомпилировать свой код разработчики компании CompanyB получают от компилятора C# предупреждение:

```
warning CS0108: The keyword new is required on 'BetterPhone.Dial()' because it
hides inherited member 'Phone.Dial()'
```

Смысл в том, что метод Dial, определяемый в типе BetterPhone, скроет одноименный метод в Phone. В новой версии метода Dial его семантика может стать совсем иной, нежели та, что определена программистами компании CompanyA в исходной версии метода.

Предупреждение о таких потенциальных семантических несоответствиях — очень полезная функция компилятора. Компилятор также подсказывает, как избавиться от этого предупреждения: нужно поставить ключевое слово new перед определением метода Dial в классе BetterPhone. Вот как выглядит исправленный класс BetterPhone:

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {

        // Этот метод Dial никак не связан с одноименным методом класса Phone
        public new void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // Выполнить действия по установлению соединения
        }
    }
}
```

Теперь компания CompanyB может использовать в своем приложении тип BetterPhone следующим образом:

```
public sealed class Program {  
    public static void Main() {  
        CompanyB.BetterPhone phone = new CompanyB.BetterPhone(); phone.Dial();  
    }  
}
```

При выполнении этого кода выводится следующая информация:

```
BetterPhone.Dial  
BetterPhone.EstablishConnection  
Phone.Dial
```

Результат свидетельствует о том, что код выполняет именно те действия, которые нужны компании CompanyB. При вызове Dial вызывается новая версия этого метода, определенная в типе BetterPhone. Она сначала вызывает виртуальный метод EstablishConnection, а затем — исходную версию метода Dial из базового типа Phone.

А теперь представим, что несколько компаний решили использовать тип Phone, созданный в компании CompanyA. Допустим также, что все они сочли полезным установление соединения в самом методе Dial. Эти отзывы заставили разработчиков компании CompanyA усовершенствовать класс Phone:

```
namespace CompanyA {  
    public class Phone {  
        public void Dial() {  
            Console.WriteLine("Phone.Dial");  
            EstablishConnection();  
            // Выполнить действия по набору телефонного номера  
        }  
  
        protected virtual void EstablishConnection() {  
            Console.WriteLine("Phone.EstablishConnection");  
            // Выполнить действия по установлению соединения  
        }  
    }  
}
```

В результате теперь разработчики компании CompanyB при компиляции своего типа BetterPhone (производного от новой версии Phone) получают следующее предупреждение:

```
warning CS0114: 'BetterPhone.EstablishConnection()' hides inherited member  
'Phone.EstablishConnection()'. To make the current member override that  
implementation, add the override keyword. Otherwise, add the new keyword
```

В нем говорится о том, что 'BetterPhone.EstablishConnection()' скрывает унаследованный член 'Phone.EstablishConnection()', и чтобы текущий член

переопределил реализацию, нужно вставить ключевое слово `override`; в противном случае нужно вставить ключевое слово `new`.

То есть компилятор предупреждает, что как `Phone`, так и `BetterPhone` предлагают метод `EstablishConnection`, семантика которого может отличаться в разных классах. В этом случае простая перекомпиляция `BetterPhone` больше не может гарантировать, что новая версия метода будет работать так же, как прежняя, определенная в типе `Phone`.

Если в компании `CompanyB` решат, что семантика метода `EstablishConnection` в этих двух типах отличается, компилятору будет указано, что «правильными» являются методы `Dial` и `EstablishConnection`, определенные в `BetterPhone`, и они не связаны с одноименными методами из базового типа `Phone`. Разработчики компании `CompanyB` смогут заставить компилятор выполнить нужные действия, оставив в определении метода `Dial` ключевое слово `new` и добавив его же в определение `EstablishConnection`:

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {

        // Ключевое слово 'new' оставлено, чтобы указать,
        // что этот метод не связан с методом Dial базового типа
        public new void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        // Здесь добавлено ключевое слово 'new', чтобы указать, что этот
        // метод не связан с методом EstablishConnection базового типа
        protected new virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // Выполнить действия для установления соединения
        }
    }
}
```

Здесь ключевое слово `new` заставляет компилятор генерировать метаданные, информирующие CLR, что определенные в `BetterPhone` методы `Dial` и `EstablishConnection` следует рассматривать как новые функции, введенные в этом типе. При этом CLR будет известно, что одноименные методы типов `Phone` и `BetterPhone` никак не связаны.

При выполнении того же приложения (метода `Main`) выводится информация:

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
Phone.EstablishConnection
```

Отсюда видно, что, когда Main обращается к методу Dial, вызывается версия, определенная в BetterPhone. Далее Dial вызывает виртуальный метод EstablishConnection, также определенный в BetterPhone. Когда метод EstablishConnection типа BetterPhone возвращает управление, вызывается метод Dial типа Phone, вызывающий метод EstablishConnection этого типа. Но поскольку метод EstablishConnection в типе BetterPhone помечен ключевым словом new, вызов этого метода не считается переопределением виртуального метода EstablishConnection, исходно определенного в типе Phone. В результате метод Dial типа Phone вызывает метод EstablishConnection, определенный в типе Phone, что и требовалось от программы.

ПРИМЕЧАНИЕ

Без ключевого слова new разработчики типа BetterPhone не смогли бы использовать в нем имена методов Dial и EstablishConnection. Если же изменить имена этих методов, то негативный эффект изменений, скорее всего, затронет всю программную основу, нарушая совместимость на уровне исходного текста и двоичного кода. Обычно такого рода изменения с далеко идущими последствиями нежелательны, особенно в средних и крупных проектах. Однако если изменение имени метода коснется лишь необходимости обновления исходного текста, следует пойти на это, чтобы одинаковые имена методов Dial и EstablishConnection, обладающие разной семантикой в разных типах, не вводили в заблуждение других разработчиков.

Альтернативный вариант таков: компания CompanyB, получив от компании CompanyA новую версию типа Phone, решает, что текущая семантика методов Dial и EstablishConnection типа Phone — это именно то, что нужно. В этом случае в компании CompanyB полностью удаляют метод Dial из типа BetterPhone. Кроме того, поскольку теперь разработчикам компании CompanyB нужно указать компилятору, что метод EstablishConnection типа BetterPhone связан с одноименным методом типа Phone, нужно удалить из его определения ключевое слово new. Однако простого удаления ключевого слова здесь недостаточно, так как компилятор не поймет предназначения метода EstablishConnection типа BetterPhone. Чтобы выразить свои намерения явно, разработчик из компании CompanyB должен, помимо прочего, изменить модификатор определенного в типе BetterPhone метода EstablishConnection с virtual на override. Итак, представим код новой версии BetterPhone:

```
namespace CompanyB {  
    public class BetterPhone : CompanyA.Phone {  
  
        // Метод Dial удален (так как он наследуется от базового типа)
```

```
// Здесь ключевое слово new удалено, а модификатор virtual заменен
// на override, чтобы указать, что этот метод связан с методом
// EstablishConnection из базового типа
protected override void EstablishConnection() {
    Console.WriteLine("BetterPhone.EstablishConnection");
    // Выполнить действия по установлению соединения
}
}
```

Теперь при исполнении того же приложения (метода Main) выводится:

```
Phone.Dial
BetterPhone.EstablishConnection
```

Видно, что когда Main вызывает метод Dial, вызывается версия этого метода, определенная в типе Phone и унаследованная от него типом BetterPhone. Далее, когда метод Dial, определенный в типе Phone, вызывает виртуальный метод EstablishConnection, вызывается одноименный метод типа BetterPhone, так как он переопределяет виртуальный метод EstablishConnection, определяемый типом Phone.

Глава 7. Константы и поля

В этой главе показано, как добавить к типу члены, являющиеся данными. В частности, мы рассмотрим константы и поля.

Константы

Константа (constant) — это идентификатор, значение которого никогда не меняется. При определении идентификатора константы компилятор должен получить его значение во время компиляции. Затем компилятор сохраняет значение константы в метаданных модуля. Это значит, что константы можно определять только для таких типов, которые компилятор считает элементарными. В C# следующие типы считаются элементарными и могут быть использованы для определения констант: Boolean, Char, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal и String. Тем не менее C# позволяет определить константную переменную, не относящуюся к элементарному типу, если присвоить ей значение null:

```
using System;
```

```
public sealed class SomeType {  
    // Некоторые типы не являются элементарными, но C# допускает существование  
    // константных переменных этих типов после присваивания значения null  
    public const SomeType Empty = null;  
}
```

Другой важный момент: поскольку значение констант никогда не меняется, константы всегда считаются частью типа. Иначе говоря, константы считаются статическими, а не экземплярными членами. Определение константы приводит в конечном итоге к созданию метаданных.

Встретив в исходном тексте идентификатор константы, компилятор просматривает метаданные модуля, в котором она определена, извлекает значение константы и внедряет его в генерируемый им IL-код. Поскольку значение константы внедряется прямо в код, в период выполнения память для констант не выделяется. Кроме того, нельзя получать адрес константы и передавать ее по ссылке. Эти ограничения также означают, что изменять значения константы в разных версиях модуля нельзя, поэтому константу надо использовать, только когда точно известно, что ее значение никогда не изменится (хороший пример — определение константы `MaxInt16` со значением

32767). Поясню на примере, что я имею в виду. Возьмем код и скомпилируем его в DLL-сборку:

```
using System;
```

```
public sealed class SomeLibraryType {
    // ПРИМЕЧАНИЕ: C# не позволяет использовать для констант модификатор
    // static, поскольку всегда подразумевается, что константы являются
    // статическими
    public const Int32 MaxEntriesInList = 50;
}
```

Затем скомпилируем сборку приложения из следующего кода:

```
using System;
```

```
public sealed class Program {
    public static void Main() {
        Console.WriteLine("Max entries supported in list: " +
            SomeLibraryType.MaxEntriesInList);
    }
}
```

Нетрудно заметить, что код приложения содержит ссылку на константу `MaxEntriesInList`. При компоновке этого кода компилятор, обнаружив, что `MaxEntriesInList` — это литерал константы со значением 50, внедрит значение 50 типа `Int32` прямо в IL-код приложения. В сущности, после компоновки кода приложения DLL-сборка даже не будет загружаться в период выполнения, поэтому ее можно просто удалить с диска.

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 25 (0x19)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr "Max entries supported in list: "
    IL_0006: ldc.i4.s 50
    IL_0008: box [mscorlib]System.Int32
    IL_000d: call string [mscorlib]System.String::Concat(object, object)
    IL_0012: call void [mscorlib]System.Console::WriteLine(string)
    IL_0017: nop
    IL_0018: ret
} // Закрываем метод Program::Main
```

Теперь проблема управления версиями при использовании констант должна стать очевидной. Если разработчик изменит значение константы `MaxEntriesInList` на 1000 и заново опубликует DLL-сборку, это не повлияет на код самого приложения. Для того чтобы в приложении использовалось новое значение константы, его придется перекомпилировать. Нельзя применять константы во время

выполнения (а не во время компиляции), если модуль должен задействовать значение, определенное в другом модуле. В этом случае вместо констант следует использовать предназначенные только для чтения поля, о которых речь идет в следующем разделе.

Поля

Поле (field) — это член данных, который хранит экземпляр значимого типа или ссылку на ссылочный тип. В табл. 7.1 приведены модификаторы, применяемые по отношению к полям.

Таблица 7.1. Модификаторы полей

Термин CLR	Термин C#	Описание
Static	static	Поле является частью состояния типа, а не объекта
Instance	(по умолчанию)	Поле связано с экземпляром типа, а не самим типом
InitOnly	readonly	Запись в поле разрешается только из кода конструктора
Volatile	volatile	Код, обращающийся к полю, не должен оптимизироваться компилятором, CLR или оборудованием с целью обеспечения безопасности потоков. Только следующие типы могут объявляться как волатильные: все ссылочные типы, Single, Boolean, Byte, SByte, Int16, UInt16, Int32, UInt32, Char, а также все перечислимые типы, основанные на типе Byte, SByte, Int16, UInt16, Int32 или UInt32. Волатильные поля рассматриваются в главе 28

Как видно из таблицы, общезыковая среда (CLR) поддерживает поля как типов (статические), так и экземпляров (нестатические). Динамическая память для хранения поля типа выделяется в пределах объекта типа, который создается при загрузке типа в домен приложений (см. главу 22), что обычно происходит при JIT-компиляции любого метода, ссылающегося на этот тип. Динамическая память для хранения экземплярных полей выделяется при создании экземпляра данного типа.

Поскольку поля хранятся в динамической памяти, их значения можно получить лишь в период выполнения. Поля также решают проблему управления версиями, возникающую при использовании констант. Кроме того, полю можно назначить любой тип данных, поэтому при определении полей можно не ограничиваться встроенными элементарными типами компилятора (что приходится делать при определении констант).

CLR поддерживает поля, предназначенные для чтения и записи (изменяемые), а также поля, предназначенные только для чтения (неизменяемые). Большинство полей изменяемые. Это значит, что во время исполнения кода

значение таких полей может многократно меняться. Данные же в неизменяемые поля можно записывать только при исполнении конструктора (который вызывается лишь раз — при создании объекта). Компилятор и механизм верификации гарантируют, что ни один метод, кроме конструктора, не сможет записать данные в поле, предназначенные только для чтения. Замечу, что для изменения такого поля можно задействовать отражение.

Попробуем решить проблему управления версиями в примере из раздела «Константы», используя статические неизменяемые поля. Вот новая версия кода DLL-сборки:

```
using System;
```

```
public sealed class SomeLibraryType {  
    // Модификатор static необходим, чтобы ассоциировать поле с его типом  
    public static readonly Int32 MaxEntriesInList = 50;  
}
```

Это единственное изменение, которое придется внести в исходный текст, при этом код приложения можно вовсе не менять, но чтобы увидеть его новые свойства, его придется перекомпилировать. Теперь при исполнении метода Main этого приложения CLR загружает DLL-сборку (так как она требуется во время выполнения) и извлекает значение поля MaxEntriesInList из динамической памяти, выделенной для его хранения. Естественно, это значение равно 50.

Допустим, разработчик сборки изменил значение поля с 50 на 1000 и скомпоновал сборку заново. При повторном исполнении код приложения автоматически задействует новое значение — 1000. В этом случае не обязательно компоновать код приложения заново, оно просто работает в том виде, в котором было (хотя и чуть медленнее). Однако здесь есть подводный камень: этот сценарий предполагает, что у новой сборки нет строгого имени или что политика управления версиями приложения заставляет CLR загружать именно эту новую версию сборки.

В следующем примере показано, как определять изменяемые статические поля, а также изменяемые и неизменяемые экземплярные поля:

```
public sealed class SomeType {  
    // Это статическое неизменяемое поле. Его значение рассчитывается  
    // и сохраняется в памяти при инициализации класса во время выполнения  
    public static readonly Random s_random = new Random();  
  
    // Это статическое изменяемое поле  
    private static Int32 s_numberOfWrites = 0;  
  
    // Это неизменяемое экземплярное поле  
    public readonly String Pathname = "Untitled";  
  
    // Это изменяемое экземплярное поле  
    private System.IO.FileStream m_fs;
```

```
public SomeType(String pathname) {  
    // Эта строка изменяет значение неизменяемого поля  
    // В данном случае это возможно, так как показанный далее код  
    // расположен в конструкторе  
    this.Pathname = pathname;  
}  
  
public String DoSomething() {  
    // Эта строка читает и записывает значение статического изменяемого поля  
    s_numberOfWrites = s_numberOfWrites + 1;  
  
    // Эта строка читает значение неизменяемого экземплярного поля  
    return Pathname;  
}  
}
```

Многие поля в нашем примере инициализируются при подстановке (inline). C# позволяет использовать этот удобный синтаксис для инициализации констант, а также изменяемых и неизменяемых полей. Как продемонстрировано в главе 8, C# считает, что инициализация поля при подстановке — это краткий синтаксис, позволяющий инициализировать поле во время исполнения конструктора. Вместе с тем, в C# возможны проблемы производительности, которые нужно учитывать при инициализации поля с использованием синтаксиса подстановки, а не присвоения в конструкторе. Они также обсуждаются в главе 8.

ВНИМАНИЕ

Неизменность поля ссылочного типа означает неизменность ссылки, которую этот тип содержит, а вовсе не объекта, на которую указывает ссылка, например:

```
public sealed class AType {  
    // InvalidChars всегда ссылается на один объект массива  
    public static readonly Char[] InvalidChars = new Char[] { 'A', 'B', 'C' };  
}  
  
public sealed class AnotherType {  
    public static void M() {  
        // Следующие строки кода вполне корректны, компилируются  
        // и успешно изменяют символы в массиве InvalidChars  
        AType.InvalidChars[0] = 'X';  
        AType.InvalidChars[1] = 'Y';  
        AType.InvalidChars[2] = 'Z';  
  
        // Следующая строка некорректна и не скомпилируется.  
        // так InvalidChars ссылается на неизменяемое  
        AType.InvalidChars = new Char[] { 'X', 'Y', 'Z' };  
    }  
}
```


Глава 8. Методы

В этой главе обсуждаются разновидности методов, которые могут определяться в типе, и разбирается ряд вопросов, касающихся методов. В частности, показано, как определяются методы-конструкторы (создающие экземпляры типов и сами типы), методы перегрузки операторов и методы преобразования (выполняющие явное и неявное приведение типов). Кроме того, здесь рассказывается, как передать методу параметры по ссылкам и как определить метод, принимающий переменное число параметров.

Конструкторы экземпляров и классы (ссылочные типы)

Конструкторы — это специальные методы, позволяющие корректно инициализировать новый экземпляр типа. В таблице определений, входящих в метаданные, методы-конструкторы всегда отмечают сочетанием `.ctor` (от *constructor*). При создании экземпляра объекта ссылочного типа выделяется память для полей данных экземпляра и инициализируются служебные поля (указатель на объект-тип и индекс блока синхронизации `SyncBlockIndex`), после чего вызывается конструктор экземпляра, устанавливающий исходное состояние нового объекта.

При создании объекта ссылочного типа выделяемая для него память всегда обнуляется до вызова конструктора экземпляра типа. Любые поля, не перезаписываемые конструктором явно, гарантированно содержат `0` или `null`.

В отличие от других методов конструкторы экземпляров не наследуются. Иначе говоря, у класса есть только те конструкторы экземпляров, которые определены в этом классе. Невозможность наследования означает, что к конструктору экземпляров нельзя применять модификаторы `virtual`, `new`, `override`, `sealed` и `abstract`. Если определить класс без явно заданных конструкторов, многие компиляторы (в том числе компилятор C#) создадут конструктор по умолчанию (без параметров), реализация которого просто вызывает конструктор без параметров базового класса.

Например, рассмотрим следующее определение класса:

```
public class SomeType { }
```

Это определение идентично определению:

```
public class SomeType {  
    public SomeType() : base() { }  
}
```

Для *абстрактных* классов компилятор создает конструктор по умолчанию с модификатором `protected`, в противном случае область действия будет открытой (`public`). Если в базовом классе нет конструктора без параметров, производный класс должен явно вызвать конструктор базового класса, иначе компилятор вернет ошибку. Для *статических* классов (изолированных и абстрактных) компилятор не создает конструктор по умолчанию.

В типе может определяться несколько конструкторов, при этом сигнатуры и уровни доступа к конструкторам обязательно должны отличаться. В случае верифицируемого кода конструктор экземпляров должен вызывать конструктор базового класса до обращения к какому-либо из унаследованных от него полей. Многие компиляторы, включая C#, генерируют вызов конструктора базового класса автоматически, поэтому вам, как правило, об этом можно не беспокоиться. В конечном счете всегда вызывается открытый конструктор объекта `System.Object` без параметров. Этот конструктор ничего не делает — просто возвращает управление по той простой причине, что в `System.Object` не определено никаких экземплярных полей данных, поэтому конструктору просто нечего делать.

В редких ситуациях экземпляр типа может создаваться без вызова конструктора экземпляров. В частности, метод `MemberwiseClone` объекта `Object` выделяет память, инициализирует служебные поля объекта, а затем копирует байты исходного объекта в область памяти, выделенную для нового объекта. Кроме того, конструктор обычно не вызывается при десериализации объекта. Код десериализации выделяет память для объекта без вызова конструктора, используя метод `GetUninitializedObject` или `GetSafeUninitializedObject` типа `System.Runtime.Serialization.FormatterServices` (см. главу 24).

ВНИМАНИЕ

Нельзя вызывать какие-либо виртуальные методы конструктора, которые могут повлиять на создаваемый объект. Причина проста: если вызываемый виртуальный метод переопределен в типе, экземпляр которого создается, происходит реализация производного типа, но к этому моменту еще не завершилась инициализация всех полей в иерархии. В таких обстоятельствах последствия вызова виртуального метода непредсказуемы.

C# предлагает простой синтаксис, позволяющий инициализировать поля во время создания объекта ссылочного типа:

```
internal sealed class SomeType {  
    private Int32 m_x = 5;  
}
```

При создании объекта `SomeType` его поле `m_x` инициализируется значением 5. Вы можете спросить: как это происходит? Изучив IL-код метода-конструктора

этого объекта (этот метод также фигурирует под именем `.ctor`), вы увидите следующий код:

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size 14 (0xe)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldc.i4.5
    IL_0002: stfld int32 SomeType::m_x
    IL_0007: ldarg.0
    IL_0008: call instance void [mscorlib]System.Object::.ctor()
    IL_000d: ret
} // end of method SomeType::.ctor
```

Как видите, конструктор объекта `SomeType` содержит код, записывающий в поле `m_x` значение 5 и вызывающий конструктор базового класса. Иначе говоря, компилятор C# предлагает удобный синтаксис, позволяющий инициализировать поля экземпляра при их объявлении. Компилятор транслирует этот синтаксис в метод-конструктор, выполняющий инициализацию. Это значит, что нужно быть готовым к разрастанию кода, как это показано на следующем примере:

```
internal sealed class SomeType {
    private Int32 m_x = 5;
    private String m_s = "Hi there";
    private Double m_d = 3.14159;
    private Byte m_b;

    // Это конструкторы
    public SomeType() { ... }
    public SomeType(Int32 x) { ... }
    public SomeType(String s) { ...; m_d = 10; }
}
```

ПРИМЕЧАНИЕ

Компилятор инициализирует все поля при помощи соответствующего синтаксиса перед вызовом конструктора базового класса для поддержания представления о том, что все поля имеют корректные значения, обозначенные в исходном коде. Потенциальная проблема может возникнуть в тех случаях, когда конструктор базового класса вызывает виртуальный метод, осуществляющий обратный вызов в метод, определенный в порожденном классе. В этом случае поля инициализируются при помощи соответствующего синтаксиса перед вызовом виртуального метода.

Генерируя IL-код для трех методов-конструкторов из этого примера, компилятор помещает в начало каждого из методов код, инициализирующий

поля `m_x`, `m_s` и `m_d`. Затем он добавляет к методу код, расположенный внутри методов-конструкторов. Например, IL-код, сгенерированный для конструктора с параметром типа `String`, состоит из кода, инициализирующего поля `m_x`, `m_s` и `m_d`, и кода, перезаписывающего поле `m_d` значением 10. Заметьте: поле `m_b` гарантированно инициализируется значением 0, даже если нет кода, инициализирующего это поле явно.

Поскольку в показанном ранее классе определены три конструктора, компилятор трижды генерирует код, инициализирующий поля `m_x`, `m_s` и `m_d`: по одному разу для каждого из конструкторов. Если имеется несколько инициализируемых экземплярных полей и множество перегруженных методов-конструкторов, стоит подумать о том, чтобы определить поля, не инициализируя их; создать единственный конструктор, выполняющий общую инициализацию, и заставить каждый метод-конструктор явно вызывать конструктор, выполняющий общую инициализацию. Этот подход позволит уменьшить размер генерируемого кода. Следующий пример иллюстрирует использование способности C# явно заставлять один конструктор вызывать другой конструктор посредством зарезервированного слова `this`:

```
internal sealed class SomeType {  
    // Здесь нет кода, явно инициализирующего поля  
    private Int32 m_x;  
    private String m_s;  
    private Double m_d;  
    private Byte m_b;  
  
    // Код этого конструктора инициализирует поля значениями по умолчанию  
    // Этот конструктор должен вызываться всеми остальными конструкторами  
    public SomeType() {  
        m_x = 5;  
        m_s = "Hi there";  
        m_d = 3.14159;  
        m_b = 0xff;  
    }  
  
    // Этот конструктор инициализирует поля значениями по умолчанию,  
    // а затем изменяет значение m_x  
    public SomeType(Int32 x) : this() {  
        m_x = x;  
    }  
  
    // Этот конструктор инициализирует поля значениями по умолчанию,  
    // а затем изменяет значение m_s  
    public SomeType(String s) : this() {  
        m_s = s;  
    }  
}
```

```
// Этот конструктор инициализирует поля значениями по умолчанию,  
// а затем изменяет значения m_x и m_s  
public SomeType(Int32 x, String s) : this() {  
    m_x = x;  
    m_s = s;  
}  
}
```

Конструкторы экземпляров и структуры (значимые типы)

Конструкторы значимых типов (struct) работают иначе, чем ссылочных (class). CLR всегда разрешает создание экземпляров значимых типов и этому ничто не может помешать. Поэтому, по большому счету, конструкторы у значимого типа можно не определять. Фактически многие компиляторы (включая C#) не определяют для значимых типов конструкторы по умолчанию, не имеющие параметров. Разберем следующий код:

```
internal struct Point {  
    public Int32 m_x, m_y;  
}  
internal sealed class Rectangle {  
    public Point m_topLeft, m_bottomRight;  
}
```

Для того чтобы создать объект Rectangle, надо использовать оператор new, указав конструктор. В этом случае вызывается конструктор, автоматически сгенерированный компилятором C#. Память, выделенная для объекта Rectangle, включает место для двух экземпляров значимого типа Point. Из соображений повышения производительности CLR не пытается вызвать конструктор для каждого экземпляра значимого типа, содержащегося в объекте ссылочного типа. Однако, как отмечалось ранее, поля значимого типа инициализируются нулевыми или пустыми значениями.

CLR действительно позволяет программистам определять конструкторы для значимых типов, но эти конструкторы выполняются лишь при наличии кода, явно вызывающего один из них, например, как в конструкторе объекта Rectangle:

```
internal struct Point {  
    public Int32 m_x, m_y;  
  
    public Point(Int32 x, Int32 y) {  
        m_x = x;  
        m_y = y;  
    }  
}
```

```
internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;

    public Rectangle() {
        // В C# оператор new, использованный для создания экземпляра значимого
        // типа, вызывает конструктор для инициализации полей значимого типа
        m_topLeft = new Point(1, 2);
        m_bottomRight = new Point(100, 200);
    }
}
```

Конструктор экземпляра значимого типа выполняется, только если вызвать его явно. Так что если конструктор объекта `Rectangle` не инициализировал его поля `m_topLeft` и `m_bottomRight` вызовом с помощью оператора `new` конструктора `Point`, поля `m_x` и `m_y` у обеих структур `Point` будут содержать 0.

Если значимый тип `Point` уже определен, то конструктор по умолчанию, не имеющий параметров, не определяется. Однако давайте перепишем наш код:

```
internal struct Point {
    public Int32 m_x, m_y;

    public Point() {
        m_x = m_y = 5;
    }
}

internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;

    public Rectangle() {
    }
}
```

А теперь ответьте, какими значениями — 0 или 5 — будут инициализированы поля `m_x` и `m_y`, принадлежащие структурам `Point` (`m_topLeft` и `m_bottomRight`)? Предупреждаю, вопрос с подвохом.

Многие разработчики (особенно с опытом программирования на C++) решат, что компилятор C# поместит в конструктор `Rectangle` код, автоматически вызывающий конструктор структуры `Point` по умолчанию, не имеющий параметров, для двух полей `Rectangle`. Однако чтобы повысить быстродействие приложения во время выполнения, компилятор C# не сгенерирует такой код автоматически. Фактически большинство компиляторов никогда не генерирует автоматически код для вызова конструктора по умолчанию для значимого типа даже при наличии конструктора без параметров. Чтобы принудительно исполнить конструктор значимого типа без параметров, разработчик должен добавить код его явного вызова.

С учетом сказанного можно ожидать, что поля `m_x` и `m_y` обеих структур `Point` из объекта `Rectangle` в показанном коде будут инициализированы нуле-

выми значениями, так как в этой программе нет явного вызова конструктора `Point`.

Однако, как я отмечал, мой первый вопрос был с подвохом. Подвох в том, что `C#` не позволяет определять для значимого типа конструкторы без параметров. Поэтому показанный код на самом деле даже не компилируется. При попытке скомпилировать его компилятор `C#` генерирует сообщение об ошибке (ошибка `CS0568`: структура не может содержать явные конструкторы без параметров):

```
error CS0568: Structs cannot contain explicit parameterless constructors
```

`C#` преднамеренно запрещает определять конструкторы без параметров у значимых типов, чтобы не вводить разработчиков в заблуждение относительно того, какой конструктор вызывается. Если конструктор определить нельзя, компилятор никогда не будет автоматически генерировать код, вызывающий такой конструктор. В отсутствие конструктора без параметров поля значимого типа всегда инициализируются нулевыми или пустыми значениями.

ПРИМЕЧАНИЕ

Строго говоря, в поля значимого типа обязательно заносятся значения `0` или `null`, если значимый тип вложен в объект ссылочного типа. Однако гарантии, что поля значимых типов, работающие со стеком, будут инициализированы значениями `0` или `null`, нет. Чтобы код был верифицируемым, перед чтением любого поля значимого типа, работающего со стеком, нужно записать в него значение. Если код сможет прочитать значение поля значимого типа до того, как туда будет записано какое-то значение, может нарушиться безопасность. `C#` и другие компиляторы, генерирующие верифицируемый код, гарантируют, что поля любых значимых типов, работающие со стеком, перед чтением обнуляются или хотя бы в них записываются некоторые значения. Поэтому при верификации во время выполнения исключение выброшено не будет. Однако обычно можно предполагать, что поля значимых типов инициализируются нулевыми значениями и все сказанное в этом примечании можно полностью игнорировать.

Хотя `C#` не допускает использования значимых типов с конструкторами без параметров, это допускает CLR. Так что если вас не беспокоят упомянутые скрытые особенности работы системы, можно на другом языке (например, на `IL`) определить собственный значимый тип с конструктором без параметров.

Поскольку `C#` не допускает использования значимых типов с конструкторами без параметров, при компиляции следующего типа компилятор сообщает об ошибке: (ошибка `CS0573`: '`SomeValType.m_x`': нельзя создавать конструкторы экземплярных полей в структурах):

```
error CS0573: 'SomeValType.m_x': cannot have instance field initializers in structs
```

А вот как выглядит код, вызвавший эту ошибку:

```
internal struct SomeValType {  
    // В значимый тип нельзя подставлять инициализацию экземплярных полей  
    private Int32 m_x = 5;  
}
```

Кроме того, поскольку верифицируемый код перед чтением любого поля значимого типа требует записывать в него какое-либо значение, любой конструктор, определенный для значимого типа, должен инициализировать все поля этого типа. Следующий тип определяет конструктор для значимого типа, но не может инициализировать все его поля:

```
internal struct SomeValType {  
    private Int32 m_x, m_y;  
  
    // C# допускает наличие у значимых типов конструкторов с параметрами  
    public SomeValType(Int32 x) {  
        m_x = x;  
        // Обратите внимание: поле m_y здесь не инициализируется  
    }  
}
```

При компиляции этого типа компилятор C# генерирует сообщение об ошибке: (ошибка CS0171: поле 'SomeValType.m_y' должно быть полностью определено до возвращения управления конструктором):

```
error CS0171: Field 'SomeValType.m_y' must be fully assigned before control  
leaves the constructor
```

Чтобы разрешить проблему, конструктор должен ввести значение (обычно 0) в поле y.

В качестве альтернативного варианта вы можете инициализировать все поля значимого типа, как это сделано здесь:

```
// C# позволяет значимым типам иметь конструкторы с параметрами  
public SomeValType(Int32 x) {  
    // Выглядит необычно, но компилируется прекрасно  
    // и все поля инициализируются значениями 0 или null  
    this = new SomeValType();  
  
    m_x = x; // Присваивает m_x значение x  
    // Обратите внимание, что поле m_y было инициализировано нулем  
}
```

В конструкторе значимого типа `this` представляет экземпляр значимого типа и ему можно приписать значение нового экземпляра значимого типа, у которого все поля инициализированы нулями. В конструкторах ссылочного типа указатель `this` является неизменяемым.

Конструкторы типов

Помимо конструкторов экземпляров, CLR поддерживает конструкторы типов (также известные как *статические конструкторы*, *конструкторы классов* и *инициализаторы типов*). Конструкторы типов можно применять и к интерфейсам (хотя C# этого не допускает), ссылочным и значимым типам. Подобно тому, как конструкторы экземпляров используются для установки первоначального состояния экземпляра типа, конструкторы типов служат для установки первоначального состояния типа. По умолчанию у типа не определено конструктора. У типа может быть один и только один конструктор. Кроме того, у конструкторов типов никогда не бывает параметров. Вот как определяются ссылочные и значимые типы с конструкторами в программах на C#:

```
internal sealed class SomeRefType {
    static SomeRefType() {
        // Исполняется при первом обращении к ссылочному типу SomeRefType
    }
}

internal struct SomeValType {
    // C# на самом деле допускает определять для значимых типов
    // конструкторы без параметров
    static SomeValType() {
        // Исполняется при первом обращении к значимому типу SomeValType
    }
}
```

Обратите внимание, что конструкторы типов определяют так же, как конструкторы экземпляров без параметров за исключением того, что их помечают как статические. Кроме того, конструкторы типов всегда должны быть закрытыми (C# делает их закрытыми автоматически). Однако если явно пометить в исходном тексте программы конструктор типа как закрытый (или как-то иначе), компилятор C# выведет сообщение об ошибке: (ошибка CS0515: 'SomeValType.SomeValType()': в статических конструкторах нельзя использовать модификаторы доступа):

```
error CS0515: 'SomeValType.SomeValType()': access modifiers are not allowed on static constructors
```

Конструкторы типов всегда должны быть закрытыми, чтобы код разработчика не смог их вызвать, напротив, в то же время среда CLR всегда способна вызвать конструктор типа.

ВНИМАНИЕ

Хотя конструктор типа можно определить в значимом типе, этого никогда не следует делать, так как иногда CLR не вызывает статический конструктор значимого типа.

Например:

```
internal struct SomeValType {
    static SomeValTypeQ {
        Console.WriteLine("This never gets displayed");
    }
    public Int32 m_x;
}

public sealed class Program {
    public static void Main() {
        SomeValType[] a = new SomeValType[10];
        a[0].m_x = 123;
        Console.WriteLine(a[0].m_x); // Выводим 123
    }
}
```

Есть определенные особенности вызова конструктора типа. При компиляции метода JIT-компилятор обнаруживает типы, на которые есть ссылки из кода. Если в каком-либо из типов определен конструктор, JIT-компилятор проверяет, был ли исполнен конструктор типа в данном домене приложений. Если нет, JIT-компилятор создает в IL-коде вызов конструктора типа. Если же код уже исполнялся, JIT-компилятор вызова конструктора типа не создает, так как «знает», что тип уже инициализирован (пример подобного поведения см. в разделе «Производительность конструкторов типов»).

Затем, после JIT-компиляции метода, начинается выполнение потока, и в конечном итоге очередь доходит до кода вызова конструктора. В реальности может оказаться, что несколько потоков одновременно начнут выполнять метод. CLR стремится обеспечить, чтобы конструктор типа выполнялся только раз в каждом домене приложений. Для этого при вызове конструктора типа вызывающий поток в рамках синхронизации потоков получает возможность взаимноисключающего записывания. Это означает, что если несколько потоков одновременно попытаются вызывать конструктор типа, только один получит такую возможность, остальные блокируются. Первый поток выполнит код статического конструктора. После выхода из конструктора первого потока «проснутся» простаивающие потоки и проверят, был ли выполнен конструктор. Они не станут снова выполнять код, а просто обеспечат возврат управления из метода конструктора. Кроме того, при последующем вызове какого-либо из этих методов CLR будет «в курсе», что конструктор типа уже выполнялся, и предотвратит его повторное выполнение.

ПРИМЕЧАНИЕ

Поскольку CLR гарантирует, что конструктор типа выполняется только однажды в каждом домене приложений, а также обеспечивает его безопасность по отношению к потокам, конструктор типа лучше всего подходит для инициализации всех объектов-одиночек (singleton), необходимых для существования типа.

В рамках одного потока возможна неприятная ситуация, когда существуют два конструктора типа, содержащих перекрестно ссылающийся код. Например, конструктор типа `ClassA` содержит код, ссылающийся на `ClassB`, а последний содержит конструктор типа, ссылающийся на `ClassA`. Даже в таких условиях CLR заботится, чтобы код конструкторов типов выполнялся лишь однажды, но исполняющая среда не в состоянии обеспечить завершение исполнения конструктора типа `ClassA` до начала исполнения конструктора типа `ClassB`. При написании кода следует избегать подобных ситуаций. В действительности, поскольку за вызов конструкторов типов отвечает CLR, не нужно писать код, который требует вызова конструкторов типов в определенном порядке.

Наконец, если конструктор типа генерирует необрабатываемое исключение, CLR считает такой тип непригодным. При попытке обращения к любому полю или методу такого типа вбрасывается исключение `System.TypeInitializationException`.

Код конструктора типа может обращаться только к статическим полям типа, обычно это делается, чтобы их инициализировать. Как и в случае экземплярных полей, C# предлагает простой синтаксис:

```
internal sealed class SomeType {
    private static Int32 s_x = 5;
}
```

ПРИМЕЧАНИЕ

C# не позволяет в значимых типах использовать встроенный синтаксис инициализации полей, но разрешает это в статических полях. Иначе говоря, если в приведенном ранее коде заменить тип `class` типом `struct`, код прекрасно скомпилируется и будет работать, как задумано.

При компоновке этого кода компилятор автоматически генерирует конструктор типа `SomeType`. Иначе говоря, получается тот же эффект, как если бы этот код был написан следующим образом:

```
internal sealed class SomeType {
    private static Int32 s_x;
    static SomeType() { s_x = 5; }
}
```

При помощи утилиты `ILDasm.exe` нетрудно проверить, какой код на самом деле сгенерировал компилятор. Для этого нужно изучить IL-код конструктора типа. В таблице определений методов, составляющей метаданные модуля, метод-конструктор типа всегда называется `.cctor` (от *class constructor*).

Из представленного далее IL-кода видно, что метод `.cctor` является закрытым и статическим. Заметьте также, что код этого метода действительно записывает в статическое поле `s_x` значение 5.

```
.method private hidebysig specialname rtspecialname static
void .cctor() cil managed
{
```

```
// Code size 7 (0x7)
.maxstack 8
IL_0000: ldc.i4.5
IL_0001: stsfld int32 SomeType::s_x
IL_0006: ret
} // end of method SomeType::.cctor
```

Конструктор типа не должен вызывать конструктор базового класса. Этот вызов не обязателен, так как ни одно статическое поле типа не используется совместно с базовым типом и не наследуется от него.

ПРИМЕЧАНИЕ

В ряде языков, таких как Java, ожидается, что при обращении к типу будет вызван его конструктор, а также конструкторы всех его базовых типов. Кроме того, интерфейсы, реализованные этими типами, тоже должны вызывать свои конструкторы. CLR не поддерживает такую семантику, но позволяет компиляторам и разработчикам предоставлять поддержку подобной семантики через метод `RunClassConstructor`, предлагаемый типом `System.Runtime.CompilerServices.RuntimeHelpers`. Компилятор любого языка, требующего подобную семантику, генерирует в конструкторе типа код, вызывающий этот метод для всех базовых типов. При использовании метода `RunClassConstructor` для вызова конструктора типа CLR определяет, был ли он исполнен ранее, и если да, то не вызывает его снова.

В завершение этого раздела рассмотрим следующий код:

```
internal sealed class SomeType {
    private static Int32 s_x = 5;

    static SomeTypeQ {
        s_x = 10;
    }
}
```

Здесь компилятор C# генерирует единственный метод-конструктор типа, который сначала инициализирует поле `s_x` значением 5, затем — значением 10. Иначе говоря, при генерации IL-кода конструктора типа компилятор C# сначала генерирует код, инициализирующий статические поля, затем обрабатывает явный код, содержащийся внутри метода-конструктора типа.

ВНИМАНИЕ

Иногда разработчики спрашивают меня: можно ли исполнить код во время загрузки типа? Во-первых, следует знать, что типы выгружаются только при закрытии домена приложений. Когда домен приложений закрывается, объект, идентифицирующий тип, становится недоступным, и сборщик мусора освобождает занятую им память. Многим разработчикам такой сценарий дает основание полагать, что можно добавить к типу статический метод `Finalize`, автоматически вызываемый при выгрузке типа. Увы, CLR не под-

держивает статические методы `Finalize`. Однако не все потеряно: если при закрытии домена приложений нужно исполнить некоторый код, можно зарегистрировать метод обратного вызова для события `DomainUnload` типа `System.AppDomain`.

Производительность конструкторов типов

В предыдущем разделе я говорил о сложностях вызова конструкторов типов и рассказал, как JIT-компилятор принимает решение о том, нужно ли создавать IL-код конструктора, а CLR обеспечивает безопасный с точки зрения потоков вызов конструктора. Однако это всего лишь начало проблем — есть еще нюансы, связанные с производительностью.

При компиляции метода JIT-компилятор самостоятельно решает, нужно ли создавать вызов для исполнения конструктора типа. Если принимается решение создать вызов, нужно еще решить, где его разместить в IL-коде. Есть две возможности.

- ❑ JIT-компилятор может вставить вызов непосредственно перед кодом, который создает первый экземпляр типа или обращается к ненаследуемому полю или члену класса. Это называют *точной* (precise) семантикой, так как CLR вызовет конструктор типа ровно тогда, когда будет нужно.
- ❑ JIT-компилятор может создать вызов в месте, предшествующем коду, обращаемому к ненаследуемому статическому полю. Это называют семантикой вызова *до инициации поля* (before-field-init), потому что в этом случае CLR обеспечивает выполнение статического конструктора заранее, до обращения к статическому полю, то есть конструктор может выполняться значительно раньше.

Семантика вызова до инициации поля предпочтительнее, так как предоставляет CLR свободу выбора времени вызова конструктора типа, что позволяет CLR по мере возможности создавать более эффективный код. Например, CLR может менять время вызова конструктора типа в зависимости от вида загрузки типа — в домен приложений или вне доменов приложений, а также от типа кода, — созданного JIT-компилятором или утилитой `NGen.exe`.

По умолчанию компиляторы языков программирования выбирают наиболее подходящую для определяемого типа семантику и информируют CLR о своем выборе, определяя флаг `BeforeFieldInit` в одной из строк определения в таблице определений, входящих в метаданные. Здесь я расскажу, как эта задача решается компилятором C# и как это влияет на производительность. Начнем с изучения следующего кода:

```
using System;
using System.Diagnostics;
```

////////////////////////////////////

```
// Так как в этом классе конструктор типа не задан явно,
// C# отмечает определение типа в метаданных ключевым словом BeforeFieldInit
internal sealed class BeforeFieldInit {
    public static Int32 s_x = 123;
}

// Так как в этом классе конструктор типа задан явно,
// C# не отмечает определение типа ключевым словом BeforeFieldInit
internal sealed class Precise {
    public static Int32 s_x;
    static Precise() { s_x = 123; }
}

////////////////////////////////////

public sealed class Program {
    public static void Main() {
        const Int32 iterations = 1000 * 1000 * 1000;
        PerfTest1(iterations);
        PerfTest2(iterations);
    }

    // При JIT-компиляции этого метода конструкторы типов для классов
    // BeforeFieldInit и Precise еще НЕ ВЫПОЛНЕНЫ, поэтому вызовы
    // этих конструкторов встроены в код метода,
    // что снижает эффективность программы
    private static void PerfTest1(Int32 iterations) {
        Stopwatch sw = Stopwatch.StartNew();
        for (Int32 x = 0; x < iterations; x++) {
            // JIT-компилятор создает код вызова конструктора типа
            // BeforeFieldInit, чтобы он выполнялся до начала цикла
            BeforeFieldInit.s_x = 1;
        }
        Console.WriteLine("PerfTest1: {0} BeforeFieldInit", sw.Elapsed);

        sw = Stopwatch.StartNew();
        for (Int32 x = 0; x < iterations; x++) {
            // JIT-компилятор создает код вызова конструктора типа Precise,
            // чтобы тот проверил, нужно ли вызывать конструктор в каждом цикле
            Precise.s_x = 1;
        }
        Console.WriteLine("PerfTest1: {0} Precise", sw.Elapsed);
    }

    // При JIT-компиляции этого метода конструкторы типов
    // для классов BeforeFieldInit и Precise уже завершили работу,
    // поэтому вызовы этих конструкторов НЕ ВСТРАИВАЮТСЯ

```

```
// в код метода, благодаря чему он выполняется быстрее
private static void PerfTest2(Int32 iterations) {
    Stopwatch sw = Stopwatch.StartNew();
    for (Int32 x = 0; x < iterations; x++) {
        BeforeFieldInit.s_x = 1;
    }
    Console.WriteLine("PerfTest2: {0} BeforeFieldInit", sw.Elapsed);

    sw = Stopwatch.StartNew();
    for (Int32 x = 0; x < iterations; x++) {
        Precise.s_x = 1;
    }
    Console.WriteLine("PerfTest2: {0} Precise", sw.Elapsed);
}
}
/////////////////////// Конец файла /////////////////////////
```

После компоновки и выполнения этого кода я получил следующий результат:

```
PerfTest1: 00:00:02.1997770 BeforeFieldInit
PerfTest1: 00:00:07.6188948 Precise
PerfTest2: 00:00:02.0843565 BeforeFieldInit
PerfTest2: 00:00:02.0843732 Precise
```

Обнаружив в коде класс со статическими полями, где имеет место встроенная инициализация (класс `BeforeFieldInit`), компилятор C# создает в таблице определений класса запись с флагом метаданных `BeforeFieldInit`. А для класса с явно заданным конструктором типа (класс `Precise`) компилятор C# создает в таблице определений класса запись без такого флага. Логика создателей этого алгоритма проста: статические поля должны инициализироваться до обращения к ним, а явно заданный конструктор типа может содержать дополнительный код, способный делать определенную видимую работу, поэтому его нужно выполнять в заданное разработчиком время.

Как видно из выходных данных программы, решение значительно сказывается на производительности кода. В `PerfTest1` цикл выполняется за 1,96 секунды, что сильно отличается от последней строки — целых 6,24 секунды, то есть в три раза быстрее. Разброс времени выполнения `PerfTest2` намного уже, так как JIT-компилятор «знал», что конструкторы типов уже вызывались, поэтому изъять из IL-кода вызовы методов конструкторов типов.

Было бы разумно предоставить разработчикам возможность явно задавать флаг `BeforeFieldInit` в коде, не поручая принятие этого решения компилятору на основе того, как создается конструктор типа — явно или неявно. Так разработчики получили бы дополнительный прямой рычаг управления производительностью и семантикой создаваемого кода.

Методы перегруженных операторов

В некоторых языках тип может определять, как операторы должны манипулировать его экземплярами. В частности, многие типы (например, `System.String`) используют перегрузку операторов равенства (`==`) и неравенства (`!=`). CLR ничего не известно о перегрузке операторов — ведь среда даже не знает, что такое оператор. Смысл операторов и код, который должен быть сгенерирован, когда тот или иной оператор встретится в исходном тексте, определяется языком программирования.

Например, если в программе на C# поставить между обычными числами оператор `+`, компилятор генерирует код, выполняющий сложение двух чисел. Когда оператор `+` применяют к строкам, компилятор C# генерирует код, выполняющий конкатенацию этих строк. Для обозначения неравенства в C# используется оператор `!=`, а в Visual Basic — оператор `<>`. Наконец, оператор `^` в C# задает операцию «исключающее или» (XOR), тогда как в Visual Basic это возведение в степень.

Хотя CLR находится в неведении относительно операторов, среда не регламентирует, как языки программирования должны поддерживать перегрузку операторов. Смысл в том, чтобы без труда использовать перегрузку при написании кода на разных языках. В случае каждого конкретного языка принимается отдельное решение, будет ли этот язык поддерживать перегрузку операторов и, если да, какой синтаксис задействовать для представления и использования перегруженных операторов. С точки зрения CLR перегруженные операторы представляют собой просто методы.

От выбора языка зависит наличие поддержки перегруженных операторов и их синтаксис, а при компиляции исходного текста компилятор генерирует метод, определяющий работу оператора. Спецификация CLR требует, чтобы перегруженные операторные методы были открытыми и статическими. Дополнительно C# (и многие другие языки) требует, чтобы у операторного метода тип, по крайней мере, одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод. Причина этого ограничения в том, что оно позволяет компилятору C# в разумное время находить кандидатуры операторных методов для привязки.

Пример метода перегруженного оператора, заданного в определении класса C#:

```
public sealed class Complex {  
    public static Complex operator+(Complex c1, Complex c2) { ... }  
}
```

Компилятор генерирует определение метода `op_Addition` и устанавливает в записи с определением этого метода флаг `specialname`, свидетельствующий о том, что это «особый» метод. Когда компилятор языка (в том числе компи-

лятор C#) видит в исходном тексте оператор +, он исследует типы его операндов. При этом компилятор пытается выяснить, не определен ли для одного из них метод op_Addition с флагом specialname, параметры которого совместимы с типами операндов. Если такой метод существует, компилятор генерирует код, вызывающий этот метод, иначе возникает ошибка компиляции.

В табл. 8.1 и 8.2 приведен набор унарных и бинарных операторов, которые C# позволяет перегружать, их обозначения и рекомендованные имена соответствующих методов, которые должен генерировать компилятор. Третий столбец я прокомментирую в следующем разделе.

Таблица 8.1. Унарные операторы C# и CLS-совместимые имена соответствующих методов

Оператор C#	Имя специального метода	Рекомендуемое CLS-совместимое имя метода
+	op UnaryPlus	Plus
−	op UnaryNegation	Negate
!	op LogicalNot	Not
~	op OnesComplement	OnesComplement
++	op Increment	Increment
--	op Decrement	Decrement
Her	op True	IsTrue { get; }
Her	op False	IsFalse { get; }

Таблица 8.2. Бинарные операторы и их CLS-совместимые имена методов

Оператор C#	Имя специального метода	Рекомендуемое CLS-совместимое имя метода
+	op_Addition	Add
−	op_Subtraction	Subtract
*	op_Multiply	Multiply
/	op_Division	Divide
%	op_Modulus	Mod
&	op_BitwiseAnd	BitwiseAnd
	op_BitwiseOr	BitwiseOr
^	op_ExclusiveOr	Xor
<<	op_LeftShift	LeftShift
>>	op_RightShift	RightShift
==	op_Equality	Equals

Оператор C#	Имя специального метода	Рекомендуемое CLS-совместимое имя метода
!=	op_Inequality	Compare
<	op_LessThan	Compare
>	op_GreaterThan	Compare
<=	op_LessThanOrEqual	Compare
>=	op_GreaterThanOrEqual	Compare

В спецификации CLR определены многие другие операторы, поддающиеся перегрузке, но C# их не поддерживает. Они не очень распространены, поэтому я их здесь не указал. Полный список есть в спецификации ECMA (www.ecma-international.org/publications/standards/Ecma-335.htm) общезыковой инфраструктуры CLI, разделы 10.3.1 (унарные операторы) и 10.3.2 (бинарные операторы).

ПРИМЕЧАНИЕ

Если изучить фундаментальные типы библиотеки классов .NET Framework (FCL) — `Int32`, `Int64`, `UInt32` и т. д., — можно заметить, что они не определяют методы перегруженных операторов. Дело в том, что CLR поддерживает IL-команды, позволяющие манипулировать экземплярами этих типов. Если бы эти типы поддерживали соответствующие методы, а компиляторы генерировали вызывающий их код, то каждый такой вызов снижал бы быстродействие во время выполнения. Кроме того, чтобы реализовать ожидаемое действие, такой метод все равно исполнял бы те же инструкции языка IL. Для вас это означает следующее: если язык, на котором вы пишете, не поддерживает какой-либо из фундаментальных типов FCL, вы не сможете выполнять действия над экземплярами этого типа.

Операторы и взаимодействие языков программирования

Перегрузка операторов очень полезна, поскольку позволяет разработчикам лаконично выражать свои мысли в компактном коде. Однако не все языки поддерживают перегрузку операторов, например при использовании языка, не поддерживающего перегрузку, он не будет знать, как интерпретировать оператор `+` (если только соответствующий тип не является элементарным в этом языке), и компилятор сгенерирует ошибку. При использовании языков, не поддерживающих перегрузку, язык должен позволять вызывать методы с приставкой `op_` (например, `op_Addition`) напрямую.

Если вы пишете на языке, не поддерживающем перегрузку оператора `+` путем определения в типе, наверняка этот тип может предоставить вам метод `op_Addition`. Логично ожидать, что в C# тоже можно вызвать метод `op_Addition`, указав оператор `+`, но это не так. Обнаружив оператор `+`, компилятор C# ищет

метод `op_Addition` с флагом метаданных `specialname`, который информирует компилятор, что `op_Addition` — это перегруженный операторный метод. А поскольку метод `op_Addition` создан на языке, не поддерживающем перегрузку, в методе флага `specialname` не будет, и компилятор C# вернет ошибку. Ясно, что код любого языка может явно вызывать метод по имени `op_Addition`, но компиляторы не преобразуют оператор `+` в вызов этого метода.

Особое мнение автора о правилах Microsoft, связанных с именами методов операторов

Я уверен, что все эти правила, касающиеся случаев, когда можно или нельзя вызвать метод перегруженного оператора, излишне сложны. Если бы компиляторы, поддерживающие перегрузку операторов, просто не генерировали флаг метаданных `specialname`, можно было бы заметно упростить эти правила, и программистам стало бы намного легче работать с типами, поддерживающими методы перегруженных операторов. Если бы языки, поддерживающие перегрузку операторов, поддерживали бы и синтаксис операторов, все языки также поддерживали бы явный вызов методов с приставкой `op_`. Я не могу назвать ни одной причины, заставившей Microsoft так усложнить эти правила, и надеюсь, что в следующих версиях своих компиляторов Microsoft упростит их.

Для типа с методами перегруженных операторов Microsoft также рекомендует определять открытые экземплярные методы с дружественными именами. В коде этих методов содержатся вызовы реальных методов перегруженных операторов. Например, тип с перегруженными методами `op_Addition` или `op_AdditionAssignment` должен также определять открытый метод с дружественным именем `Add`. Список рекомендованных дружественных имен для всех методов операторов приводится в третьем столбце табл. 8.1 и 8.2. Таким образом, показанный ранее тип `Complex` можно было бы определить и так:

```
public sealed class Complex {
    public static Complex operator+(Complex c1, Complex c2) { ... }
    public static Complex Add(Complex c1, Complex c2) { return(c1 + c2); }
}
```

Ясно, что код, написанный на любом языке, способен вызывать любой из операторных методов по его дружественному имени, скажем `Add`. Правила же Microsoft, предписывающие дополнительно определять методы с дружественными именами, лишь осложняют ситуацию. Думаю, это излишняя сложность, к тому же вызов методов с дружественными именами вызовет снижение быстродействия, если только JIT-компилятор не будет способен подставлять код в метод с дружественным именем. Подстановка кода позволит JIT-компилятору оптимизировать весь код путем удаления дополнительного вызова метода и тем самым повысить скорость выполнения.

ПРИМЕЧАНИЕ

Примером типа, в котором перегружаются операторы и используются дружественные имена методов в соответствии с правилами Microsoft, может служить класс `System.Decimal` библиотеки FCL.

Методы операторов преобразования

Время от времени требуется преобразовывать объект одного типа в объект другого типа. Уверен, что вам приходилось преобразовывать значение `Byte` в `Int32`. Когда исходный и целевой типы являются элементарными, компилятор способен без посторонней помощи генерировать код, необходимый для преобразования объекта. Однако если ни один из типов не является элементарным, компилятор порождает код, заставляющий CLR выполнить преобразование (приведение типов). В этом случае CLR просто проверяет, является ли тип исходного объекта таким же, как целевой тип (или производный от целевого). Однако иногда требуется преобразовать объект одного типа в совершенно другой тип. Представьте, что в FCL есть тип данных `Rational`, в который удобно преобразовывать объекты типа `Int32` или `Single`. Более того, обратное преобразование выполнять тоже удобно.

Чтобы выполнить эти преобразования, в типе `Rational` должны определяться открытые конструкторы, принимающие в качестве единственного параметра экземпляр преобразуемого типа. Кроме того, нужно определить открытый экземплярный метод `ToXxx`, не принимающий параметров (примером может служить популярный метод `ToString`). Каждый такой метод преобразует экземпляр типа, в котором определен этот метод, в экземпляр типа `Xxx`. Вот как правильно определить соответствующие конструкторы и методы для типа `Rational`:

```
public sealed class Rational {  
    // Создает Rational из Int32  
    public Rational(Int32 num) { ... }  
  
    // Создает Rational из Single  
    public Rational(Single num) { ... }  
  
    // Преобразует Rational в Int32  
    public Int32 ToInt32() { ... }  
  
    // Преобразует Rational в Single  
    public Single ToSingle() { ... }  
}
```

Вызывая эти конструкторы и методы, разработчик, используя любой язык, может преобразовать объект типа `Int32` или `Single` в `Rational` и обратно. Подобные преобразования могут быть довольно удобны, и при разработке типа

стоит подумать, какие конструкторы и методы преобразования имело бы смысл включить в разрабатываемый тип.

Ранее мы обсуждали способы поддержки перегрузки операторов в разных языках. Некоторые (например, C#) наряду с этим поддерживают перегрузку *операторов преобразования* — методы, преобразующие объекты одного типа в объекты другого типа. Методы операторов преобразования определяются при помощи специального синтаксиса. Спецификация CLR требует, чтобы перегруженные методы преобразования были открытыми и статическими. Кроме того, C# (и многие другие языки) требуют, чтобы у метода преобразования тип, по крайней мере, одного из параметров или возвращаемого значения совпадал с типом, в котором определен операторный метод. Причина этого ограничения в том, что оно позволяет компилятору C# в разумное время находить кандидатуры операторных методов для привязки. Следующий код добавляет в тип `Rational` четыре метода операторов преобразования:

```
public sealed class Rational {
    // Создает Rational из Int32
    public Rational(Int32 num) { ... }

    // Создает Rational из Single
    public Rational(Single num) { ... }

    // Преобразует Rational в Int32
    public Int32 ToInt32() { ... }
    // Преобразует Rational в Single
    public Single ToSingle() { ... }

    // неявно создает Rational из Int32 и возвращает полученный объект
    public static implicit operator Rational(Int32 num) {
        return new Rational(num); }

    // неявно создает Rational из Single и возвращает полученный объект
    public static implicit operator Rational(Single num) {
        return new Rational(num); }

    // Явно возвращает объект типа Int32, полученный из Rational
    public static explicit operator Int32(Rational r) {
        return r.ToInt32(); }

    // Явно возвращает объект типа Single, полученный из Rational
    public static explicit operator Single(Rational r) {
        return r.ToSingle(); }
}
```

При определении методов для операторов преобразования следует указать, должен ли компилятор генерировать код для их неявного вызова автоматически или лишь при наличии явного указания в исходном тексте. Ключевое слово

`implicit` указывает компилятору C#, что наличие в исходном тексте явного приведения типов не обязательно для генерации кода, вызывающего метод оператора преобразования. Ключевое слово `explicit` позволяет компилятору вызывать метод только тогда, когда в исходном тексте происходит явное приведение типов.

После ключевого слова `implicit` или `explicit` следует поместить указание (ключевое слово `operator`) компилятору, что данный метод представляет собой оператор преобразования. После ключевого слова `operator` надо указать целевой тип, в который преобразуется объект, а в скобках — исходный тип объекта.

Определив в показанном ранее типе `Rational` операторы преобразования, можно написать (на C#):

```
public sealed class Program {  
    public static void Main() {  
        Rational r1 = 5;           // Неявное приведение Int32 к Rational  
        Rational r2 = 2.5F;        // Неявное приведение Single к Rational  
        Int32 x = (Int32) r1;      // Явное приведение Rational к Int32  
        Single s = (Single) r2;    // Явное приведение Rational к Single  
    }  
}
```

При исполнении этого кода «за кулисами» происходит следующее. Компилятор C# обнаруживает в исходном тексте операции приведения (преобразования типов) и при помощи внутренних механизмов генерирует IL-код, который вызывает методы операторов преобразования, определенные в типе `Rational`. Но каковы имена этих методов? На этот вопрос можно ответить, скомпилировав тип `Rational` и изучив его метаданные. Оказывается, компилятор генерирует по одному методу для каждого из определенных операторов преобразования. Метаданные четырех методов операторов преобразования, определенных в типе `Rational`, выглядят примерно так:

```
public static Rational op_Implicit(Int32 num)  
public static Rational op_Implicit(Single num)  
public static Int32 op_Explicit(Rational r)  
public static Single op_Explicit(Rational r)
```

Как видите, методы, выполняющие преобразование объектов одного типа в объекты другого типа, всегда называются `op_Implicit` или `op_Explicit`. Определять оператор неявного преобразования следует, только когда точность или величина значения не теряется в результате преобразования, например при преобразовании `Int32` в `Rational`. Если же точность или величина значения в результате преобразования теряется (например, при преобразовании объекта типа `Rational` в `Int32`), следует определять оператор явного преобразования. На случай сбоя явного преобразования в операторном методе следует предусмотреть выбрасывание исключения `OverflowException` или `InvalidOperationException`.

ПРИМЕЧАНИЕ

Два метода с именем `op_Explicit` принимают одинаковый параметр — объект типа `Rational`. Но эти методы возвращают значения разных типов: `Int32` и `Single` соответственно. Это пример пары методов, отличающихся лишь типом возвращаемого значения. CLR в полном объеме поддерживает предоставление типам возможности определить несколько методов, отличающихся только типами возвращаемых значений. Однако эта возможность доступна лишь очень немногим языкам. Как вы, вероятно, знаете, C++, C#, Visual Basic и Java не поддерживают определение нескольких методов, единственное различие которых — в типе возвращаемого значения. Лишь несколько языков (например, IL) позволяют разработчику явно выбирать, какой метод вызвать. Конечно, IL-программистам не следует использовать эту возможность, так как определенные таким образом методы будут недоступны для вызова из программ, написанных на других языках программирования. Хотя C# не предоставляет эту возможность, внутренние механизмы компилятора все равно используют ее, если в типе определены методы операторов преобразования.

Компилятор C# полностью поддерживает операторы преобразования. Обнаружив код, в котором вместо ожидаемого типа используется объект совсем другого типа, компилятор ищет метод оператора неявного преобразования, способный выполнить нужное преобразование, и генерирует код, вызывающий этот метод. Если подходящий метод оператора неявного преобразования обнаруживается, компилятор вставляет в результирующий IL-код вызов этого метода. Найдя в исходном тексте явное приведение типов, компилятор ищет метод оператора явного или неявного преобразования. Если он существует, компилятор генерирует вызывающий его код. Если компилятор не может найти подходящий метод оператора преобразования, он генерирует ошибку, и код не компилируется.

ПРИМЕЧАНИЕ

C# генерирует код вызова операторов неявного преобразования в случае, когда используется выражение приведения типов. Однако операторы неявного преобразования никогда не вызываются, если используется оператор `as` или `is`.

Чтобы по-настоящему разобраться в методах перегруженных операторов и операторов преобразования, я настоятельно рекомендую использовать тип `System.Decimal` как наглядное пособие. В типе `Decimal` определено несколько конструкторов, позволяющих преобразовывать в `Decimal` объекты различных типов. Он также поддерживает несколько методов `ToXxx` для преобразования объектов типа `Decimal` в объекты других типов. Наконец, в этом типе определен ряд методов операторов преобразования и перегруженных операторов.

Методы расширения

Лучше всего понять механизм методов расширения — это проиллюстрировать его на примере. В главе 14 я обращаю внимание на то, что для управления строками класс `StringBuilder` предлагает меньше методов, чем класс `String`, и это довольно странно, потому что класс `StringBuilder` является предпочтительнее для управления строками, так как он изменяем. Например, пусть вам необходимо определить некоторые отсутствующие в классе `StringBuilder` методы самостоятельно. Возможно, вы решите определить собственный метод `IndexOf`:

```
public static class StringBuilderExtensions {  
    public static Int32 IndexOf(StringBuilder sb, Char value) {  
        for (Int32 index = 0; index < sb.Length; index++)  
            if (sb[index] == value) return index;  
        return -1;  
    }  
}
```

Теперь, когда этот метод у вас определен, вы можете использовать его:

```
// Инициализирующая строка  
StringBuilder sb = new StringBuilder("Hello. My name is Jeff.");  
  
// Замена точки восклицательным знаком  
// и получение номера символа в первом предложении (5)  
Int32 index = StringBuilderExtensions.IndexOf(sb.Replace('.', '!'), '!');
```

Этот программный код прекрасно работает, но в перспективе он не идеален. Первой проблемой для программистов, желающих получить индекс символа при помощи класса `StringBuilder`, будет существование класса `StringBuilderExtensions`. Второй проблемой окажется то, что программный код не отражает последовательность операторов, представленных в объекте `StringBuilder`, что плохо для понимания, чтения и поддержки. Программистам удобнее было бы вызывать сначала метод `Replace`, а затем метод `IndexOf`, но когда вы прочитаете последнюю строчку кода слева направо, первым в строке окажется `IndexOf`, а затем — `Replace`. Конечно, вы можете исправить ситуацию и сделать поведение программного кода более понятным, написав следующий код:

```
// Замена точки восклицательным знаком  
sb.Replace('.', '!');  
  
// Получение номера символа в первом предложении (5)  
Int32 index = StringBuilderExtensions.IndexOf(sb, '!');
```

Однако существует и третья проблема с обеими версиями этого программного кода, мешающая понять его работу. Везде присутствует класс `StringBuilderExtensions`, что отвлекает программиста от использования метода

IndexOf. Если в классе `StringBuilder` определен метод `IndexOf`, то представленный код можно переписать следующим образом:

```
// Замена точки восклицательным знаком
// и получение номера символа в первом предложении (5)
Int32 index = sb.Replace('.', '!').IndexOf('!');
```

В контексте поддержки программного кода это выглядит великолепно! В объекте `StringBuilder` мы заменили точку восклицательным знаком и затем нашли индекс этого знака.

А сейчас я попробую объяснить, что именно делают методы расширения. Они позволяют вам определить статический метод, который вызывается посредством синтаксиса экземплярного метода. Или, другими словами, мы можем определить собственный метод `IndexOf` — и тремя проблемами станет меньше. Для того чтобы превратить метод `IndexOf` в метод расширения, мы просто добавим ключевое слово `this` перед первым аргументом:

```
public static class StringBuilderExtensions {
    public static Int32 IndexOf(this StringBuilder sb, Char value) {
        for (Int32 index = 0; index < sb.Length; index++)
            if (sb[index] == value) return index;
        return -1;
    }
}
```

Компилятор увидит следующий код:

```
Int32 index = sb.IndexOf('X');
```

Сначала компилятор проверит класс `StringBuilder` или все его базовые классы, предлагающие экземплярные методы с названием `IndexOf` и единственным параметром `Char`. Если таковые существуют, тогда компилятор сгенерирует IL-код для вызова метода. Если же они не существуют, тогда компилятор будет искать любой статический класс с определенным методом `IndexOf`, у которого первый параметр соответствует типу выражения, используемого при вызове метода. Этот тип должен быть отмечен при помощи ключевого слова `this`. В данном примере выражением является `sb` типа `StringBuilder`. В этом случае компилятор ищет метод `IndexOf` с двумя параметрами: `StringBuilder` (отмеченное словом `this`) и `Char`. Компилятор найдет наш метод `IndexOf` и сгенерирует IL-код для вызова нашего статического метода.

Теперь понятно, как компилятор решает две последние упомянутые мной проблемы, относящиеся к читабельности кода. Однако до сих пор непонятно, как решается первая проблема, то есть как программисты узнают о том, что метод `IndexOf` существует и может использоваться в объекте `StringBuilder`? Ответ на этот вопрос в `Microsoft Visual Studio` дает механизм `Intellisense`. В редакторе, когда вы напечатаете точку, появится `Intellisense`-окно со списком доступных методов сущности. Кроме того, в `Intellisense`-окне будут

представлены все методы расширения, существующие для типа выражения, написанного слева от точки. IntelliSense-окно показано на рис. 8.1. Как видите, рядом с методами расширения имеется стрелочка, а контекстная подсказка показывает, что метод действительно является методом расширения. Это по-настоящему прекрасно, потому что теперь при помощи этого инструмента легко определить собственный метод для управления различными типами объектов и предоставить его для изучения другим программистам, использующим объекты этих типов.

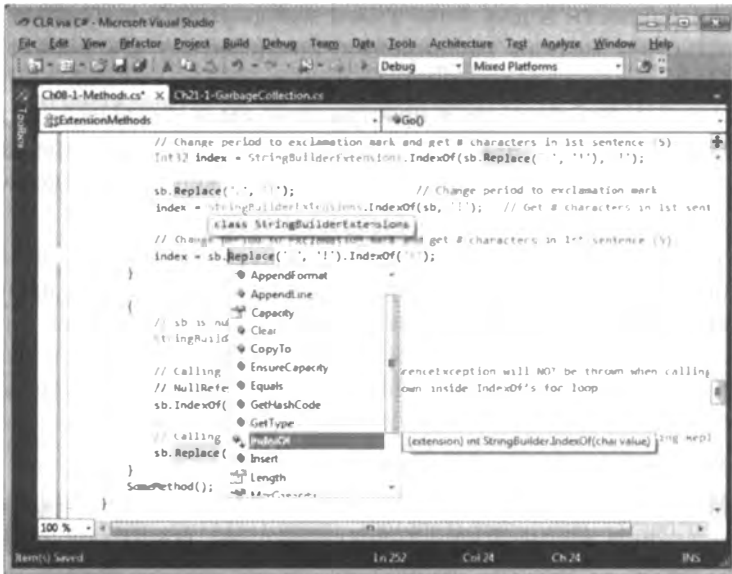


Рис. 8.1. IntelliSense-окно в Visual Studio, показывающее методы расширения

Правила и рекомендации

Приведу несколько правил и рекомендаций, касающихся методов расширения.

- ❑ Язык C# поддерживает только методы расширения, он не поддерживает свойств расширения, событий расширения, операторов расширения и т. д.
- ❑ Методы расширения (методы со словом `this` перед первым аргументом) должны быть объявлены в статическом необобщенном классе. Однако нет ограничения на имя этого класса, вы можете назвать его как вам угодно. Конечно, метод расширения должен иметь, по крайней мере, один параметр, и только первый параметр может быть отмечен ключевым словом `this`.
- ❑ Компилятор C# ищет методы расширения, заданные только в статических классах, определенных в области файла. Другими словами, если вы определили статический класс, унаследованный от другого класса, компилятор C# выдаст следующее сообщение (ошибка CS1109: Метод расширения должен

быть определен в статическом классе первого уровня, `StringBuilderExtensions` является унаследованным классом):

```
error CS1109: Extension method must be defined in a top-level static
class; StringBuilderExtensions is a nested class
```

- ❑ Так как статическим классам можно давать любые имена по вашему желанию, компилятору C# необходимо какое-то время для того, чтобы найти методы расширения; он просматривает все статические классы, определенные в области файла, и сканирует их статические методы. Для повышения производительности и для того, чтобы игнорировать ненужные в данных обстоятельствах методы расширения, компилятору C# необходимо импортировать методы расширения. Например, пусть кто-нибудь определил класс `StringBuilderExtensions` в пространстве имен `Wintellect`, тогда другой программист, которому нужно иметь доступ к методу расширения данного класса, в начале файла программного кода должен указать команду `using Wintellect`.
- ❑ Существует возможность определения в нескольких статических классах одинаковых методов расширения. Если компилятор выяснит, что существуют два и более методов расширения, то тогда он выдает следующее сообщение (ошибка CS0121: Неочевидный вызов следующих методов или свойств '`StringBuilderExtensions.IndexOf(string, char)`' и '`AnotherStringBuilderExtensions.IndexOf(string, char)`'):


```
error CS0121: The call is ambiguous between the following methods
or properties: 'StringBuilderExtensions.IndexOf(string, char)'
and 'AnotherStringBuilderExtensions.IndexOf(string, char)'.
```

Для того чтобы исправить эту ошибку, вы должны модифицировать программный код. Нельзя использовать синтаксис экземплярного метода для вызова статического метода, вместо этого должен применяться синтаксис статического метода, где указывается имя статического класса, чтобы явно указать компилятору, какой именно метод нужно вызвать.

- ❑ Прибегать к этому механизму следует не слишком часто, так как не все разработчики к нему привычны. Например, когда вы расширяете тип с методом расширения, вы действительно расширяете унаследованные типы с этим методом. Следовательно, вы не должны определять метод выражения, чей первый параметр — `System.Object`, так как этот метод будет вызываться для всех типов выражения, и соответствующие ссылки заполнят собой IntelliSense-окно.
- ❑ Существует потенциальная проблема с версиями. Если в будущем разработчики Microsoft добавят экземплярный метод `IndexOf` к классу `StringBuilder` с тем же прототипом, что и в моем примере, то когда я перекомпилирую свой программный код, компилятор свяжет с программой экземплярный метод `IndexOf` компании Microsoft вместо моего статического метода `IndexOf`.

Из-за этого моя программа будет вести себя по-другому. Эта проблема версий является еще одной причиной, по которой этот механизм следует использовать редко.

Различные расширяющие типы с методами расширения

В этой главе я продемонстрировал, как определять методы расширения для класса `StringBuilder`. Я хотел бы отметить, что так как метод расширения на самом деле является вызовом статического метода, то среда CLR не генерирует код для проверки значения выражения, используемого для вызова метода (равно ли оно `null`).

```
// sb не null
StringBuilder sb = null;

// Вызов метода выражения: исключение NullReferenceException НЕ БУДЕТ
// вброшено при вызове IndexOf
// Исключение NullReferenceException будет вброшено внутри цикла IndexOf
sb.IndexOf('X');

// Вызов экземплярного метода: исключение NullReferenceException БУДЕТ
// вброшено при вызове Replace
sb.Replace('.', '!');
```

Я также хотел бы отметить, что вы можете определять методы расширения для интерфейсных типов, как в следующем программном коде:

```
public static void ShowItems<T>(this IEnumerable<T> collection) {
    foreach (var item in collection)
        Console.WriteLine(item);
}
```

Представленный здесь метод расширения может быть вызван при помощи любого выражения, результат выполнения которого имеет тип, реализуемый интерфейсом `IEnumerable<T>`:

```
public static void Main() {
    // Показывает каждый символ в каждой строке консоли
    "Grant".ShowItems();

    // Показывает каждую строку в каждой строке консоли
    new[] { "Jeff", "Kristin" }.ShowItems();

    // Показывает каждый Int32 в каждой строки консоли.
    new List<Int32>() { 1, 2, 3 }.ShowItems();
}
```

ВНИМАНИЕ

Методы расширения являются краеугольным камнем предлагаемой Microsoft технологии Language Integrated Query (LINQ). В качестве хорошего примера класса с большим количеством методов расширения обратите внимание на статический класс `System.Linq.Enumerable` и все его статические методы расширения в документации `Microsoft .NET Framework SDK`. Каждый метод расширения в этом классе расширяет либо интерфейс `IEnumerable`, либо интерфейс `IEnumerable<T>`.

Вы также можете определять методы расширения для типов-делегатов (примеры см. в главе 11). Кроме того, можно добавлять методы расширения к перечислимым типам (примеры см. в главе 15).

И наконец, компилятор C# позволяет создавать делегатов, ссылающихся на метод расширения поверх объекта (см. главу 17):

```
public static void Main () {  
    // Создание делегата Action, ссылающегося на статический метод расширения  
    // ShowItems, и его первый аргумент инициализируется  
    // ссылкой на строку "Jeff"  
    Action a = "Jeff".ShowItems;  
    .  
    .  
    .  
    // Вызов делегата, вызывающего ShowItems и передающего  
    // ссылку на строку "Jeff"  
    a();  
}
```

В представленном программном коде компилятор C# генерирует IL-код для того, чтобы создать делегата `Action`. После создания делегата конструктор передается в вызываемый метод, также передается ссылка на объект, который должен быть передан в этот метод в качестве скрытого параметра. Обычно, когда вы создаете делегата, ссылающегося на статический метод, объектная ссылка равна `null`, потому что статический метод не имеет этого параметра. Однако в данном примере компилятор C# сгенерирует специальный код, создающий делегата, ссылающегося на статический метод `ShowItems`, и целью объекта статического метода будет ссылка на строку `"Jeff"`. Позднее, когда делегат будет вызван, CLR вызовет статический метод и передаст ему ссылку на строку `"Jeff"`. Это довольно хорошо работает и выглядит естественно в силу того, что позволяет не думать о том, что происходит внутри.

Атрибут расширения

Было бы отлично, если бы эта концепция методов расширения относилась бы не только к C#. Хотелось бы, чтобы программисты определяли набор методов

расширения на разных языках программирования и, таким образом, способствовали развитию других языков программирования. Для того чтобы этот механизм работал, компилятор должен поддерживать поиск статических типов и методов для сопоставления с методами расширения. И компиляторы должны это проделывать быстро.

В языке C#, когда вы помечаете первый параметр статического метода ключевым словом `this`, компилятор применяет соответствующий атрибут к методу, и данный атрибут сохраняется в метаданных результирующего файла. Этот атрибут определен в сборке `System.Core.dll` и выглядит следующим образом:

```
// Определен в пространстве имен System.Runtime.CompilerServices
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class |
AttributeTargets.Assembly)]
public sealed class ExtensionAttribute : Attribute {
}
```

К тому же этот атрибут применяется к метаданным любого статического класса, содержащего, по крайней мере, один метод расширения. Итак, когда скомпилированный код вызывает несуществующий экземплярный метод, компилятор может быстро просканировать все ссылающиеся сборки, чтобы определить, какая из них содержит методы расширения. В дальнейшем он может сканировать только те сборки статических классов, которые содержат методы расширения, выполняя поиск потенциальных соответствий компилируемому коду настолько быстро, насколько это возможно.

ПРИМЕЧАНИЕ

Класс `ExtensionAttribute` определен в сборке `System.Core.dll`. Это означает, что результирующая сборка, сгенерированная компилятором, будет иметь ссылку на встроенную в нее библиотеку `System.Core.dll`, даже если не использовать какой-либо тип из `System.Core.dll` и даже если не ссылаться на него во время компиляции программного кода. Однако это не такая уже большая проблема, потому что `ExtensionAttribute` применяется только один раз во время компиляции, и во время выполнения `System.Core.dll` не загрузится, пока приложение занято чем-либо другим в этой сборке.

Частичные методы

Представьте, что вы используете инструмент, который производит исходный программный код на C#, содержащий определение типа. Этому инструменту известно, что внутри программного кода есть места, в которых вы хотели бы настроить поведение типа. Обычно такая настройка производится при помо-

щи сгенерированного инструментом кода, вызывающего виртуальные методы. Такой код также содержит определения для этих виртуальных методов, где их реализация ничего не делает, а просто осуществляет возврат управления. Чтобы настроить поведение класса, нужно определить собственный класс, унаследованный от базового, и затем переопределить все его виртуальные методы, реализующие желаемое поведение. Вот пример:

```
internal class Base {
    private String m_name;
    // Вызывается перед изменением поля m_name
    protected virtual void OnNameChanging(String value) {
    }
    public String Name {
        get { return m_name; }
        set {
            // Информировать класс о возможных изменениях
            OnNameChanging(value.ToUpper());
            m_name = value; // Изменение поля
        }
    }
}

// Написанный программистом код из другого файла
internal class Derived : Base {
    protected override void OnNameChanging(string value) {
        if (String.IsNullOrEmpty(value))
            throw new ArgumentNullException("value");
    }
}
```

К сожалению, для представленного кода характерны две проблемы.

- ❑ Тип не должен быть изолированным (sealed) классом. Нельзя использовать этот подход для изолированных классов или для значимых типов (потому что значимые типы неявно изолированы). К тому же нельзя использовать этот подход для статических методов, потому что они не могут быть переопределены.
- ❑ Существует проблема эффективности. Тип, будучи определенным, переопределяет метод, что отнимает некоторое количество системных ресурсов. И даже если вы не хотите переопределять поведение типа `OnNameChanging`, код базового класса по-прежнему вызовет виртуальный метод, который помимо возврата управления ничего больше не делает. Метод `ToUpper` вызывается и тогда, когда `OnNameChanging` получает доступ к переданным аргументам, и тогда, когда не получает.

Для решения проблемы переопределения поведения можно задействовать частичные методы языка C#. В следующем коде для достижения той же цели, что и в предыдущем коде, используются частичные методы:

```
// Сгенерированный при помощи инструмента программный код
internal sealed partial class Base {
    private String m_name;
    // Это объявление с определением частичного метода вызывается
    // перед изменением поля m_name
    partial void OnNameChanging(String value);
    public String Name {
        get { return m_name; }
        set {
            // Информирование класса о потенциальном изменении
            OnNameChanging(value.ToUpper());
            m_name = value; // Изменение поля
        }
    }
}
```

```
// Написанный программистом код, содержащийся в другом файле
internal sealed partial class Base {
    // Это объявление с реализацией частичного метода вызывается перед тем,
    // как будет изменено поле m_name
    partial void OnNameChanging(String value) {
        if (String.IsNullOrEmpty(value))
            throw new ArgumentNullException("value");
    }
}
```

Есть несколько мест в этом коде, на которые необходимо обратить внимание.

- ❑ Класс изолирован (хотя не должен бы). В действительности, класс мог бы быть статическим классом или даже значимым типом.
- ❑ Код, сгенерированный инструментом, и код, написанный программистом, на самом деле являются двумя частичными определениями, которые в конце концов дают одно определение типа (детали см. в главе 6).
- ❑ Код, сгенерированный инструментом, представляет собой объявление частичного метода. Этот метод помечен ключевым словом `partial` и не имеет тела.
- ❑ Код, написанный программистом, реализует объявление частичного метода. Этот метод также помечен ключевым словом `partial` и тоже не имеет тела.

Когда вы скомпилируете этот код, вы увидите то же самое, что и в представленном ранее коде. Польза здесь в том, что вы можете перезапустить ин-

струмент и сгенерировать новый код в новый файл, но ваш программный код по-прежнему останется нетронутым в отдельном файле. Этот подход работает для изолированных классов, статических классов и значимых типов.

ПРИМЕЧАНИЕ

В редакторе Visual Studio, если вы напишете `partial` и нажмете пробел, в IntelliSense-окне появятся объявления всех частичных методов вложенного типа, которые пока не имеют соответствия объявлениям выполняемого частичного метода. Вы легко можете выбрать частичный метод в IntelliSense-окне, и Visual Studio сгенерирует прототип метода автоматически. Это прекрасная функциональная особенность, повышающая производительность.

Существует и еще одно серьезное улучшение, которое мы получаем благодаря частичным методам. Скажем, у вас теперь нет нужды модифицировать поведение типа, сгенерированного инструментом, и менять файл исходного кода. Если просто скомпилировать такой код, компилятор создаст IL-код и метаданные, как если бы сгенерированный инструментом код выглядел следующим образом:

```
// Логический эквивалент сгенерированного инструментом кода в случае,  
// когда нет объявления выполняемого частичного метода  
internal sealed class Base {  
    private String m_name;  
    public String Name {  
        get { return m_name; }  
        set {  
            m_name = value; // Измените поле  
        }  
    }  
}
```

Если у вас не будет объявления выполняемого частичного метода, компилятор не сгенерирует метаданные, представляющие частичный метод. К тому же компилятор не сгенерирует IL-команды вызова частичного метода, он не сгенерирует код, вычисляющий аргументы, которые необходимо передать частичному методу. В приведенном примере компилятор не сгенерирует код для вызова метода `ToUpper`. В результате будет меньше метаданных и IL-кода, и производительность во время выполнения повысится!

ПРИМЕЧАНИЕ

Подобным образом частичные методы работают с атрибутом `System.Diagnostics.ConditionalAttribute`. Однако они работают только с одним типом, тогда как атрибут `ConditionalAttribute` может быть использован для необязательного вызова методов, определенных в другом типе.

Правила и рекомендации

Предлагаю вам несколько дополнительных правил и рекомендаций, касающихся частичных методов.

- ❑ Частичные методы могут объявляться внутри частичного класса или структуры.
- ❑ Частичные методы должны всегда иметь возвращаемый тип `void` и не могут иметь параметров, помеченных ключевым словом `out`. Эти ограничения связаны с тем, что во время выполнения программы метода не существует, и вы не можете инициализировать переменную, возвращаемую методом, потому что этого метода не существует. По той же причине нельзя использовать параметр, помеченный словом `out`, потому что иначе метод должен будет инициализировать этот параметр, но этого метода не существует. Частичный метод может иметь параметры, помеченные ключевым словом `ref`, а также универсальные параметры, экземплярные или статические, или даже параметры, помеченные как `unsafe`.
- ❑ Естественно, определяющее объявление частичного метода и его реализующее объявление должны иметь идентичные сигнатуры. И оба должны иметь настраиваемые атрибуты, применяющиеся к ним, когда компилятор объединяет атрибуты обоих методов вместе. Все атрибуты, применяемые к параметрам, тоже объединяются.
- ❑ Если не существует реализующего объявления частичного метода, вы не сможете иметь программный код, пытающийся создать делегата, ссылающегося на частичный метод. Это причина, по которой метод не существует во время выполнения программы. Компилятор выдаст следующее сообщение (ошибка CS0762: Не могу создать делегата из метода '`Base.OnNameChanging(string)`', потому что это частичный метод без реализующего объявления):

```
"error CS0762: Cannot create delegate from method  
'Base.OnNameChanging(string)' because it is a partial method  
without an implementing declaration
```
- ❑ Хотя частичные методы всегда считаются закрытыми, компилятор C# запрещает писать ключевое слово `private` перед объявлением частичного метода.

Глава 9. Параметры

В этой главе рассмотрены различные способы передачи параметров в метод. В числе прочего вы узнаете, как определить необязательный параметр, задать параметр по имени и передать его по ссылке. Также рассмотрена процедура определения методов, принимающих различное количество аргументов.

Необязательные и именованные параметры

При выборе параметров метода некоторым из них или даже им всем можно присваивать значения по умолчанию. В результате в вызывающем такой метод коде можно не указывать эти аргументы, а принимать уже имеющиеся значения. Кроме того, при вызове метода существует возможность указать аргументы, воспользовавшись именами их параметров. Вот код, демонстрирующий применение, как необязательных, так и именованных параметров:

```
public static class Program {
    private static Int32 s_n = 0;
    private static void M(Int32 x = 9, String s = "A",
        DateTime dt = default(DateTime), Guid guid = new Guid()) {
        Console.WriteLine("x={0}, s={1}, dt={2}, guid={3}", x, s, dt, guid);
    }
    public static void Main() {
        // 1. Аналогично: M(9, "A", default(DateTime), new Guid());
        M();
        // 2. Аналогично: M(8, "X", default(DateTime), new Guid());
        M(8, "X");
        // 3. Аналогично: M(5, "A", DateTime.Now, Guid.NewGuid());
        M(5, guid: Guid.NewGuid(), dt: DateTime.Now);
        // 4. Аналогично: M(0, "1", default(DateTime), new Guid());
        M(s_n++, s_n++.ToString());
        // 5. Аналогично: String t1 = "2"; Int32 t2 = 3;
        // M(t2, t1, default(DateTime), new Guid());
        M(s: (s_n++).ToString(), x: s_n++);
    }
}
```

Запустив этот код, получим:

```
x=9, s=A, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=8, s=X, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
```

```
x=5, s=A, dt=7/2/2009 10:14:25 PM, guid=d24a59da-6009-4aae-9295-839155811309  
x=0, s=1, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000  
x=3, s=2, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
```

Как видите, в случае если при вызове метода аргументы отсутствуют, компилятор берет их значения, предлагаемые по умолчанию. В третьем и пятом вызовах метода М заданы *именованные параметры* (named parameter). Я в явном виде передал значение переменной *x* и указал, что хочу передать аргумент для параметров *guid* и *dt*.

Передаваемые в метод аргументы компилятор рассматривает слева направо. В четвертом вызове метода М значение аргумента *s_n* (0) передается в переменную *x*, затем *s_n* увеличивается на единицу и аргумент *s_n* (1) передается как строка в параметр *s*. После чего *s_n* увеличивается до 2. Передача аргументов с помощью именованных параметров опять же осуществляется компилятором слева направо. В пятом вызове метода М значение параметра *s_n* (2) преобразуется в строку и сохраняется в созданной компилятором временной переменной (*t1*). Затем *s_n* увеличивается до 3, и это значение сохраняется в еще одной созданной компилятором временной переменной (*t2*). После этого *s_n* увеличивается до 4. В конце концов, вызывается метод М, в который передаются переменные *t2*, *t1*, переменная *DateTime* со значением по умолчанию и переменная *Guid*, помеченная ключевым словом *new*.

Правила работы с параметрами

Определяя метод, задающий для части своих параметров значения по умолчанию, следует руководствоваться следующими правилами:

- ❑ Значения по умолчанию указываются для параметров методов, конструкторов методов и параметрических свойств (индексаторов C#). Также их можно указывать для параметров, являющихся частью определения делегатов. В результате при вызове этого типа делегата аргументы можно опускать, используя их значения по умолчанию.
- ❑ Параметры со значениями по умолчанию должны следовать за всеми остальными параметрами. Другими словами, если указан параметр со значением по умолчанию, значения по умолчанию должны иметь и все параметры, расположенные справа от него. Например, если при определении метода М удалить значение по умолчанию ("A") для параметра *s*, компилятор выдаст сообщение об ошибке. Существует только одно исключение из правил — параметр массива, помеченный ключевым словом *params* (о котором мы подробно поговорим чуть позже). Он должен располагаться после всех прочих параметров, в том числе имеющих значение по умолчанию. При этом сам массив значения по умолчанию иметь не может.
- ❑ Во время компиляции значения по умолчанию должны оставаться неизменными. То есть задавать значения по умолчанию можно для параметров

примитивных типов, перечисленных в табл. 5.1 главы 5. Сюда относятся также перечислимые типы и ссылочные типы, допускающие присвоение значения `null`. Для параметров произвольного значимого типа значение по умолчанию задается как экземпляр этого типа с полями, содержащими нули. Можно использовать как ключевое слово `default`, так и ключевое слово `new`, в обоих случаях генерируется одинаковый IL-код. С примерами обоих вариантов синтаксиса мы уже встречались в методе `M` при задании значений по умолчанию для параметров `dt` и `guid` соответственно.

- ❑ Запрещается переименовывать параметрические переменные, так как это влечет за собой необходимость редактирования вызывающего кода, который передает аргументы по имени параметра. Скажем, если в объявлении метода `M` переименовать переменную `dt` в `dateTime`, то третий вызов метода станет причиной появления следующего сообщения компилятора (ошибка CS1739: в перегруженной версии 'M' отсутствует параметр с именем 'dt'):

```
"error CS1739: The best overload for 'M' does not have a parameter
named 'dt'"
```

- ❑ При вызове метода извне модуля изменение значения параметров по умолчанию является потенциально опасным. Вызывающая сторона использует значение по умолчанию в процессе работы. Если изменить его и не перекомпилировать код, содержащий вызов, в вызываемый метод будет передано прежнее значение. В качестве индикатора поведения можно использовать значение по умолчанию `0` или `null`. В результате исчезает необходимость повторной компиляции кода вызывающей стороны. Вот пример:

```
// Не делайте так:
private static String MakePath(String filename = "Untitled") {
    return String.Format(@"C:\{0}.txt", filename);
}
// Вместо этого напишите:
private static String MakePath(String filename = null) {
    // Здесь применяется оператор, поддерживающий
    // значение null (??); см. главу 19
    return String.Format(@"C:\{0}.txt", filename ?? "Untitled");
}
```

- ❑ Для параметров, помеченных ключевыми словами `ref` или `out`, значения по умолчанию не задаются.

Существуют также дополнительные правила вызова методов с использованием необязательных или именованных параметров:

- ❑ Аргументы можно передавать в произвольном порядке; но именованные аргументы должны находиться в конце списка.
- ❑ Передать аргумент по имени можно и в параметры, не имеющие значения по умолчанию, но при этом компилятору должны быть переданы все требуемые аргументы (с указанием их позиции или имени).

- ❑ В C# между запятыми не могут отсутствовать аргументы. То есть вот такая запись `M(1, .DateTime.Now)` недопустима, так как ведет к нечитабельному коду. Чтобы опустить аргумент для параметра со значением по умолчанию, передавайте аргументы по именам параметров.
- ❑ Вот как передать аргумент по имени параметра, требующего ключевого слова `ref/out`:

```
// Объявление метода:  
private static void M(ref Int32 x) { ... }  
// Вызов метода:  
Int32 a = 5;  
M(x: ref a);
```

ПРИМЕЧАНИЕ

Синтаксис необязательных и именованных параметров в C# весьма удобен при написании кода, поддерживающего объектную модель COM из Microsoft Office. При вызове COM-компонентов C# позволяет опускать ключевые слова `ref/out` в процессе передачи аргументов по ссылке. Это еще больше упрощает код. Если же COM-компонент не вызывается, наличие рядом с аргументом ключевого слова `ref/out` обязательно.

Атрибут `DefaultParameterValue` и необязательные атрибуты

Хотелось бы, чтобы концепция заданных по умолчанию и необязательных аргументов выходила за пределы C#. Особенно здорово было бы, если бы программисты могли определять методы, указывающие, какие параметры являются необязательными и каковы должны быть заданные по умолчанию значения параметров в разных языках программирования, дав попутно возможность вызывать их из разных языковых сред. Но такой поход срабатывает только при условии, что выбранный компилятор позволяет при вызове опускать некоторые аргументы, а также умеет определять заданные по умолчанию значения этих аргументов.

В C# параметрам со значением по умолчанию назначается настраиваемый атрибут `System.Runtime.InteropServices.OptionalAttribute`, сохраняющийся в метаданных итогового файла. Кроме того, компилятор применяет к параметру атрибут `System.Runtime.InteropServices.DefaultParameterValueAttribute`, опять же сохраняя его в метаданных итогового файла. После чего конструктору `DefaultParameterValueAttribute` передаются постоянные значения, указанные в первоначальном коде.

В итоге встречая код, вызывающий метод, в котором не хватает аргументов, компилятор проверяет, являются ли эти аргументы необязательными, берет их значения из метаданных и автоматически вставляет в вызов метода.

Неявно типизированные локальные переменные

В C# поддерживается возможность определения типа используемых в методе локальных переменных по типу используемого при их инициализации выражения. Вот пример:

```
private static void ImplicitlyTypedLocalVariables() {
    var name = "Jeff";
    ShowVariableType(name); // Вывод: System.String
    // var n = null; // Ошибка
    var x = (Exception)null; // ОК, хотя соответствующее значение отсутствует
    ShowVariableType(x); // Вывод: System.Exception
    var numbers = new Int32[] { 1, 2, 3, 4 };
    ShowVariableType(numbers); // Вывод: System.Int32[]
    // Меньше символов при вводе сложных типов
    var collection = new Dictionary<String, Single>() { { ".NET", 4.0f } };
    // Вывод: System.Collections.Generic.Dictionary`2[System.String,System.
Single]
    ShowVariableType(collection);
    foreach (var item in collection) {
        // Вывод: System.Collections.Generic.KeyValuePair`2[System.String,System.
Single]
        ShowVariableType(item);
    }
}
private static void ShowVariableType<T>(T t) {
    Console.WriteLine(typeof(T));
}
```

Первая строка кода метода `ImplicitlyTypedLocalVariables` вводит новую локальную переменную при помощи ключевого слова `var`. Чтобы определить ее тип, компилятор смотрит на тип выражения с правой стороны от оператора присваивания (`=`). Так как `"Jeff"` — это строка, компилятор присваивает переменной `name` тип `String`. Для доказательства корректности полученного результата я написал универсальный метод `ShowVariableType`. Он определяет тип своего аргумента и выводит его на консоль. Для простоты чтения выводимые методом `ShowVariableType` значения я добавил в виде комментариев внутрь метода `ImplicitlyTypedLocalVariables`.

Вторая операция присваивания (превращенная в комментарий) в методе `ImplicitlyTypedLocalVariables` во время компиляции привела бы к ошибке (ошибка `CS0815`: невозможно присвоить значение `null` локальной переменной с неявно заданным типом):

```
error CS0815: Cannot assign <null> to an implicitly-typed local variable
```

Дело в том, что значение `null` неявно приводится к любому ссылочному типу или значимому типу, допускающему значение `null`. Соответственно, компилятор не в состоянии однозначно определить его тип. Однако в третьей операции присваивания я показал, что инициализировать локальную переменную с неявно заданным типом значением `null` все-таки можно, если в явном виде указать тип (в моем примере это тип `Exception`). Впрочем, это не самая полезная возможность, так как, написав `Exception x = null;`, вы получите тот же самый результат.

В четвертом примере в полной мере демонстрируется ценность локальных переменных неявно заданного типа. Ведь без этой возможности вам бы потребовалось с обеих сторон от оператора присваивания писать `Dictionary<String, Single>`. Это не просто увеличивает объем набираемого текста, но и заставляет редактировать код с обеих сторон от оператора присваивания в случае, если вы решите поменять тип коллекции или любой из типов обобщенных параметров.

В цикле `foreach` я также воспользовался ключевым словом `var`, заставив компилятор автоматически определить тип элементов коллекции. Этот пример демонстрирует пользу ключевого слова `var` внутри инструкций `foreach`, `using` и `for`. Кроме того, оно полезно в процессе экспериментов с кодом. К примеру, вы инициализируете локальную переменную с неявно заданным типом, взяв за основу тип возвращаемого методом значения. Но в будущем может появиться необходимость поменять тип возвращаемого значения. В этом случае компилятор автоматически определит, что тип возвращаемого методом значения изменился, и изменит тип локальной переменной! К сожалению, остальной код внутри метода, работающий с этой переменной при условии доступа к членам, использующим эту переменную в предположении, что она принадлежит к старому типу, может перестать компилироваться.

В Microsoft Visual Studio при наведении указателя мыши на ключевое слово `var` появляется всплывающая подсказка с названием типа, определяемого компилятором. Функцию неявного задания типа локальных переменных в C# следует задействовать при работе с методами, использующими анонимные типы. Они подробно рассматриваются в главе 10.

Тип параметра метода при помощи ключевого слова `var` объявлять нельзя. Ведь компилятор будет определять его, исходя из типа аргументов, передаваемых при вызове метода. Вызова же может вообще не быть или же, наоборот, их может быть несколько. Аналогично, нельзя объявлять при помощи этого ключевого слова тип поля. Для такого ограничения в C# существует множество причин. Одной из них является возможность доступа к полю из нескольких методов. В этом случае контракт (тип переменной) следует указывать явно. Второй причиной является тот факт, что в данном случае анонимные типы (обсуждаемые в главе 10) начнут выходить за границы одного метода.

ВНИМАНИЕ

Нельзя путать ключевые слова `dynamic` и `var`. Объявление локальной переменной с ключевым словом `var` является не более чем синтаксическим сокращением, заставляющим компилятор определить тип данных по выражению. Данное ключевое слово служит только для объявления локальных переменных внутри метода, в то время как ключевое слово `dynamic` используется для локальных переменных, полей и аргументов. Невозможно привести выражение к типу `var`, но такая операция вполне допустима для типа `dynamic`. Переменные, объявленные с ключевым словом `var`, следует инициализировать явно, что не обязательно для переменных типа `dynamic`. Более подробную информацию о динамическом типе вы найдете в главе 5.

Передача параметров в метод по ссылке

По умолчанию CLR предполагает, что все параметры методов передаются по значению. При передаче объекта ссылочного типа методу передается ссылка (или указатель) на этот объект. То есть метод может изменить переданный объект, влияя на состояние вызывающего кода. Если параметром является экземпляр значимого типа, методу передается его копия. В этом случае метод получает собственную копию объекта, позволяя сохранить неизменным исходный материал.

ВНИМАНИЕ

Следует знать тип каждого объекта, передаваемого методу в качестве параметра, поскольку манипулирующий параметрами код может существенно различаться в зависимости от типа параметров.

CLR также позволяет передавать параметры по ссылке, а не по значению. В C# это делается с помощью ключевых слов `out` и `ref`. Оба заставляют компилятор генерировать метаданные, описывающие параметр как переданный по ссылке. Компилятор использует эти метаданные для генерации кода, передающего вместо самого параметра его адрес.

С точки зрения CLR, ключевые слова `out` и `ref` не различаются, то есть для них генерируются одинаковые метаданные и IL-код. Однако компилятор C# различает эти ключевые слова при выборе метода, используемого для инициализации объекта, на который указывает переданная ссылка. Если параметр метода помечен ключевым словом `out`, вызывающий код может не инициализировать его, пока не вызван сам метод. В этом случае вызванный метод не может прочесть значение параметра и должен записать его, прежде чем вернуть управление. Если же параметр помечен ключевым словом `ref`, вызывающий

код должен инициализировать его перед вызовом метода, а вызванный метод может как читать, так и записывать значение параметра.

Поведение ссылочных и значимых типов при использовании ключевых слов `out` и `ref` различается значительно. Вот как это выглядит в случае значимого типа:

```
public sealed class Program {
    public static void Main() {
        Int32 x;           // Инициализация x
        GetVal(out x);      // Инициализация x не обязательна
        Console.WriteLine(x); // Выводится 10
    }
    private static void GetVal(out Int32 v) {
        v = 10; // Этот метод должен инициализировать переменную v
    }
}
```

Здесь переменная `x` объявлена в стеке `Main`, а ее адрес передается методу `GetVal`. Параметр этого метода `v` представляет собой указатель на значимый тип `Int32`. Внутри метода `GetVal` значение типа `Int32`, на которое указывает `v`, изменяется на 10. Именно это значение выводится на консоль, когда метод `GetVal` возвращает управление. Использование ключевого слова `out` со значимыми типами повышает эффективность кода, так как предотвращает копирование экземплярных полей значимого типа при вызовах методов.

А теперь рассмотрим пример, в котором фигурирует ключевое слово `ref`:

```
public sealed class Program {
    public static void Main() {
        Int32 x = 5;       // Инициализация x
        AddVal(ref x);      // x требуется инициализировать
        Console.WriteLine(x); // Выводится 15
    }
    private static void AddVal(ref Int32 v) {
        v += 10; // Этот метод может использовать инициализированный параметр v
    }
}
```

Здесь объявленной в стеке `Main` переменной `x` присваивается начальное значение 5. Затем ее адрес передается методу `AddVal`, параметр `v` которого представляет собой указатель на значимый тип `Int32` в стеке `Main`. Внутри метода `AddVal` должно быть уже инициализированное значение типа `Int32`, на которое указывает параметр `v`. Таким образом, метод `AddVal` может использовать первоначальное значение `v` в любом выражении. Он может менять это значение, возвращая вызывающему коду новый вариант. В рассматриваемом примере метод `AddVal` прибавляет к исходному значению 10. Соответственно, когда он

возвращает управление, переменная `x` метода `Main` содержит значение 15, которое и выводится на консоль.

В завершение отметим, что с точки зрения IL или CLR ключевые слова `out` и `ref` ничем не различаются: оба заставляют передать указатель на экземпляр объекта. Разница в том, что они помогают компилятору гарантировать корректность кода. В представленном далее коде делается попытка передать методу, ожидающему параметр `ref`, не инициализированное значение, что вызывает ошибку компиляции (ошибка CS0165: использование локальной переменной `x`, у которой не задано значение):

error CS0165: Use of unassigned local variable 'x'

А вот сам фрагмент кода, вызывающий это сообщение:

```
public sealed class Program {
    public static void Main() {
        Int32 x; // x не инициализируется
        // Следующая строка не компилируется, а выводится сообщение:
        // error CS0165: Use of unassigned local variable 'x'
        AddVal(ref x);
        Console.WriteLine(x);
    }
    private static void AddVal(ref Int32 v) {
        v += 10; // Этот метод может использовать инициализированный параметр v
    }
}
```

ВНИМАНИЕ

Меня часто спрашивают, почему при вызовах методов в программах на C# надо в явном виде указывать ключевые слова `out` или `ref`. В конце концов, компилятор в состоянии самостоятельно определить, какое из ключевых слов ему требуется, а значит, должен корректно компилировать код. Однако разработчики C# сочли, что вызывающий код должен явно указывать свои намерения, чтобы при вызове метода сразу было ясно, что этот метод должен менять значение передаваемой переменной.

Кроме того, CLR позволяет по-разному перегружать методы в зависимости от выбора параметра `out` или `ref`. Например, следующий код на C# вполне допустим и прекрасно компилируется:

```
public sealed class Point {
    static void Add(Point p) { ... }
    static void Add(ref Point p) { ... }
}
```

Не допускается перегружать методы, отличающиеся только типом параметров (`out` или `ref`), так как результатом их JIT-компиляции становится идентичный код метаданных, представляющих сигнатуру методов. Поэтому в показанном ранее типе `Point` я не могу определить вот такой метод:

```
static void Add(out Point p) { ... }
```

При попытке включить такой метод в тип `Point` компилятор `C#` вернет ошибку (ошибка `CS0663`: в `'Add'` нельзя определять перегруженных методов, отличных от `ref` и `out`):

```
error CS0663: 'Add' cannot define overloaded methods that differ only on ref and out
```

Со значимыми типами ключевые слова `out` и `ref` дают тот же результат, что и передача ссылочного типа по значению. Они позволяют методу управлять единственным экземпляром значимого типа. Вызывающий код должен выделить память для этого экземпляра, а вызванный метод управляет выделенной памятью. В случае ссылочных типов вызывающий код выделяет память для указателя на передаваемый объект, а вызванный код управляет этим указателем. В силу этих особенностей использование ключевых слов `out` и `ref` со ссылочными типами полезно, лишь когда метод собирается «вернуть» ссылку на известный ему объект. Рассмотрим это на примере:

```
using System;
using System.IO;
public sealed class Program {
    public static void Main() {
        FileStream fs; // Объект fs не инициализирован
        // Первый файл открывается для обработки
        StartProcessingFiles(out fs);
        // Продолжаем, пока остаются файлы для обработки
        for (; fs != null; ContinueProcessingFiles(ref fs)) {
            // Обработка файла
            fs.Read(...);
        }
    }
    private static void StartProcessingFiles(out FileStream fs) {
        fs = new FileStream(...); // в этом методе объект fs
                                   // должен инициализироваться
    }
    private static void ContinueProcessingFiles(ref FileStream fs) {
        fs.Close(); // Закрытие последнего обрабатываемого файла
        // Открыть следующий файл или вернуть null, если файлов больше нет
        if (noMoreFilesToProcess) fs = null;
        else fs = new FileStream (...);
    }
}
```

Как видите, главная особенность этого кода в том, что методы с параметрами ссылочного типа, помеченными ключевыми словами `out` или `ref`, создают объект и возвращают вызывающему коду указатель на него. Обратите внимание, что метод `ContinueProcessingFiles` может управлять передаваемым ему объектом, прежде чем вернет новый объект. Это возможно, благодаря тому, что его па-

параметр помечен ключевым словом `ref`. Показанный здесь код можно немного упростить:

```
using System;
using System.IO;
public sealed class Program {
    public static void Main() {
        FileStream fs = null; // Обязательное присвоение
                               // начального значения null
        // Открытие первого файла для обработки
        ProcessFiles(ref fs);
        // Продолжаем, пока остаются необработанные файлы
        for (; fs != null; ProcessFiles(ref fs)) {
            // Обработка файла
            fs.Read(...);
        }
    }
    private static void ProcessFiles(ref FileStream fs) {
        // Если предыдущий файл открыт, закрываем его
        if (fs != null) fs.Close(); // Закрыть последний обрабатываемый файл
        // Открыть следующий файл или вернуть null, если файлов больше нет
        if (noMoreFilesToProcess) fs = null;
        else fs = new FileStream (...);
    }
}
```

Вот еще один пример, демонстрирующий использование ключевого слова `ref` для реализации метода, меняющего местами два ссылочных типа:

```
public static void Swap(ref Object a, ref Object b) {
    Object t = b;
    b = a;
    a = t;
}
```

Кажется, что код, меняющий местами ссылки на два объекта типа `String`, должен выглядеть так:

```
public static void SomeMethod() {
    String s1 = "Jeffrey";
    String s2 = "Richter";
    Swap(ref s1, ref s2);
    Console.WriteLine(s1); // Выводит "Richter"
    Console.WriteLine(s2); // Выводит "Jeffrey"
}
```

Однако компилироваться этот код не будет: ведь переменные, передаваемые методу по ссылке, должны быть одного типа, объявленного в сигнатуре мето-

да. Иначе говоря, метод `Swap` ожидает ссылок на тип `Object`, а не на тип `String`. Решение же нашей задачи выглядит следующим образом:

```
public static void SomeMethod() {
    String s1 = "Jeffrey";
    String s2 = "Richter";
    // Тип передаваемых по ссылке переменных должен
    // соответствовать ожидаемому
    Object o1 = s1, o2 = s2;
    Swap(ref o1, ref o2);
    // Приведение объектов к строковому типу
    s1 = (String) o1;
    s2 = (String) o2;
    Console.WriteLine(s1); // Выводит "Richter"
    Console.WriteLine(s2); // Выводит "Jeffrey"
}
```

Такая версия метода `SomeMethod` будет компилироваться и работать нужным нам образом. Причиной ограничения, которое нам пришлось обходить, является обеспечение безопасности типов. Вот пример кода (к счастью, он не компилируется), нарушающего безопасность типов:

```
internal sealed class SomeType {
    public Int32 m_val;
}

public sealed class Program {
    public static void Main() {
        SomeType st;
        // Следующая строка выдает ошибку CS1503: Argument '1':
        // cannot convert from 'ref SomeType' to 'ref object'.
        GetAnObject(out st);
        Console.WriteLine(st.m_val);
    }

    private static void GetAnObject(out Object o) {
        o = new String('X', 100);
    }
}
```

Совершенно ясно, что здесь метод `Main` ожидает от метода `GetAnObject` объект `SomeType`. Однако поскольку в сигнатуре `GetAnObject` задана ссылка на `Object`, метод `GetAnObject` может инициализировать параметр `o` объектом произвольного типа. В этом примере параметр `st` в момент, когда метод `GetAnObject` возвращает управление методу `Main`, ссылается на объект типа `String`, в то время как ожидается тип `SomeType`. Соответственно, вызов метода `Console.WriteLine` закончится неудачей. Впрочем, компилятор `C#` откажется компилировать этот код, так как `st` представляет собой ссылку на объект типа `SomeType`, тогда как метод `GetAnObject` требует ссылку на `Object`.

Однако, как оказалось, эти методы можно заставить работать при помощи обобщений. Вот так следует исправить показанный ранее метод Swap:

```
public static void Swap<T>(ref T a, ref T b) {
    T t = b;
    b = a;
    a = t;
}
```

После этого следующий код (идентичный ранее показанному) будет без проблем компилироваться и выполняться:

```
public static void SomeMethod() {
    String s1 = "Jeffrey";
    String s2 = "Richter";
    Swap(ref s1, ref s2);
    Console.WriteLine(s1); // Выводит "Richter"
    Console.WriteLine(s2); // Выводит "Jeffrey"
}
```

За другими примерами решения этой задачи при помощи обобщений отсылаю вас к классу `System.Threading.Interlocked` с его универсальными методами `CompareExchange` и `Exchange`.

Передача в метод переменного количества аргументов

Иногда разработчику удобно определить метод, способный принимать переменное число параметров. Например, тип `System.String` предлагает методы, выполняющие объединение произвольного числа строк, а также методы, при вызове которых можно задать набор единообразно форматируемых строк.

Метод, принимающий переменное число аргументов, объявляют так:

```
static Int32 Add(params Int32[] values) {
    // ПРИМЕЧАНИЕ: при необходимости этот массив
    // можно передать другим методам
    Int32 sum = 0;
    if (values != null) {
        for (Int32 x = 0; x < values.Length; x++)
            sum += values[x];
    }
    return sum;
}
```

Незнакомым в этом методе является только ключевое слово `params`, примененное к последнему параметру в сигнатуре метода. Если не обращать

на него внимания, станет ясно, что метод принимает массив значений типа `Int32`, складывает все элементы этого массива и возвращает полученную сумму.

Очевидно, этот метод можно вызвать так:

```
public static void Main() {  
    // Выводит "15"  
    Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 } ));  
}
```

Не вызывает сомнений утверждение, что этот массив легко инициализировать произвольным числом элементов и передать для обработки методу `Add`. Показанный здесь код немного неуклюж, хотя он корректно компилируется и работает. Разработчики, конечно, предпочли бы вызывать метод `Add` так:

```
public static void Main() {  
    // Выводит "15"  
    Console.WriteLine(Add(1, 2, 3, 4, 5));  
}
```

Наверное, вам будет приятно узнать, что это возможно благодаря ключевому слову `params`. Именно оно заставляет компилятор рассматривать параметр как экземпляр настраиваемого атрибута `System.ParamArrayAttribute`.

Обнаружив такой вызов, компилятор `C#` проверяет все методы с заданным именем, у которых ни один из параметров не помечен атрибутом `ParamArray`. Найдя метод, способный принять вызов, компилятор генерирует вызывающий его код. В противном случае ищутся методы с атрибутом `ParamArray` и проверяется, могут ли они принять вызов. Если компилятор находит подходящий метод, то прежде чем сгенерировать код его вызова, он генерирует код, создающий и заполняющий массив.

В предыдущем примере вы не обнаружите метод `Add`, принимающий пять совместимых с типом `Int32` аргументов. Компилятор же видит в тексте исходного кода вызов метода `Add`, которому передается список значений `Int32`, и метод `Add`, у которого массив типа `Int32` помечен атрибутом `ParamArray`. Компилятор считает данный метод подходящим для вызова и генерирует код, собирающий все параметры в массив типа `Int32` и вызывающий метод `Add`. В конечном итоге получается, что можно написать вызов, без труда передающий методу `Add` набор параметров, и компилятор сгенерирует тот же код, что и для первой версии вызова метода `Add`, в которой массив создается и инициализируется явно.

Ключевым словом `params` может быть помечен только последний параметр метода (`ParamArrayAttribute`). Он должен указывать на одномерный массив произвольного типа. В последнем параметре метода допустимо передавать значение `null` или ссылку на массив, состоящий из нуля элементов. Следующий вызов

метода `Add` прекрасно компилируется, отлично работает и дает в результате сумму, равную 0 (как и ожидалось):

```
public static void Main() {  
    // Обе строки выводят "0"  
    Console.WriteLine(Add()); // передает новый элемент Int32[0] методу Add  
    Console.WriteLine(Add(null)); // передает методу Add значение  
                                // null, что более эффективно  
}
```

Все показанные до сих пор примеры демонстрировали методы, принимающие произвольное количество параметров типа `Int32`. А как написать метод, принимающий параметры любого типа? Ответ прост: достаточно отредактировать прототип метода, заставив его вместо `Int32[]` принимать `Object[]`. Вот метод, отображающий значения `Type` всех переданных ему объектов:

```
public sealed class Program {  
    public static void Main() {  
        DisplayTypes(new Object(), new Random(), "Jeff", 5);  
    }  
    private static void DisplayTypes(params Object[] objects) {  
        if (objects != null) {  
            foreach (Object o in objects)  
                Console.WriteLine(o.GetType());  
        }  
    }  
}
```

Такой код даст следующий результат:

```
System.Object  
System.Random  
System.String  
System.Int32
```

ВНИМАНИЕ

Вызов метода, принимающего переменное число аргументов, снижает производительность, если, конечно, не передавать в явном виде значение `null`. В любом случае всем объектам массива нужно выделить место в куче и инициализировать элементы массива, а по завершении работы занятая массивом память должна быть очищена сборщиком мусора. Чтобы уменьшить негативное влияние этих операций на производительность, можно определить несколько перегруженных методов, в которых не используется ключевое слово `params`. За примерами обратитесь к методу `Concat` класса `System.String`, который перегружен следующим образом:

```
public sealed class String : Object. ... {  
    public static string Concat(object arg0);  
    public static string Concat(object arg0, object arg1);
```

```
public static string Concat(object arg0,  
    object arg1, object arg2);  
public static string Concat(params object[] args);  
public static string Concat(string str0, string str1);  
public static string Concat(string str0,  
    string str1, string str2);  
public static string Concat(string str0,  
    string str1, string str2, string str3);  
public static string Concat(params string[] values);  
}
```

Как видите, для метода `Concat` определены несколько вариантов перегрузки, в которых ключевое слово `params` не используется. Здесь представлены наиболее распространенные варианты перегрузки, которые, собственно, и предназначены для повышения эффективности работы в стандартных ситуациях. Варианты перегрузки с ключевым словом `params` предназначены для более редких ситуаций, поскольку при этом страдает производительность. К счастью, такие ситуации возникают не так уж часто.

Типы параметров и возвращаемых значений

Объявляя тип параметров метода, нужно по возможности указывать наиболее «мягкие» типы, предпочитая интерфейсы базовым классам. Например, при написании метода, работающего с набором элементов, лучше всего объявить параметр метода, используя интерфейс `IEnumerable<T>` вместо строгого типа данных, например `List<T>`, или еще более строгого интерфейсного типа `ICollection<T>` или `IList<T>`:

```
// Рекомендуется в этом методе использовать параметр мягкого типа  
public void ManipulateItems<T>(IEnumerable<T> collection) { ... }  
// Не рекомендуется в этом методе использовать параметр строгого типа  
public void ManipulateItems<T>(List<T> collection) { ... }
```

Причина, конечно же, в том, что первый метод можно вызывать, передав в него массив, объект `List<T>`, объект `String` и т. п., то есть любой объект, тип которого реализует интерфейс `IEnumerable<T>`. Второй метод принимает только объекты `List<T>`, с массивами или объектами `String` он работать уже не может. Ясно, что первый метод предпочтительнее, так как он гибче и может использоваться в более разнообразных ситуациях.

Естественно, создавая метод, принимающий список (а не просто любой перечислимый объект), нужно объявлять тип параметра как `IList<T>`, в то время как типа `List<T>` лучше избегать. Именно такой подход позволит вызывающему коду передавать массивы и другие объекты, тип которых реализует `IList<T>`.

Обратите внимание, что в приводимых примерах речь идет о коллекциях, созданных с использованием архитектуры интерфейсов. Этот же подход применим к классам, опирающимся на архитектуру базовых классов. Потому, к примеру, при реализации метода, обрабатывающего байты из потока, пишем следующее:

```
// Рекомендуется в этом методе использовать параметр мягкого типа
public void ProcessBytes(Stream someStream) { ... }
// Не рекомендуется в этом методе использовать параметр строгого типа
public void ProcessBytes(FileStream fileStream) { ... }
```

Первый метод может обрабатывать байты из потока любого вида: `FileStream`, `NetworkStream`, `MemoryStream` и т. п. Второй поддерживает только `FileStream`, то есть область его применения ограничена.

В то же время, объявляя тип возвращаемого методом объекта, желательно выбирать самый строгий из доступных вариантов (пытаясь не ограничиваться конкретным типом). Например, лучше объявлять метод, возвращающий объект `FileStream`, а не `Stream`:

```
// Рекомендуется в этом методе использовать
// строгий тип возвращаемого объекта
public FileStream OpenFile() { ... }
// Не рекомендуется в этом методе использовать
// мягкий тип возвращаемого объекта
public Stream OpenFile() { ... }
```

Здесь предпочтительнее первый метод, так как он позволяет вызывающему коду обращаться с возвращаемым объектом как с объектом `FileStream` или `Stream`. А вот второму методу требуется, чтобы вызывающий код рассчитывал только на объект `Stream`, то есть область его применения более ограничена.

Иногда требуется сохранить возможность изменять внутреннюю реализацию метода, не влияя на вызывающий код. В приведенном ранее примере изменение реализации метода `OpenFile` в будущем маловероятно, он вряд ли будет возвращать что-либо отличное от объекта типа `FileStream` (или типа, производного от `FileStream`). Однако для метода, возвращающего объект `List<String>`, вполне возможно изменение реализации, после которого он начнет возвращать тип `String[]`. В подобных случаях следует выбирать более мягкий тип возвращаемого объекта. Например:

```
// Гибкий вариант: в этом методе используется
// мягкий тип возвращаемого объекта
public IList<String> GetStringCollection() { ... }
// Негибкий вариант: в этом методе используется
// строгий тип возвращаемого объекта
public List<String> GetStringCollection() { ... }
```

Хотя в коде метода `GetStringCollection` используется и возвращается объект `List<String>`, в прототипе метода лучше указать в качестве возвращаемого объ-

екта `IList<String>`. Даже если в будущем указанная в коде метода коллекция изменит свой тип на `String[]`, вызывающий код не потребует ни редактировать, ни даже перекомпилировать. Обратите внимание, что в этом примере я выбрал самый «строгий» из самых «мягких» типов. К примеру, я не воспользовался типом `IEnumerable<String>` или `ICollection<String>`.

Константность

В некоторых языках, в том числе в неуправляемом языке C++, можно объявлять методы и параметры как константы. Этим вы запрещаете коду в экземплярном методе менять поля объекта или объекты, передаваемые в метод. В CLR эта возможность не поддерживается, что стало причиной жалоб со стороны программистов. Так как сама исполняющая среда не поддерживает такой функции, естественно, что она не поддерживается во всех языках (в том числе в C#).

В первую очередь следует заметить, что в неуправляемом коде C++ пометка экземплярного метода или параметра ключевым словом `const` гарантировала неизменность этого метода или параметра стандартными средствами кода. При этом код, меняющий объект или параметр путем игнорирования их «константной» природы или же путем получения адреса объекта или параметра и записи по этому адресу, можно было поместить внутрь метода. В определенном смысле неуправляемый код C++ «врал» программистам, утверждая, что константные объекты или константные параметры вообще нельзя менять.

Создавая реализацию типа, разработчик может просто избегать написания кода, меняющего объекты и параметры. Например, неизменяемыми являются строки, так как класс `String` не предоставляет нужных для этого методов.

Кроме того, специалисты Microsoft не предусмотрели в CLR возможность проверки неизменности константных объектов или константных параметров. CLR пришлось бы при каждой операции записи проверять, не выполняется ли запись в константный объект, что сильно снизило бы эффективность работы программы. Естественно, обнаружение нарушения должно приводить к вбрасыванию исключения. Более того, поддержка констант создает дополнительные сложности для разработчиков. В частности, при наследовании неизменяемого типа производные типы должны соблюдать это ограничение. Кроме того, неизменяемый тип, скорее всего, должен состоять из полей, которые тоже представляют собой неизменяемые типы.

Это лишь несколько причин, по которым CLR не поддерживает константные объекты/аргументы.

Глава 10. Свойства

В этой главе рассказывается о свойствах. Свойства позволяют обратиться к методу в исходном тексте программы посредством упрощенного синтаксиса. CLR поддерживает два вида свойств: без параметров, их называют просто — *свойства*, и с параметрами — у них в разных языках разное название. Например, в С# свойства с параметрами называют *индексаторами*, а в Visual Basic — *свойствами по умолчанию*. Кроме того, в этой главе рассказывается об инициализации свойств при помощи инициализаторов объектов и коллекций, а также о механизме объединения свойств посредством анонимных типов и типа System.Tuple.

Свойства без параметров

Во многих типах определены сведения о состоянии, которые можно извлечь или изменить. Часто эти сведения о состоянии реализуют в виде таких членов типа, как поля. Вот, например, определение типа с двумя полями:

```
public sealed class Employee {  
    public String Name; // Имя сотрудника  
    public Int32 Age;   // Возраст сотрудника  
}
```

Создавая экземпляр этого типа, можно получить или задать любые сведения о его состоянии при помощи примерно такого кода:

```
Employee e = new Employee();  
e.Name = "Jeffrey Richter"; // Задаем имя сотрудника  
e.Age = 41;                 // Задаем возраст сотрудника
```

```
Console.WriteLine(e.Name); // Выводим на экран "Jeffrey Richter"
```

Этот способ ввода и вывода информации о состоянии объекта очень прост. Однако я готов спорить, что аналогичный код ни в коем случае не следует писать так, как в предыдущем примере. Одним из соглашений объектно-ориентированного программирования и разработки является инкапсуляция данных. Инкапсуляция данных означает, что поля типа ни в коем случае не следует открывать для общего доступа, так как в этом случае слишком просто написать код, способный испортить сведения о состоянии объекта путем ненадлежащего применения полей. Например, следующим кодом разработчик может легко повредить объект Employee:

```
e.Age = -5; // Можете вообразить человека, которому минус 5 лет?
```

Есть и другие причины для инкапсуляции доступа к полям данных типа. Допустим, вам нужен доступ к полю, чтобы что-то сделать, разместить в кэше некоторое значение или создать какой-то внутренний объект, создание которого было отложено, причем обращение к полю не должно нарушать безопасность потоков. Или, скажем, поле является логическим и его значение представлено не байтами в памяти, а вычисляется по некоторому алгоритму.

Каждая из этих причин заставляет при разработке типов, во-первых, помечать все поля как закрытые (`private`), во-вторых, давать пользователю вашего типа возможность получения и задания сведений о состоянии через специальные методы, предназначенные исключительно для этого. Методы, выполняющие функции оболочки для доступа к полю, обычно называют *аксессорами* (`accessor`). Аксессоры могут выполнять дополнительную «зачистку», гарантируя, что сведения о состоянии объекта никогда не будут искажены. Я переписал класс из предыдущего примера следующим образом:

```
public sealed class Employee {
    private String m_Name; // Поле стало закрытым
    private Int32 m_Age;    // Поле стало закрытым

    public String GetName() {
        return(m_Name);
    }

    public void SetName(String value) {
        m_Name = value;
    }

    public Int32 GetAge() {
        return(m_Age);
    }

    public void SetAge(Int32 value) {
        if (value < 0)
            throw new ArgumentOutOfRangeException("value".value.ToString(),
                "The value must be greater than or equal to 0");
        m_Age = value;
    }
}
```

Несмотря на всю простоту, этот пример демонстрирует огромное преимущество инкапсуляции полей данных и простоту создания свойств, доступных только для чтения или только для записи — для чтения или записи достаточно опустить один из аксессоров. В качестве альтернативы можно позволить изменять значения, помеченные методами `SetXxx` как защищенные, только наследуемым типам.

Как видите, у инкапсуляции данных есть два недостатка: во-первых, из-за реализации дополнительных методов приходится писать более длинный код, во-вторых, вместо простой ссылки на имя поля пользователям типа приходится вызывать соответствующие методы:

```
e.SetName("Jeffrey Richter");    // Обновление имени сотрудника
String EmployeeName = e.GetName(); // Получение возраста сотрудника
e.SetAge(41);                    // Обновление возраста сотрудника
e.SetAge(-5);                    // Вброс исключения
                                // ArgumentOutOfRangeException
Int32 EmployeeAge = e.GetAge();   // Получение возраста сотрудника
```

Лично я считаю эти недостатки незначительными. Тем не менее CLR поддерживает механизм свойств, частично компенсирующий первый недостаток и полностью устраняющий второй.

Следующий класс функционально идентичен предыдущему, но в нем используются свойства:

```
public sealed class Employee {
    private String m_Name;
    private Int32 m_Age;

    public String Name {
        get { return(m_Name); }
        set { m_Name = value; } // Ключевое слово value
                                // идентифицирует новое значение
    }

    public Int32 Age {
        get { return(m_Age); }
        set {
            if (value < 0)    // Ключевое слово value всегда
                            // идентифицирует новое значение
                throw new ArgumentOutOfRangeException("value", value.ToString(),
                    "The value must be greater than or equal to 0");
            m_Age = value;
        }
    }
}
```

Как видите, хотя свойства немного усложняют определение типа, они более чем компенсируют дополнительную работу, поскольку позволяют писать код так:

```
e.Name = "Jeffrey Richter";    // "Задать" имя сотрудника
String EmployeeName = e.Name;  // "Получить" имя сотрудника
e.Age = 41;                    // "Задать" возраст сотрудника
e.Age = -5;                    // Вброс исключения
                                // ArgumentOutOfRangeException
Int32 EmployeeAge = e.Age;     // "Получить" возраст сотрудника
```

Можно считать свойства «умными» полями, то есть полями с дополнительной логикой. CLR поддерживает статические, экземплярные, абстрактные и виртуальные свойства. Кроме того, свойства могут помечаться модификатором доступа (см. главу 6) и определяться в интерфейсах (см. главу 13).

У каждого свойства есть имя и тип (но не `void`). Нельзя перегружать свойства (то есть определять несколько свойств с одинаковыми именами, но разным типом). Определяя свойство, обычно описывают пару методов: `get` и `set`. Однако опустив метод `set`, можно определить свойство, доступное только для чтения, а оставив только метод `get`, мы получим свойство, доступное только для записи.

Методы `get` и `set` свойства довольно часто манипулируют закрытым полем, определенным в типе. Это поле обычно называют *вспомогательным* (*backing field*). Однако методам `get` и `set` не приходится обращаться к вспомогательному полю. Например, тип `System.Threading.Thread` поддерживает свойство `Priority`, взаимодействующее непосредственно с ОС, а объект `Thread` не поддерживает поле, хранящее приоритет потока. Другой пример свойств, не имеющих вспомогательных полей, — это неизменяемые свойства, вычисляемые при выполнении: длина массива, заканчивающегося нулем, или область прямоугольника, заданного шириной и высотой и т. д.

При определении свойства компилятор генерирует и помещает в результирующий управляемый модуль следующее:

- ❑ метод `get` свойства генерируется, только если для свойства определен аксессор `get`;
- ❑ метод `set` свойства генерируется, только если для свойства определен аксессор `set`;
- ❑ определение свойства в метаданных управляемого модуля генерируется всегда.

Вернемся к показанному ранее типу `Employee`. При его компиляции компилятор обнаруживает свойства `Name` и `Age`. Поскольку у обоих есть методы-аксессоры `get` и `set`, компилятор генерирует в типе `Employee` четыре определения методов. Результат получается такой, как если бы тип был исходно написан следующим образом:

```
public sealed class Employee {  
    private String m_Name;  
    private Int32 m_Age;  
  
    public String get_Name(){  
        return m_Name;  
    }  
    public void set_Name(String value) {  
        m_Name = value; // Аргумент value всегда идентифицирует новое значение  
    }  
}
```



```

public Int32 get_Age() {
    return m_Age;
}

public void set_Age(Int32 value) {
    if (value < 0) { // value всегда идентифицирует новое значение
        throw new ArgumentOutOfRangeException("value", value.ToString(),
            "The value must be greater than or equal to 0");
    }
    m_Age = value;
}
}

```

Компилятор автоматически генерирует имена этих методов, прибавляя приставку `get_` или `set_` к имени свойства, заданному разработчиком.

Поддержка свойств встроена в C#. Обнаружив код, пытающийся получить или задать свойство, компилятор генерирует вызов соответствующего метода. Если используемый язык не поддерживает свойства напрямую, к ним все равно можно обратиться посредством вызова нужного аксессуора. Эффект тот же, только исходный текст выглядит менее изящно.

Помимо аксессуаров, для каждого из свойств, определенных в исходном тексте, компиляторы генерируют в метаданных управляемого модуля запись с определением свойства. Такая запись содержит несколько флагов и тип свойства, а также ссылки на аксессуары `get` и `set`. Эта информация существует лишь затем, чтобы провести связь между абстрактным понятием «свойства» и его методами доступа. Компиляторы и другие инструменты могут использовать эти метаданные через класс `System.Reflection.PropertyInfo`. И все же CLR не использует эти метаданные, требуя при выполнении только методы-аксессуары.

Автоматически реализуемые свойства

Если необходимо создать свойства для инкапсуляции резервных полей, то в C# есть упрощенный синтаксис, называемый *автоматически реализуемыми свойствами* (Automatically Implemented Properties, AIP). Приведу пример для свойства `Name`:

```

public sealed class Employee {
    // Это свойство является автоматически реализуемым
    public String Name { get; set; }
    private Int32 m_Age;
    public Int32 Age {
        get { return(m_Age); }
        set {
            if (value < 0) // value всегда идентифицирует новое значение
                throw new ArgumentOutOfRangeException("value", value.ToString(),

```

```
        "The value must be greater than or equal to 0");  
        m_Age = value;  
    }  
}
```

Если вы объявите свойства и не обеспечите реализацию методов `set` и `get`, то компилятор C# автоматически объявит их закрытыми полями. В данном примере поле будет иметь тип `String` — тип свойства. И компилятор автоматически реализует методы `get_Name` и `set_Name` для правильного возвращения значения из поля и назначения значения полю. Кажется непонятным, для чего этого нужно, особенно в сравнении с обычным объявлением строкового поля `Name`. И все же есть большая разница. Использование AIP-синтаксиса означает, что вы создаете свойство. Любой программный код, имеющий доступ к этому свойству, вызывает методы `get` и `set`. Если вы позднее решите реализовать эти методы самостоятельно, заменив их реализацию, предложенную компилятором по умолчанию, то код, имеющий доступ к свойству, не нужно будет перекомпилировать. Однако если вы объявите `Name` как поле и позднее замените его свойством, то весь программный код, имеющий доступ к полю, придется перекомпилировать, поскольку он будет иметь доступ к методам свойства.

Лично мне не нравятся автоматически реализуемые свойства, обычно я стараюсь их избегать в силу нескольких причин.

- ❑ Синтаксис объявления поля может включать инициализацию, таким образом, вы объявляете и инициализируете поле в одной строке кода. Однако нет подходящего синтаксиса для установки при помощи AIP начального значения. Следовательно, необходимо неявно инициализировать все автоматически реализуемые свойства во всех конструкторах.
- ❑ Механизм сериализации на этапе выполнения сохраняет имя поля в сериализованном потоке. Имя возвращаемого поля для AIP определяется компилятором, и он может менять это имя каждый раз, когда компилирует код, сводя на нет возможность десериализации экземпляров всех типов, содержащих автоматически реализуемые свойства. Не используйте этот механизм для всех типов, подлежащих сериализации и десериализации.
- ❑ Во время отладки нельзя указать точку останова в AIP-методах `set` и `get`, можно только узнать, когда приложение получит и задаст значение автоматически реализуемого свойства. Точки останова можно устанавливать только в тех свойствах, которые программист пишет самостоятельно.

Вы также должны знать, что при использовании AIP свойства должны иметь уровень доступа для чтения и записи, так как компилятор генерирует методы `set` и `get`. Это разумно, поскольку поля для чтения и записи бесполезны без возможности чтения их значения, более того, поля для чтения бесполезны, если в них будет храниться только значение по умолчанию. К тому же из-за того, что вы не знаете имени автоматически генерируемого вспомогательного

поля, ваш программный код должен всегда иметь доступ к свойству через имя свойства. И если вы решите явно реализовать один из аксессоров, то вам придется явно реализовать оба аксессора и при этом вообще не использовать механизм AIP. Для единственного свойства механизм AIP воплощает подход «все или ничего».

Осторожный подход к определению свойств

Лично мне свойства не нравятся и я был бы рад, если бы в Microsoft решили убрать их поддержку из .NET Framework и сопутствующих языков программирования. Причина в том, что свойства выглядят как поля, являясь по сути методами. Это порождает массу заблуждений и непонимания. Столкнувшись с кодом, обращающимся к полю, разработчик привычно предполагает наличие массы условий, которые далеко не всегда соблюдаются, если речь идет о свойстве.

- ❑ Свойства могут быть доступны только для чтения или только для записи, в то время как поля всегда доступны и для чтения, и для записи. Определяя свойство, лучше всего создавать для него оба аксессора (get и set).
- ❑ Свойство, являясь по сути методом, может привести к исключению, а при доступе к полям исключений не бывает.
- ❑ Свойства нельзя передавать в метод в качестве параметров с ключевым словом `out` или `ref`, в частности следующий код скомпилировать нельзя:

```
using System;
```

```
public sealed class SomeType {
    private static String Name {
        get { return null; }
        set {}
    }

    static void MethodWithoutParam(out String n) { n = null; }

    public static void Main() {
        // При попытке скомпилировать следующую строку
        // компилятор вернет сообщение об ошибке:
        // error CS0206: A property or indexer may not
        // be passed as an out or ref parameter.
        MethodWithoutParam(out Name);
    }
}
```

- ❑ Свойство-метод может выполняться довольно долго, в то время как доступ к полям выполняется моментально. Часто свойства применяют для синхронизации потоков, но это может привести к приостановке потока на

неопределенное время, поэтому свойства не следует использовать для этих целей — в такой ситуации лучше задействовать метод. Кроме того, если предусмотрен удаленный доступ к классу (например, если он наследует от `System.MarshalByRefObject`), вызов свойства-метода выполняется очень медленно, поэтому предпочтение следует отдать методу. Я считаю, что в классах, производных от `MarshalByRefObject`, никогда не следует использовать свойства.

- ❑ При нескольких вызовах подряд свойство-метод может возвращать разные значения, а поле возвращает одно и то же значение. В классе `System.DateTime` есть неизменяемое свойство `Now`, которое возвращает текущие дату и время. При каждом последующем вызове свойство возвращает новое значение. Это ошибка, и в Microsoft с удовольствием исправили бы этот класс, превратив `Now` в метод. Свойство `Environment.TickCount` — еще один пример ошибки.
- ❑ Свойство-метод может порождать видимые сторонние эффекты, невозможные при доступе к полю. Иначе говоря, порядок определения значений различных свойств типа никак не должен влиять на поведение типа, однако в действительности часто бывает не так.
- ❑ Свойству-методу может требоваться дополнительная память или ссылка на объект, не являющийся частью состояния объекта, поэтому изменение возвращаемого объекта никак не сказывается на исходном объекте; при запросе поля всегда возвращается ссылка на объект, который гарантированно относится к состоянию исходного объекта. Свойство, возвращающее копию, — источник путаницы для разработчиков, причем об этом часто забывают упомянуть в документации.

Я узнал, что разработчики используют свойства намного чаще, чем следовало бы. Достаточно внимательно изучить список различий между свойствами и полями, чтобы понять: есть очень немного ситуаций, в которых определение свойства действительно полезно, удобно и не запутывает разработчика. Единственная привлекательная черта свойств — упрощенный синтаксис, все остальное — недостатки, в числе которых потеря в производительности и читабельности кода. Если бы я разрабатывал .NET Framework и компиляторы, я бы вообще отказался от свойств, вместо этого я предоставил бы разработчикам полную свободу реализации методов `GetXxx` и `SetXxx`. Позже создатели компиляторов могли бы предоставить особый упрощенный синтаксис вызова этих методов, но только при условии его отличия от синтаксиса обращения к полям, чтобы программист четко понимал, что выполняется вызов метода!

Свойства и отладчик Visual Studio

Microsoft Visual Studio позволяет указывать свойства объектов в окне просмотра отладчика. В результате при задании точки останова отладчик будет вызывать метод `get` и показывать возвращаемое значение. Это может быть полезно

при поиске ошибок, но также может сказаться на точности и производительности приложения. Например, пусть вы создали поток `FileStream` для файла, передаваемого по сети, и затем добавили свойство `FileStream.Length` в окно просмотра отладчика. Каждый раз, когда вы будете устанавливать точку останова, отладчик вызовет аксессор `get`, который выполнит внутренний сетевой запрос к серверу для получения текущей длины файла.

Таким же образом, если метод-аксессор `get` производит какой-то дополнительный эффект, то этот эффект всегда будет выполняться при достижении точки останова. Например, если метод-аксессор `get` увеличивает счетчик каждый раз во время вызова, то этот счетчик будет каждый раз увеличиваться на точке останова. Из-за этих потенциальных проблем Visual Studio позволяет отключить режим вычислений для свойств, указанных в окне просмотра отладчика. Для этого выберите команду **Tools** ► **Options**, в списке открывшегося окна **Options** раскройте ветвь **Debugging** ► **General** и сбросьте флажок **Enable Property Evaluation And Other Implicit Function Calls** (рис. 10.1). Обратите внимание, что даже отключив таким образом вычисления для свойства, все равно можно будет добавить свойство в окно просмотра отладчика и вручную запустить вычисления, щелкнув мышью на значке вычислений в колонке **Value column** окна просмотра отладчика Visual Studio.

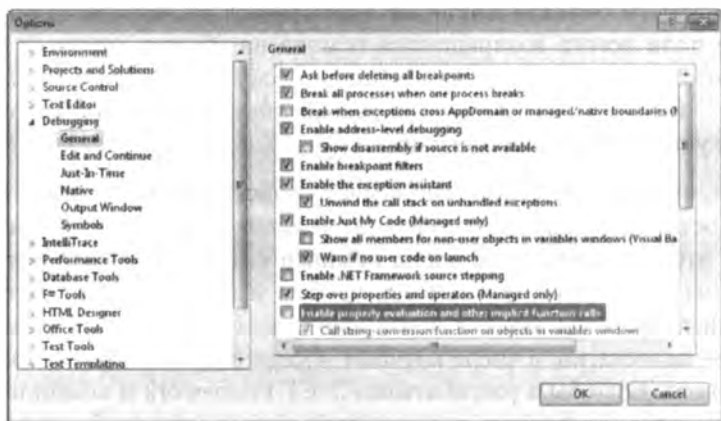


Рис. 10.1. Настройки главного отладчика Visual Studio

Инициализаторы объектов и коллекций

Очень просто создать объект и затем назначить некоторые открытые свойства (или поля) объекта. Для упрощения этого эталона программирования C# предлагает специальный синтаксис инициализации объекта.

Приведу пример:

```
Employee e = new Employee() { Name = "Jeff", Age = 45 };
```

В этом выражении я создаю объект `Employee`, вызываемый непараметризованным конструктором, и затем назначаю открытому свойству `Name` значение `Jeff`, а открытому свойству `Age` — значение `45`. Этот код идентичен следующему коду, в чем вы можете убедиться, преобразуя оба фрагмента на язык IL:

```
Employee e = new Employee();  
e.Name = "Jeff";  
e.Age = 45;
```

Реальная выгода от синтаксиса инициализатора объекта состоит в том, что он позволяет программировать в контексте выражения, строя функции, которые улучшают читабельность кода. Например, можно написать:

```
String s = new Employee() { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

В одном выражении я сконструировал объект `Employee`, вызвал его конструктор, инициализировал два открытых свойства, вызвал метод `ToString`, а затем метод `ToUpper`.

C# позволяет также отказаться от круглых скобок перед открывающейся фигурной скобкой, если вы хотите вызвать непараметризованный конструктор. Вот программный код, производящий IL-код, идентичный предыдущему фрагменту:

```
String s = new Employee { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

Если тип свойства реализует интерфейс `IEnumerable` или `IEnumerable<T>`, то свойство является коллекцией, и инициализация коллекции в противоположность операции замещения является дополнительной операцией. Например, пусть имеется следующее определение класса:

```
public sealed class Classroom {  
    private List<String> m_students = new List<String>();  
    public List<String> Students { get { return m_students; } }  
  
    public Classroom() {}  
}
```

Следующий код создает объект `Classroom` и инициализирует коллекцию `Students`:

```
public static void M() {  
    Classroom classroom = new Classroom {  
        Students = { "Jeff", "Kristin", "Aidan", "Grant" }  
    };  
  
    // Показать 4 студентов в классе  
    foreach (var student in classroom.Students)  
        Console.WriteLine(student);  
}
```

Во время компиляции этого кода компилятор увидит, что свойство `Students` имеет тип `List<String>` и что этот тип реализует интерфейс `IEnumerable<String>`. Компилятор примет, что тип `List<String>` предлагает вызвать метод `Add` (потому что большинство классов коллекций используют метод `Add` для того, чтобы добавлять элементы в коллекцию). Затем компилятор сгенерирует код для вызова метода `Add` коллекции. В результате представленный код будет преобразован компилятором в следующий:

```
public static void M() {  
    Classroom classroom = new Classroom();  
    classroom.Students.Add("Jeff");  
    classroom.Students.Add("Kristin");  
    classroom.Students.Add("Aidan");  
    classroom.Students.Add("Grant");  
  
    // Показать 4 студентов в классе  
    foreach (var student in classroom.Students)  
        Console.WriteLine(student);  
}
```

Если тип свойства реализует интерфейс `IEnumerable` или `IEnumerable<T>`, но не предлагает метод `Add`, тогда компилятор не разрешит использовать синтаксис инициализации коллекции для добавления элемента в коллекцию, вместо этого компилятор выведет такое сообщение: (ошибка **CS0117**: `System.Collections.Generic.IEnumerable<string>` не содержит определения для `Add`):

```
error CS0117: 'System.Collections.Generic.IEnumerable<string>' does not contain  
a definition for 'Add'
```

Некоторые методы `Add` принимают различные аргументы. Например, вот метод `Add` класса `Dictionary`:

```
public void Add(TKey key, TValue value);
```

При инициализации коллекции при помощи фигурных скобок можно добавить в метод `Add` различные аргументы:

```
var table = new Dictionary<String, Int32> {  
    { "Jeffrey", 1 }, { "Kristin", 2 }, { "Aidan", 3 }, { "Grant", 4 }  
};
```

Это равносильно следующему коду:

```
var table = new Dictionary<String, Int32>();  
table.Add("Jeffrey", 1);  
table.Add("Kristin", 2);  
table.Add("Aidan", 3);  
table.Add("Grant", 4);
```

Анонимные типы

Механизм анонимных типов в C# позволяет автоматически объявить кортежный тип при помощи простого синтаксиса. *Кортежный тип* (tuple type)¹ — это тип, который содержит коллекцию свойств, относящихся друг к другу сходным образом. В первой строке следующего программного кода я определяю класс с двумя свойствами (Name типа String и Year типа Int32), создаю экземпляр этого типа и назначаю свойству Name значение Jeff, а свойству Year — значение 1964.

```
// Определение типа, создание сущности и инициализация свойств
var ol = new { Name = "Jeff", Year = 1964 };
// Вывод свойств на консоль
Console.WriteLine("Name={0}, Year={1}", ol.Name, ol.Year); // Выводит:
// Name=Jeff, Year=1964
```

Здесь создается анонимный тип, потому что не был определен тип имени после слова new, таким образом, компилятор автоматически создает имя типа, но не сообщает какое оно (поэтому тип и назван анонимным). Использование синтаксиса инициализации объекта обсуждалось в предыдущем разделе. Итак, я, как разработчик, не имею понятия об имени типа на этапе компиляции и не знаю, с каким типом была объявлена переменная ol. Однако проблемы здесь нет, я могу использовать механизм C# неявно типизированной локальной переменной, о котором говорится в главе 9.

Итак, посмотрим, что же действительно делает компилятор. Обратите внимание на следующий код:

```
var o = new { property1 = expression1, ..., propertyN = expressionN };
```

Когда вы пишете этот код, компилятор делает вывод о типе каждого выражения, создает закрытые поля этих типов, для каждого типа поля создает открытые свойства только для чтения и для всех этих выражений создает конструктор. Код конструктора инициализирует закрытые поля только для чтения путем вычисления результирующих значений. В дополнение к этому, компилятор переопределяет методы Equals, GetHashCode и ToString объекта и генерирует код внутри всех этих методов. Класс, создаваемый компилятором, выглядит следующим образом:

```
[CompilerGenerated]
internal sealed class <f__AnonymousType0<...>: Object {
    private readonly t1 f1;
    public t1 p1 { get { return f1; } }
```

...

продолжение ➤

¹ Термин «tuple» возник как абстракция последовательности: single, double, triple, quadruple, quintuple, n-tuple.


```

private readonly tn fn;
public tn pn { get { return fn; } }
public <f__AnonymousType0<...>(t1 a1, .... tn an) {
    f1 = a1; ...; fn = an; // Назначает все поля
}

public override Boolean Equals(Object value) {
    // Возвращает false, если поля не сопоставлены; иначе возвращает true
}

public override Int32 GetHashCode() {
    // Возвращает хэш-код, сгенерированный из хэш-кодов каждого поля
}

public override String ToString() {
    // Возвращает пары "name = value", разделенные точками
}
}

```

Компилятор генерирует методы `Equals` и `GetHashCode` таким образом, что экземпляры анонимного типа могут быть замещены в хэш-таблице коллекции. Неизменяемые свойства, в отличие от свойств для чтения и записи помогают защитить хэш-код объекта от изменений. Изменение хэш-кода объекта, используемого в качестве ключа в хэш-таблице, может помешать нахождению объекта. Компилятор генерирует метод `ToString` для того, чтобы помочь отладить приложение. В отладчике Visual Studio вы можете навести указатель мыши на переменную, связанную с экземпляром анонимного типа, и Visual Studio вызовет метод `ToString` и покажет результирующую строку в соответствующем окне. Кстати, IntelliSense-окно в Visual Studio будет предлагать то же имя свойства, которое вы написали в коде в редакторе — хорошая функциональная возможность.

Компилятор поддерживает два дополнительных варианта синтаксиса объявления свойства внутри анонимного типа, где из переменных выводятся свойства имен и типов:

```

String Name = "Grant";
DateTime dt = DateTime.Now;

// Анонимный тип с двумя свойствами
// 1. Строковому свойству Name назначено значение Grant
// 2. Свойству Year типа Int32 Year назначен год внутри dt
var o2 = new { Name, dt.Year };

```

В данном примере компилятор определяет, что первое свойство должно называться `Name`. Так как `Name` — это имя локальной переменной, то компилятор устанавливает значение типа свойства аналогичного типу локальной переменной, то есть `String`. Для второго свойства компилятор использует имя

поля/свойства: Year. Year типа Int32 класса DateTime, и, следовательно, свойство Year в анонимном типе будет типа Int32. Когда компилятор создает экземпляр анонимного типа, он назначает экземпляру Name свойство с тем же значением, что и у локальной переменной Name, так что свойство Name будет связано со строкой Grant. Компилятор назначит свойству экземпляра Year то же значение, что и возвращаемое значение из dt свойства Year.

Компилятор очень разумно выясняет анонимный тип. Если компилятор видит, что вы определили множество анонимных типов с идентичными структурами, то он создает одно определение для анонимного типа и множество экземпляров этого типа. Под одинаковой структурой я подразумеваю, что анонимные типы имеют одинаковые тип и имя для каждого свойства и что эти свойства определены в одинаковом порядке. В коде из приведенного примера тип переменной o1 и тип переменной o2 одинаков, так как в двух строках кода определен анонимный тип со свойством Name/String и Year/Int32, и Name стоит перед Year. Раз две переменные имеют один и тот же тип, то можно сделать такую замечательную вещь, как присваивание:

```
// Один тип позволяет осуществлять операции сравнения и присваивания
Console.WriteLine("Objects are equal: " + o1.Equals(o2));
o1 = o2; // Присваивание
```

Раз эти типы идентичны, то можно создать массив явных типов из анонимных типов (о массивах см. главу 16):

```
// Это работает, так как все объекты имеют один анонимный тип
var people = new[] {
    o1, // From earlier in this section
    new { Name = "Kristin", Year = 1970 },
    new { Name = "Aidan", Year = 2003 },
    new { Name = "Grant", Year = 2008 }
};
// Показано, как обрабатывать массив анонимных типов
foreach (var person in people)
    Console.WriteLine("Person={0}, Year={1}", person.Name, person.Year);
```

Анонимные типы обычно используются с технологией языка интегрированных запросов (Language Integrated Query, LINQ), когда запросы записывают результаты в коллекции объектов анонимного типа. Затем объекты обрабатываются в результирующую коллекцию. Все это делается одинаковыми методами. Приведу пример, в котором возвращаются все файлы из папки с моими документами, которые были изменены в последние семь дней:

```
String myDocuments =
    Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
var query =
    from pathname in Directory.GetFiles(myDocuments)
    let LastWriteTime = File.GetLastWriteTime(pathname)
```

продолжение ➤

```

where LastWriteTime > (DateTime.Now - TimeSpan.FromDays(7))
orderby LastWriteTime
select new { Path = pathname, LastWriteTime };

```

```

foreach (var file in query)
    Console.WriteLine("LastWriteTime={0}, Path={1}",
        file.LastWriteTime, file.Path);

```

Экземпляры анонимного типа не предполагают своего распространения за пределы метода. Метод не может иметь прототип, так как предполагается, что нельзя точно определить анонимный тип. Похожим образом, метод не может показать, что он возвращает ссылку на анонимный тип. Пока можно обходиться с экземпляром анонимного типа как с объектом (все анонимные типы наследуются от типа `Object`), нет способа приведения переменной типа `Object` обратно к анонимному типу, потому что имя анонимного типа на этапе компиляции неизвестно. Если вы хотите использовать кортежный тип, требуется тип `System.Tuple`, о котором речь идет в следующем разделе.

Тип `System.Tuple`

В пространстве имен `System` определено несколько обобщенных кортежных типов (все они наследуются от класса `Object`), которые отличаются количеством обобщенных параметров. Приведу наиболее простую и наиболее сложную формы записи.

// Простая форма:

```

[Serializable]
public class Tuple<T1> {
    private T1 m_Item1;
    public Tuple(T1 item1) { m_Item1 = item1; }
    public T1 Item1 { get { return m_Item1; } }
}

```

// Сложная форма:

```

[Serializable]
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> {
    private T1 m_Item1; private T2 m_Item2;
    private T3 m_Item3; private T4 m_Item4;
    private T5 m_Item5; private T6 m_Item6;
    private T7 m_Item7; private TRest m_Rest;
    public Tuple(T1 item1, T2 item2, T3 item3,
        T4 item4, T5 item5, T6 item6, T7 item7, TRest t) {
        m_Item1 = item1; m_Item2 = item2; m_Item3 = item3; m_Item4 = item4;
        m_Item5 = item5; m_Item6 = item6; m_Item7 = item7; m_Rest = rest;
    }
}

```

```
public T1 Item1 { get { return m_Item1; } }  
public T2 Item2 { get { return m_Item2; } }  
public T3 Item3 { get { return m_Item3; } }  
public T4 Item4 { get { return m_Item4; } }  
public T5 Item5 { get { return m_Item5; } }  
public T6 Item6 { get { return m_Item6; } }  
public T7 Item7 { get { return m_Item7; } }  
public TRest Rest { get { return m_Rest; } }  
}
```

Подобно анонимному типу, тип `Tuple` создается один раз и остается неизменным (все свойства только для чтения). Здесь не показано, но классы `Tuple` также позволяют использовать методы `CompareTo`, `Equals`, `GetHashCode` и `ToString`, как и свойство `Size`. К тому же все типы `Tuple` реализуют интерфейсы `IStructuralEquatable`, `IStructuralComparable` и `IComparable`, поэтому вы можете сравнивать два объекта типа `Tuple` друг с другом и смотреть, как их поля сравниваются. Для детального изучения этих методов и интерфейсов посмотрите документацию SDK.

Приведу пример метода, использующего тип `Tuple` для возвращения двух частей информации в вызывающий метод.

```
// Возвращает минимум в Item1 и максимум в Item2  
private static Tuple<Int32, Int32>MinMax(Int32 a, Int32 b) {  
    return new Tuple<Int32, Int32>(Math.Min(a, b), Math.Max(a, b));  
}  
// Здесь показано, как вызывать метод и как использовать Tuple  
private static void TupleTypes() {  
    var minmax = MinMax(6, 2);  
    Console.WriteLine("Min={0}, Max={1}",  
        minmax.Item1, minmax.Item2); // Min=2, Max=6  
}
```

Конечно, очень важно, чтобы и производитель, и потребитель типа `Tuple` имели ясное представление о том, что будет возвращаться в свойствах `Item#`. С анонимными типами свойства получают действительные имена на основе программного кода, определяющего анонимный тип. С типами `Tuple` свойства получают их имена автоматически, и вы не можете их изменить. К несчастью, эти имена не имеют настоящего значения и смысла, а зависят от производителя и потребителя. Это также ухудшает читаемость кода и удобство его сопровождения, так что вы должны добавлять комментарии к коду, чтобы объяснить, что именно производитель/потребитель имеет в виду.

Компилятор может только подразумевать обобщенный тип во время вызова обобщенного метода, а не тогда, когда вы вызываете конструктор. В силу этой причины пространство имен `System` имеет статический необобщенный класс `Tuple`, содержащий группу статических методов `Create`, которые могут выводить обобщенные типы аргументов. Этот класс действует как фабрика по производству объектов типа `Tuple` и нужен просто для упрощения вашего

кода. Вот переписанная с использованием статического класса `Tuple` версия метода `MinMax`:

```
// Возвращает минимум в Item1 и максимум в Item2
private static Tuple<Int32, Int32>MinMax(Int32 a, Int32 b) {
    return Tuple.Create(Math.Min(a, b), Math.Max(a, b)); // Упрощенный
                                                         // синтаксис
}
```

Если вы хотите создать тип `Tuple` с более, чем восьмью элементами, тогда вы должны записать другой тип `Tuple` для параметра `Rest`:

```
var t = Tuple.Create(0, 1, 2, 3, 4, 5, 6, Tuple.Create(7, 8));
Console.WriteLine("{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}";
t.Item1, t.Item2, t.Item3, t.Item4, t.Item5, t.Item6, t.Item7,
t.Rest.Item1.Item1, t.Rest.Item1.Item2);
```

ПРИМЕЧАНИЕ

В дополнение к анонимным типам и кортежным типам вы можете посмотреть на класс `System.Dynamic.ExpandoObject` (определенный в сборке `System.Core.dll assembly`). Когда вы используете этот класс с динамическим типом (о котором говорится в главе 5), у вас есть другой способ группировки наборов свойств (пар ключ-значение) вместе. Результатом оказывается небезопасный для этапа выполнения тип, зато синтаксис выглядит отлично (причем без всякой помощи `IntelliSense`), и вы сможете использовать объекты `ExpandoObject` в `C#` и в динамическом языке типа `Python`. Приведу пример кода с объектом `ExpandoObject`:

```
dynamic e = new System.Dynamic.ExpandoObject();
e.x = 6;           // Добавление свойства 'x' типа Int32,
                  // чье значение равно 6
e.y = "Jeff";      // Добавление свойства 'y' строкового типа,
                  // чье значение равно "Jeff"
e.z = null;        // Добавление свойства 'z' объекта,
                  // чье значение равно null
// Просмотр всех свойств и других значений
foreach (var v in (IDictionary<String, Object>)e)
    Console.WriteLine("Key={0}, V={1}", v.Key, v.Value);
// Удаление свойства 'x' и его значения
var d = (IDictionary<String, Object>)e;
d.Remove("x");
```

Свойства с параметрами

У свойств, рассмотренных в предыдущем разделе, аксессоры `get` не принимали параметры. Поэтому я называю их *свойствами без параметров* (*parameterless properties*). Они проще, так как их использование напоминает обращение к полю.

Помимо таких «полеобразных» свойств, языки программирования поддерживают то, что я называю *свойствами с параметрами* (parameterful properties). У таких свойств аксессоры `get` принимают один или несколько параметров. Разные языки поддерживают свойства с параметрами по-разному. Кроме того, в разных языках свойства с параметрами называют по-разному: в C# — это индексаторы, в Visual Basic — *свойства по умолчанию*. Здесь я остановлюсь на поддержке *индексаторов* в C# при помощи свойств с параметрами.

В C# синтаксис поддержки свойств с параметрами (индексаторов) напоминает синтаксис массивов. Иначе говоря, можно представить индексатор как средство, позволяющее разработчику на C# перегружать оператор `[]`. Приведу пример: класс `BitArray`, который позволяет индексировать набор битов, поддерживаемый экземпляром типа, используя синтаксис массива:

```
using System;
```

```
public sealed class BitArray {  
    // Закрытый байтовый массив, хранящий биты  
    private Byte[] m_byteArray;  
    private Int32 m_numBits;  
  
    // Конструктор, выделяющий память для байтового массива  
    // и устанавливающий все биты в 0  
    public BitArray(Int32 numBits) {  
        // Сохранить число битов  
        if (numBits <= 0)  
            throw new ArgumentOutOfRangeException(  
                "numBits", numBits.ToString(), "numBits must be > 0");  
  
        // Сохранить число битов  
        m_numBits = numBits;  
        // Выделить байты для массива битов  
        m_byteArray = new Byte[(numBits + 7) / 8]; }  
  
    // Это индексатор (свойство с параметрами)  
    public Boolean this[Int32 bitPos] {  
        // Это метод-аксессор get индексатора  
        get {  
            // Сначала нужно проверить аргументы  
            if ((bitPos < 0) || (bitPos >= m_numBits))  
                throw new ArgumentOutOfRangeException("bitPos");  
            // Вернуть состояние индексируемого бита  
            return (m_byteArray[bitPos / 8] & (1 << (bitPos % 8))) != 0; }  
  
        // Это метод-аксессор set индексатора  
        set {  
            if ((bitPos < 0) || (bitPos >= m_numBits))  
                throw new ArgumentOutOfRangeException(  

```

продолжение ➤

```

        "bitPos", bitPos.ToString());
    if (value) {
        // Включить индексируемый бит
        m_byteArray[bitPos / 8] = (Byte)
            (m_byteArray[bitPos / 8] | (1 << (bitPos % 8)));
    } else {
        // Выключить индексируемый бит
        m_byteArray[bitPos / 8] = (Byte)
            (m_byteArray[bitPos / 8] & ~(1 << (bitPos % 8)));
    }
}
}
}
}

```

Использовать индексатор типа `BitArray` невероятно просто:

```

// Выделить массив BitArray, который может хранить 14 бит
BitArray ba = new BitArray(14);

// Включить все четные биты, вызвав аксессор set
for (Int32 x = 0; x < 14; x++) {
    ba[x] = (x % 2 == 0);
}

// Показать состояние всех битов, вызвав аксессор get
for (Int32 x = 0; x < 14; x++) {
    Console.WriteLine("Bit " + x + " is " + (ba[x] ? "On" : "Off"));
}

```

В типе `BitArray` индексатор принимает один параметр `bitPos` типа `Int32`. У каждого индексатора должен быть хотя бы один параметр, но параметров может быть и больше. Тип параметров (как и тип возвращаемого значения) может быть любым.

Индексаторы довольно часто создаются для поиска значений в ассоциативном массиве. Действительно, тип `System.Collections.Hashtable` предлагает индексатор, который принимает ключ и возвращает связанное с ключом значение. В отличие от свойств без параметров, тип может поддерживать множество перегруженных индексаторов при условии, что их сигнатуры различны.

Подобно аксессору `set` свойства без параметров, аксессор `set` индексатора содержит скрытый параметр (в C# его называют `value`), который указывает новое значение, желаемое для «индексируемого элемента».

CLR не различает свойства без параметров и с параметрами. Для среды любое свойство — это всего лишь пара методов, определенных внутри типа. Как уже отмечалось, в различных языках синтаксис создания и использования свойств с параметрами разный. Использование для индексатора в C# конструкции `this[...]` — чистый произвол создателей языка, означающий, что в C# допускается определять индексаторы только на экземплярах объектов. В C# нет синтаксиса, позволяющего разработчику определять статистическое

свойство-индексатор напрямую, хотя на самом деле CLR поддерживает статические свойства с параметрами.

Поскольку CLR обрабатывает свойства с параметрами и без них одинаково, компилятор генерирует в результирующем управляемом модуле те же три элемента:

- ❑ метод `set` свойства с параметрами генерируется, только если у свойства определен аксессор `set`;
- ❑ метод `get` свойства с параметрами генерируется, только если у свойства определен аксессор `get`;
- ❑ определение свойства в метаданных управляемого модуля генерируется всегда; в метаданных нет отдельной таблицы для хранения определений свойств с параметрами: ведь для CLR свойства с параметрами — просто свойства.

Компиляция показанного ранее индексатора типа `BitArray` происходит так, как если бы он исходно был написан следующим образом:

```
public sealed class BitArray {  
  
    // Это метод-аксессор get индексатора  
    public Boolean get_Item(Int32 bitPos) { /* ... */ }  
  
    // Это метод-аксессор set индексатора  
    public void set_Item(Int32 bitPos, Boolean value) { /* ... */ }  
}
```

Компилятор автоматически генерирует имена для этих методов, добавляя к `Item` префикс `get_` или `set_`. Поскольку синтаксис индексаторов в C# не позволяет разработчику задавать имя, создателям компилятора C# пришлось самостоятельно выбрать имя для методов доступа, и они выбрали `Item`. Поэтому имена созданных компилятором методов — `get_Item` и `set_Item`.

Изучая справочник *.NET Framework Reference*, достаточно найти свойство `Item`, чтобы сказать, что данный тип поддерживает индексатор. Так, тип `System.Collections.Generic.List` предлагает открытое экземплярное свойство `Item`, которое является индексатором объекта `List`.

Программируя на C#, вы никогда не увидите имя `Item`, поэтому выбор его компилятором обычно не должен вызывать беспокойства. Однако если вы разрабатываете индексатор типа, который будет доступен программам, написанным на других языках, возможно, придется изменить имена аксессоров индексатора (`get` и `set`). C# позволяет переименовать эти методы, применив к индексатору пользовательский атрибут `System.Runtime.CompilerServices.IndexerNameAttribute`. Пример:

```
using System;  
using System.Runtime.CompilerServices;  
  
public sealed class BitArray {
```



```
[IndexerName("Bit")]
public Boolean this[Int32 bitPos] {
    // Здесь определен по крайней мере один метод доступа
}
}
```

Теперь компилятор сгенерирует вместо методов `get_Item` и `set_Item` методы `get_Bit` и `set_Bit`. Во время компиляции компилятор C# обнаруживает атрибут `IndexerName` и узнает, как именовать метаданные методов и свойств; сам по себе атрибут в метаданных сборки не создается¹.

Приведу фрагмент кода на языке Visual Basic, демонстрирующий получение доступа к индексатору, написанному на C#:

```
' Создать экземпляр типа BitArray
Dim ba as New BitArray(10)

' В Visual Basic элементы массива задают в круглых скобках ().
' а не в квадратных [].
Console.WriteLine(ba(2)) " Выводит True или False

' Visual Basic также позволяет обращаться к индексатору по имени
Console.WriteLine(ba.Bit(2)) ' Выводит то же, что предыдущая строка
```

В C# в одном типе можно определять несколько индексаторов при условии, что они принимают разные наборы параметров. В других языках программирования атрибут `IndexerName` позволяет задать несколько индексаторов с одинаковой сигнатурой, поскольку их имена могут отличаться. Однако C# не допускает этого, так как принятый в C# синтаксис не позволяет ссылаться на индексатор по имени, а значит, компилятор не будет знать, на какой индексатор ссылаются. Попытка компиляции показанного далее исходного текста на C# заставляет компилятор генерировать сообщение об ошибке (ошибка CS0111: в классе `SomeType` уже определен член `this` с таким же типом параметра):

```
error CS0111: Class 'SomeType' already defines a member called 'this' with the
same parameter types
```

Фрагмент кода, вызывающий появление этого сообщения:

```
using System;
using System.Runtime.CompilerServices;

public sealed class SomeType {

    // Определяем метод-аксессор get_Item
    public Int32 this[Boolean b] {
```

¹ По этой причине класс `IndexerNameAttribute` не входит в описанные в ECMA стандарты CLI и языка C#.

```
    get { return 0; }  
}  
  
// Определяем метод-аксессор get_Jeff  
[IndexerName("Jeff")]  
public String this[Boolean b] {  
    get { return null; }  
}  
}
```

Как видите, С# представляет индексаторы как средство перегрузки оператора [], и этот оператор не позволяет различать свойства с одинаковыми наборами параметров и разными именами аксессоров.

Кстати, примером типа с измененным именем индексатора может быть тип System.String, в котором индексатор String именуется Chars, а не Item. Это свойство позволяет получать отдельные символы из строки. Было принято решение, что для языков программирования, не использующих синтаксис с оператором [] для вызова этого свойства, имя Chars будет более информативно.

Обнаружив код, пытающийся получить или заказать значение индексатора, компилятор С# генерирует вызов соответствующего метода доступа. Некоторые языки могут не поддерживать свойства с параметрами. Чтобы получить доступ к свойству с параметрами из программы на таком языке, нужно явно вызвать желаемый метод доступа. CLR не различает свойства с параметрами и без параметров, поэтому для поиска связи между свойством с параметрами и его методами доступа используется все тот же класс System.Reflection.PropertyInfo.

Выбор главного свойства с параметрами

При анализе ограничений, которые С# налагает на индексаторы, возникают два вопроса.

- ❑ Что если язык, на котором написан тип, позволяет разработчику определить несколько свойств с параметрами?
- ❑ Как задействовать этот тип в программе на С#?

Ответ: в этом типе надо выбрать один из методов среди свойств с параметрами и сделать его свойством по умолчанию, применив к самому классу экземпляр System.Reflection.DefaultMemberAttribute. Кстати, DefaultMemberAttribute можно применять к классам, структурам или интерфейсам. В С# при компиляции типа, определяющего свойства с параметрами, компилятор автоматически применяет к определяющему типу экземпляр DefaultMemberAttribute и учитывает его при использовании DefaultMemberAttribute. Конструктор этого атрибута задает имя, которое будет назначено свойству с параметрами, выбранному как свойство по умолчанию для этого типа.

Итак, в случае типа С#, у которого определено свойство с параметрами, но нет атрибута IndexerName, атрибут DefaultMember, задающий определяющий тип,

будет указывать имя `Item`. Если применить к свойству с параметрами атрибут `IndexerName`, то атрибут `DefaultMember` определяющего типа будет указывать на строку, заданную атрибутом `IndexerName`. Помните: C# не будет компилировать код, содержащий свойства с параметрами, если у них разные имена.

В программах на языке, поддерживающем несколько свойств с параметрами, нужно выбрать одно свойство-метод и пометить определяющий его тип атрибутом `DefaultMember`. Это будет единственное свойство с параметрами, доступное программам на C#.

Производительность при вызове методов-аксессоров свойств

В случае простых методов доступа `get` и `set` JIT-компилятор *подставляет* (inlines) код аксессора внутрь кода вызываемого метода, поэтому характерного снижения производительности работы программы, проявляющегося при использовании свойств вместо полей, не наблюдается. Подстановка подразумевает компиляцию кода метода (или, в данном случае, аксессора) непосредственно вместе с кодом вызывающего метода. Это избавляет от дополнительной нагрузки, связанной с вызовом во время выполнения, но за счет «распухания» кода скомпилированного метода. Поскольку аксессоры свойств обычно содержат мало кода, их подстановка может приводить к сокращению общего объема машинного кода, а значит, к повышению скорости выполнения.

Заметьте, что при отладке JIT-компилятор не подставляет свойства-методы, потому что подставленный код сложнее отлаживать. Это означает, что эффективность доступа к свойству в готовой версии программы выше, чем в отладочной. Что же касается полей, то скорость доступа к ним одинакова в обеих версиях.

Доступность аксессоров свойств

Иногда при проектировании типа требуется назначить аксессорам `get` и `set` разный уровень доступа. Чаще всего применяют открытый аксессор `get` и закрытый аксессор `set`:

```
public class SomeType {  
    private String m_name;  
    public String Name {  
        get { return m_name; }  
        protected set {m_name = value; }  
    }  
}
```

Как видно из кода, свойство `Name` объявлено как `public`, а это означает, что аксессор `get` будет открытым и доступным для вызова из любого кода. Однако следует заметить, что аксессор `set` объявлен как `protected`, то есть он доступен для вызова только из кода `SomeType` или кода класса, производного от `SomeType`.

При определении для свойства аксессоров с различным уровнем доступа синтаксис `C#` требует, чтобы само свойство было объявлено с наименее строгим уровнем доступа, а более строгое ограничение было наложено только на один из методов доступа. В этом примере свойство является открытым, а аксессор `set` — закрытым (более ограниченным).

Обобщенные методы-аксессоры свойств

Поскольку свойства фактически представляют собой методы, а `C#` и `CLR` поддерживают обобщение методов, некоторые разработчики пытаются определить обобщенные методы-аксессоры свойств. Однако `C#` не позволяет этого делать. Главная причина в том, что обобщенные свойства лишены смысла с концептуальной точки зрения. Предполагается, что свойство представляет характеристику объекта, которую можно извлечь или определить. Добавление обобщенного параметра типа означало бы, что поведение операции извлечения/определения может меняться, но, в принципе, от свойства не ожидается никакого поведения. Если нужно задать какое-либо поведение объекта — обобщенное или нет, — нужно создать метод, а не свойство.

Глава 11. События

Предмет этой главы — последний вид членов, которые можно определить в типе, — события. Если в типе определен член-событие, то этот тип (или его экземпляр) может уведомлять другие объекты о некоторых особых ситуациях, которые могут случиться. Например, если в классе `Button` (кнопка) определить событие `Click` (щелчок), то в приложение можно использовать объекты, которые будут получать уведомление о щелчке объекта `Button`, а получив такое уведомление — исполнять некоторые действия. События — это члены типа, обеспечивающие такого рода взаимодействие. Тип, в котором определены события, как минимум поддерживает:

- ❑ регистрацию статического метода типа или экземплярного метода объекта, заинтересованного в получении уведомления о событии;
- ❑ отмену регистрации статического метода типа или экземплярного метода объекта, получающего уведомления о событии;
- ❑ уведомление зарегистрированных методов о том, что событие произошло.

Типы могут предоставлять эту функциональность при определении событий, так как они поддерживают список зарегистрированных методов. Когда событие происходит, тип уведомляет об этом все зарегистрированные методы.

Модель событий CLR основана на *делегатах* (`delegate`). Делегаты позволяют обращаться к методам обратного вызова, не нарушая безопасности типов. Метод обратного вызова (`callback method`) — это механизм, позволяющий объекту получать уведомления, на которые он подписался. В этой главе мы будем постоянно пользоваться делегатами, но их детальный разбор отложим до главы 17.

Чтобы помочь вам досконально разобраться в работе событий в CLR, я начну с примера ситуации, в которой могут быть полезны события. Допустим, нам нужно создать почтовое приложение. Получив сообщение по электронной почте, пользователь может изъявить желание переслать его по факсу или переправить на пейджер. Допустим, вы начали проектирование приложения с разработки типа `MailManager`, получающего входящие сообщения. Тип `MailManager` будет поддерживать событие `NewMail`. Другие типы (например, `Fax` или `Pager`) могут регистрироваться для получения уведомления об этом событии. Когда тип `MailManager` получит новое сообщение, возникнет событие, в результате чего сообщение будет передано всем зарегистрированным объектам. Далее каждый объект обрабатывает сообщение в соответствии с собственной логикой.

Пусть во время инициализации приложения создается только один экземпляр MailManager и любое число объектов Fax и Pager. На рис. 11.1 показано, как инициализируется приложение и что происходит при получении сообщения.



Рис. 11.1. Архитектура приложения, в котором используются события

Это приложение работает следующим образом. При его инициализации создается экземпляр объекта MailManager, поддерживающего событие NewMail. Во время создания объекты Fax и Pager регистрируются в качестве получателей уведомлений о событии NewMail (приход нового сообщения) объекта MailManager, в результате MailManager «знает», что эти объекты следует уведомить о наличии нового сообщения. Если в дальнейшем MailManager получит новое сообщение, это приведет к вызову события NewMail, позволяющего всем зарегистрировавшимся объектам выполнить требуемую обработку нового сообщения.

Разработка типа, поддерживающего событие

Создание типа, поддерживающего одно или более событий, требует от разработчика пройти несколько этапов. В этом разделе я расскажу о каждом из них. Наше приложение MailManager (его можно скачать с сайта <http://wintellect.com>) содержит весь необходимый код типов MailManager, Fax и Pager. Как вы заметите, типы Fax и Pager практически идентичны.

Этап 1. Определение типа для хранения всей дополнительной информации, передаваемой получателям уведомления о событии

При возникновении события объект, в котором оно возникло, должен передать дополнительную информацию объектам-получателям уведомления о событии. Для предоставления получателям эту информацию нужно инкапсулировать в собственный класс, содержащий набор закрытых полей и набор открытых неизменяемых (только для чтения) свойств. В соответствии с соглашением, классы, содержащие информацию о событиях, передаваемую обработчику события, должны наследовать от типа `System.EventArgs`, а имя типа должно заканчиваться словом `EventArgs`. В этом примере у типа `NewMailEventArgs` есть поля, идентифицирующие отправителя сообщения (`m_from`), его получателя (`m_to`) и тему (`m_subject`).

```
// Этап 1. Определение типа для хранения информации.  
// которая передается получателям уведомления о событии  
internal class NewMailEventArgs : EventArgs {  
  
    private readonly String m_from, m_to, m_subject;  
  
    public NewMailEventArgs(String from, String to, String subject) {  
        m_from = from; m_to = to; m_subject = subject;  
    }  
  
    public String From { get { return m_from; } }  
    public String To { get { return m_to; } }  
    public String Subject { get { return m_subject; } }  
}
```

ПРИМЕЧАНИЕ

Тип `EventArgs` определяется в библиотеке классов `.NET Framework Class Library (FCL)` и выглядит примерно следующим образом:

```
[ComVisible(true)]  
[Serializable]  
public class EventArgs {  
    public static readonly EventArgs Empty = new EventArgs();  
    public EventArgs() { }  
}
```

Как видите, в нем нет ничего особенного. Он просто служит базовым типом, от которого можно порождать другие типы. С большинством событий не передается дополнительной информации. Например, в случае уведомления объектом `Button` о щелчке на кнопке, обращение к методу обратного вызова — и есть вся нужная информация. Определяя событие, не передающее дополнительные данные, можно не создавать новый объект `EventArgs`, достаточно просто воспользоваться свойством `EventArgs.Empty`.

Этап 2. Определение члена-события

В C# член-событие объявляется с ключевым словом `event`. Каждому члену-событию назначаются область действия (практически всегда он открытый, поэтому доступен из любого кода), тип делегата, указывающий на прототип вызываемого метода (или методов), и имя (любой допустимый идентификатор). Вот как выглядит член-событие нашего класса `NewMail`:

```
internal class MailManager {  
  
    // Этап 2. Определение члена-события  
    public event EventHandler<NewMailEventArgs> NewMail;  
  
    ...  
}
```

Здесь `NewMail` — имя события, а типом члена-события является `EventHandler<NewMailEventArgs>`. Это означает, что получатели уведомления о событии должны предоставлять метод обратного вызова, прототип которого совпадает с типом-делегатом `EventHandler<NewMailEventArgs>`. Обобщенный делегат `System.EventHandler` определен следующим образом:

```
public delegate void EventHandler<TEventArgs>  
(Object sender, TEventArgs e) where TEventArgs: EventArgs;
```

Поэтому прототип метода должен выглядеть так:

```
void MethodName(Object sender, NewMailEventArgs e);
```

ПРИМЕЧАНИЕ

Многих удивляет, почему механизм событий требует, чтобы параметр `sender` имел тип `Object`. Вообще-то, поскольку `MailManager` — единственный тип, реализующий события с объектом `NewMailEventArgs`, было бы разумнее использовать следующий прототип метода обратного вызова:

```
void MethodName(MailManager sender, NewMailEventArgs e);
```

Причиной того, что параметр `sender` имеет тип `Object`, является наследование. Что произойдет, если `MailManager` задействовать в качестве базового класса для создания класса `SmtpMailManager`? В методе обратного вызова придется в прототипе задать параметр `sender` как `SmtpMailManager`, а не `MailManager`, но этого делать нельзя, так как тип `SmtpMailManager` просто наследует событие `NewMail`. Поэтому код, ожидающий от `SmtpMailManager` информацию о событии, все равно будет вынужден приводить аргумент `sender` к типу `SmtpMailManager`. Иначе говоря, приведение все равно необходимо, поэтому проще всего сделать так, чтобы параметр `sender` имел тип `Object`.

Еще одна причина того, что `sender` относят к типу `Object` — гибкость, поскольку делегат может применяться несколькими типами, которые поддерживают событие, передающее объект `NewMailEventArgs`. В частности, класс

PopMailManager мог бы использовать делегат, даже если бы не наследовал от класса MailManager.

И еще одно: механизм событий требует, чтобы в имени делегата и методе обратного вызова производный от EventArgs параметр назывался «e». Единственная причина — обеспечить единообразие, облегчая и упрощая для разработчиков изучение и реализацию событий. Инструменты создания кода (например, такой как Microsoft Visual Studio) также «знают», что нужно вызывать параметр e.

И последнее, механизм событий требует, чтобы все обработчики возвращали void. Это обязательно, потому что при возникновении события могут выполняться несколько методов обратного вызова и невозможно получить у них все возвращаемое значение. Тип void просто запрещает методам возвращать какое бы то ни было значение. К сожалению, в библиотеке FCL есть обработчики событий, в частности Resolve-EventHandler, в которых Microsoft не следует собственным правилам и возвращает объект типа Assembly.

Этап 3. Определение метода, ответственного за уведомление зарегистрированных объектов о событии

В соответствии с соглашением в классе должен быть виртуальный защищенный метод, вызываемый из кода класса и его потомков при возникновении события. Этот метод принимает один параметр, объект MailMsgEventArgs, содержащий дополнительные сведения о событии. Реализация по умолчанию этого метода просто проверяет, есть ли объекты, зарегистрировавшиеся для получения уведомления о событии, и при положительном результате проверки сообщает зарегистрированным методам о возникновении события. Вот как выглядит этот метод в нашем классе MailManager:

```
internal class MailManager {
    ...
    // Этап 3. Определение метода, ответственного за уведомление
    // зарегистрированных объектов о событии
    // Если этот класс изолированный, нужно сделать метод закрытым
    // или неvirtуальным
    protected virtual void OnNewMail(NewMailEventArgs e) {
        // Сохранить поле делегата во временном поле
        // для обеспечения безопасности потоков
        EventHandler<NewMailEventArgs> temp = NewMail;
        // Если есть объекты, зарегистрированные для получения
        // уведомления о событии, уведомляем их
        if (temp != null) temp(this, e);
    }
    ...
}
```

Поднятие события безопасным в отношении потоков образом

В первой поставке .NET Framework рекомендуемым способом поднятия события был такой:

```
// Версия 1
protected virtual void OnNewMail(NewMailEventArgs e) {
    if (NewMail != null) NewMail(this, e);
}
```

Проблема метода `OnNewMail` состоит в следующем. Поток может видеть, что значение `NewMail` не равно `null`, однако перед вызовом `NewMail` другой поток может удалить делегата из цепочки, присвоив `NewMail` значение `null`. В результате будет вброшено исключение `NullReferenceException`. Для решения проблемы многие разработчики пишут следующий код:

```
// Версия 2
protected void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp = NewMail;
    if (temp != null) temp(this, e);
}
```

Идея здесь в том, что ссылка на `NewMail` копируется во временную переменную `temp`, которая ссылается на цепочку делегатов в момент назначения. Этот метод сравнивает `temp` со значением `null` и вызывает `temp`, поэтому уже не имеет значения, поменял ли другой поток `NewMail` после назначения `temp`. Вспомните, что делегаты неизменяемы, и именно поэтому этот способ работает. Однако многие разработчики не знают, что компилятор может оптимизировать этот программный код, удалив переменную `temp`. В этом случае обе представленные версии кода окажутся идентичными, в результате опять-таки возможно исключение `NullReferenceException`.

Для реального решения этой проблемы необходимо переписать `OnNewMail` так:

```
// Версия 3
protected void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp = Thread.VolatileRead(ref NewMail);
    if (temp != null) temp(this, e);
}
```

Вызов `VolatileRead` заставляет считывать `NewMail` в точке вызова и именно в этот момент копировать ссылку в переменную `temp`. Затем переменная `temp` вызывается лишь тогда, когда она не равна `null`. К несчастью, из-за отсутствия перегруженной версии обобщенного метода `VolatileRead` невозможно записать программный код в этом виде. Однако существует обобщенная перегруженная версия `Interlocked.CompareExchange`, которую вы можете использовать:

```
// Версия 4
protected void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp =
        Interlocked.CompareExchange(ref NewMail, null, null);
    if (temp != null) temp(this, e);
}
```

продолжение ➤

Здесь CompareExchange изменяет ссылку temp на null, если значение NewMail равно null, и не трогает ее, если NewMail не равно null. Другими словами, CompareExchange не изменяет значение NewMail вообще, но возвращает значение внутри NewMail безопасным в отношении потоков образом. Для более детального изучения методов Thread.VolatileRead и Interlocked.CompareExchange см. главу 28.

И хотя версия 4 этого программного кода является наилучшей и технически корректной, вы также можете использовать версию 2 с JIT-компилятором, не опасаясь за последствия, так как он не будет оптимизировать программный код. Однако этого нет в официальной документации, то есть гипотетически он может внести в код свои изменения, поэтому все же лучше воспользоваться версией 4 представленного программного кода.

Важно отметить, что из-за конкуренции потоков возможна ситуация, когда метод вызывается уже после того, как он удален из цепочки делегатов события.

Для удобства вы можете определить расширенный метод (см. главу 8), инкапсулирующий безопасную в отношении потоков логику. Определите расширенный метод следующим образом:

```
public static class EventArgsExtensions {  
    public static void Raise<TEventArgs>(this TEventArgs e,  
        Object sender, ref EventHandler<TEventArgs> eventDelegate)  
        where TEventArgs : EventArgs {  
        // Копирование ссылки на поле делегата во временное поле  
        // для безопасности в отношении потоков  
        EventHandler<TEventArgs> temp =  
            Interlocked.CompareExchange(ref eventDelegate, null, null);  
        // Если зарегистрированный метод заинтересован в событии, уведомите его  
        if (temp != null) temp(sender, e);  
    }  
}
```

Теперь можно переписать метод OnNewMail следующим образом:

```
protected virtual void OnNewMail(NewMailEventArgs e) {  
    e.Raise(this, ref m_NewMail);  
}
```

Тип, производный от MailManager, может свободно переопределять метод OnNewMail, что позволяет производному типу контролировать срабатывание события. Таким образом, производный тип может обрабатывать новые сообщения любым способом по собственному усмотрению. Обычно производный тип вызывает метод OnNewMail базового типа, в результате зарегистрированный объект получает уведомление. Однако производный тип может и отказаться от переделки уведомления о событии.

Этап 4. Определение метода, транслирующего входную информацию в желаемое событие

У класса должен быть метод, принимающий некоторую входную информацию и в ответ генерирующий событие. В примере с типом `MailManager` метод `SimulateNewMail` вызывается для оповещения о получении нового сообщения в `MailManager`:

```
internal class MailManager {  
  
    // Этап 4. Определение метода, транслирующего входную  
    // информацию в желаемое событие  
    public void SimulateNewMail(String from, String to, String subject) {  
  
        // Создать объект для хранения информации, которую  
        // нужно передать получателям уведомления  
        NewMailEventArgs e = new NewMailEventArgs(from, to, subject);  
  
        // Вызвать виртуальный метод, уведомляющий объект о событии  
        // Если ни один из производных типов не переопределяет этот метод.  
        // объект уведомит всех зарегистрированных получателей уведомления  
        OnNewMail(e);  
    }  
}
```

Метод `SimulateNewMail` принимает информацию о сообщении и создает новый объект `NewMailEventArgs`, передавая его конструктору данные сообщения. Затем вызывается `OnNewMail` — собственный виртуальный метод объекта `MailManager`, чтобы формально уведомить объект `MailManager` о новом почтовом сообщении. Обычно это вызывает генерацию события, в результате уведомляются все зарегистрированные объекты. (Как уже отмечалось, тип, производный от `MailManager`, может переопределять это действие.)

Как реализуются события

Научившись определять класс с членом-событием, можно поближе познакомиться с самим событием и узнать, как оно работает. В классе `MailManager` есть строчка кода, определяющая сам член-событие:

```
public event EventHandler<NewMailEventArgs> NewMail;
```

При компиляции этой строки компилятор превращает ее в следующие три конструкции:

```
// 1. ЗАКРЫТОЕ поле делегата, инициализированное значением null  
private EventHandler<NewMailEventArgs> NewMail = null;
```

продолжение ➤

```
// 2. ОТКРЫТЫЙ метод add_Xxx (где Xxx - это имя события)
// Позволяет объектам регистрироваться для получения уведомлений о событии
public void add_NewMail(EventHandler<NewMailEventArgs> value) {
    // Цикл и вызов CompareExchange - фантастический способ добавления
    // делегата в событие безопасным в отношении потоков путем
    EventHandler<NewMailEventArgs> prevHandler;
    EventHandler<NewMailEventArgs> newMail = this.NewMail;
    do {
        prevHandler = newMail;
        EventHandler<NewMailEventArgs> newHandler =
            (EventHandler<NewMailEventArgs>) Delegate.Combine(prevHandler, value);
        newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
            ref this.NewMail, newHandler, prevHandler);
    } while (newMail != prevHandler);
}

// 3. ОТКРЫТЫЙ метод remove_Xxx (где Xxx - это имя события)
// Позволяет объектам отменять регистрацию в качестве
// получателей уведомлений о событии
public void remove_NewMail(EventHandler<NewMailEventArgs> value) {
    // Цикл и вызов CompareExchange - фантастический способ
    // удаления делегата из события безопасным в отношении потоков путем
    EventHandler<NewMailEventArgs> prevHandler;
    EventHandler<NewMailEventArgs> newMail = this.NewMail;
    do {
        prevHandler = newMail;
        EventHandler<NewMailEventArgs> newHandler =
            (EventHandler<NewMailEventArgs>) Delegate.Remove(prevHandler, value);
        newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
            ref this.NewMail, newHandler, prevHandler);
    } while (newMail != prevHandler);
}
```

Первая конструкция — просто поле соответствующего типа делегата. Оно содержит ссылку на заголовок списка делегатов, которые будут уведомляться при возникновении события. Поле инициализируется значением `null`; это означает, что нет получателей, ожидающих уведомления о событии. Когда метод регистрирует получателя уведомления, это поле начинает ссылаться на экземпляр делегата `EventHandler<NewMailEventArgs>`, который может, в свою очередь, ссылаться на дополнительные делегаты `EventHandler<NewMailEventArgs>`. Когда получатель регистрируется для получения уведомления о событии, он просто добавляет в список экземпляры типа делегата. Ясно, что отказ от регистрации означает удаление соответствующего делегата.

Обратите внимание: в примере поле делегата, `NewMail`, всегда закрытое, несмотря на то что исходная строка кода определяет событие как открытое. Причина — необходимость предотвратить доступ из кода, не относящегося к определяющему классу. Если бы поле было открытым, любой код мог бы

изменить значение поля, в том числе удалить все делегаты, подписавшиеся на событие.

Вторая создаваемая компилятором C# конструкция — метод, позволяющий другим объектам регистрироваться в качестве получателей уведомления о событии. Компилятор C# автоматически присваивает этой функции имя, добавляя приставку `add_` к имени события (`NewMail`). Компилятор C# также автоматически генерирует код метода, который всегда вызывает статический метод `Combine` типа `System.Delegate`. Метод `Combine` добавляет в список делегатов новый экземпляр и возвращает новый заголовок списка, который снова сохраняется в поле.

Третья и последняя создаваемая компилятором C# конструкция представляет собой метод, позволяющий объекту отказаться от подписки на событие. И этой функции компилятор C# присваивает имя автоматически, добавляя приставку `remove_` к имени события (`NewMail`). Код метода всегда вызывает метод `Remove` типа `System.Delegate`. Последний метод удаляет делегат из списка и возвращает новый заголовок списка, который сохраняется в поле.

ВНИМАНИЕ

Если вы попытаетесь удалить метод, который никогда не добавлялся, то метод `Delegate.Remove` ничего не сделает. Не появится ни исключения, ни предупреждения, а коллекция методов событий останется без изменений.

ПРИМЕЧАНИЕ

Оба метода — `add` и `remove` — используют хорошо известный эталон для модификации значения защищенным в отношении потоков способом. Этот эталон описывается в главе 28.

В этом примере методы `add` и `remove` являются открытыми, поскольку в соответствующей строке исходного кода событие изначально объявлено как открытое. Если бы оно было объявлено как закрытое, то методы `add` и `remove`, сгенерированные компилятором, тоже были бы объявлены как закрытые. Так что когда в типе определяется событие, модификатор доступа события указывает, какой код способен регистрироваться и отменять регистрацию для уведомления о событии, но прямым доступом к полю делегата обладает только сам тип. Члены-события также могут объявляться статическими и виртуальными; в этом случае сгенерированные компилятором методы `add` и `remove` также будут статическими или виртуальными соответственно.

Помимо генерации этих трех конструкций, компиляторы генерируют запись с определением события и помещают ее в метаданные управляемого модуля. Эта запись содержит ряд флагов и базовый тип-делегат, а также ссылки на методы-аксессуары `add` и `remove`. Эта информация нужна просто для того, чтобы

очертить связь между абстрактным понятием «событие» и его методами-аксессорами. Эти метаданные могут использовать компиляторы и другие инструменты, и, конечно же, эти сведения можно получить при помощи класса `System.Reflection.EventInfo`. Однако сама среда CLR эти метаданные не использует и во время выполнения требует лишь методы-аксессоры.

Создание типа, отслеживающего событие

Ну, самое трудное позади. В этом разделе я показываю, как определить тип, использующий событие, поддерживаемое другим типом. Начнем с изучения исходного текста типа `Fax`:

```
internal sealed class Fax {
    // Передаем конструктору объект MailManager
    public Fax(MailManager mm) {

        // Создаем экземпляр делегата EventHandler<NewMailEventArgs>,
        // ссылающийся на метод обратного вызова FaxMsg
        // Регистрируем обратный вызов для события NewMail объекта MailManager
        mm.NewMail += FaxMsg;
    }

    // MailManager вызывает этот метод для уведомления
    // объекта Fax о прибытии нового почтового сообщения
    private void FaxMsg(Object sender, NewMailEventArgs e) {

        // 'sender' можно использовать для взаимодействия с объектом MailManager,
        // если нужно вернуть ему какую-то информацию

        // 'e' указывает дополнительную информацию о событии,
        // которую пожелает предоставить MailManager

        // Обычно расположенный здесь код отправляет сообщение по факсу
        // В тестовом варианте программы этот метод
        // выводит информацию на консоль
        Console.WriteLine("Faxing mail message:");
        Console.WriteLine(" From={0}, To={1}, Subject={2}",
            e.From, e.To, e.Subject);
    }

    // Этот метод может выполняться для отмены регистрации объекта Fax
    // в качестве получателя уведомлений о событии NewMail
    public void Unregister(MailManager mm) {
```

```
// Отменить регистрацию на уведомление о событии NewMail объекта  
MailManager. mm.NewMail -= FaxMsg;  
}  
}
```

При инициализации почтовое приложение сначала создает объект MailManager и сохраняет ссылку на него в переменной. Затем оно создает объект Fax, передавая ссылку на MailManager как параметр. После создания делегата объект Fax регистрируется при помощи оператора += языка C# для уведомления о событии NewMail объекта MailManager:

```
mm.NewMail += FaxMsg;
```

Обладая встроенной поддержкой событий, компилятор C# транслирует оператор += в код, регистрирующий объект для получения уведомлений о событии:

```
mm.add_NewMail(new EventHandler<NewMailEventArgs>(this.FaxMsg));
```

Как видите, компилятор C# создает код, конструирующий делегат EventHandler<NewMailEventArgs>, который инкапсулирует метод NewMail класса Fax. Затем компилятор C# вызывает метод add_NewMail объекта MailManager, передавая ему новый делегат. Ясно, что вы можете убедиться в этом, скомпилировав код и затем изучив IL-код с помощью такого инструмента, как утилита ILDasm.exe.

Даже используя язык, не поддерживающий события напрямую, можно зарегистрировать делегат для уведомления о событии, явно вызвав метод-аксессор add. Результат идентичен, только исходный текст получается не столь изящным. Именно метод add, регистрирующий делегат для уведомления о событии, добавляет делегат в список делегатов данного события.

Когда срабатывает событие объекта MailManager, вызывается метод FaxMsg объекта Fax. Этому методу передается ссылка на объект MailManager в качестве первого параметра, или sender. Чаще всего этот параметр игнорируется, но он может и использоваться, если в ответ на уведомление о событии объект Fax пожелает получить доступ к полям или методам объекта MailManager. Второй параметр — это ссылка на объект NewMailEventArgs. Этот объект содержит всю дополнительную информацию, которая, по мнению NewMailEventArgs, может быть полезной для получателей события.

При помощи объекта NewMailEventArgs метод FaxMsg может без труда получить доступ к сведениям об отправителе и получателе сообщения, его теме и собственно тексту. Реальный объект Fax отправлял бы эти сведения адресату, но в данном примере они просто выводятся на консоль.

Когда объекту больше не нужны уведомления о событиях, он должен отменить свою регистрацию. Например, объект Fax отменит свою регистрацию в качестве получателя уведомления о событии NewMail, если пользователю больше не нужно пересылать сообщения электронной почты по факсу. Пока объект зарегистрирован в качестве получателя уведомления о событии другого

объекта, он не может попасть в сферу действия сборщика мусора. Если в вашем типе реализован метод `Dispose` объекта `IDisposable`, уничтожение объекта должно вызвать отмену его регистрации в качестве получателя уведомлений обо всех событиях (об объекте `IDisposable` см. также главу 21).

Код, иллюстрирующий отмену регистрации, показан в исходном тексте метода `Unregister` объекта `Fax`. Код этого метода фактически идентичен конструктору типа `Fax`. Единственное отличие в том, что здесь вместо `+=` использован оператор `-=`. Обнаружив код, отменяющий регистрацию делегата при помощи оператора `-=`, компилятор `C#` генерирует вызов метода `remove` этого события:

```
mm.remove_NewMail(new EventHandler<NewMailEventArgs>(FaxMsg));
```

Как и в случае оператора `+=`, даже при использовании языка, не поддерживающего события напрямую, можно отменить регистрацию делегата, явно вызывая метод-аксессор `remove`, который отменяет регистрацию делегата путем сканирования списка в поисках делегата-оболочки метода, соответствующего переданному методу обратного вызова. Если обнаружено совпадение, делегат удаляется из списка делегатов события. Если нет, ошибка не возникает, и список делегатов события остается неизменным.

Кстати, `C#` требует, чтобы для добавления и удаления делегатов из списка в ваших программах использовались операторы `+=` и `-=`. Если попытаться напрямую обратиться к методам `add` или `remove`, компилятор `C#` сгенерирует сообщение об ошибке (`CS0571`: оператор или аксессор нельзя вызывать явно):

```
CS0571: cannot explicitly call operator or accessor
```

Явное управление регистрацией событий

В типе `System.Windows.Forms.Control` определено около 70 событий. Если тип `Control` реализует события, позволяя компилятору явно генерировать методы аксессора `add` и `remove` и поля-делегаты, то каждый объект `Control` будет иметь 70 полей-делегатов для каждого события! Многие программисты оперируют несколькими событиями, следовательно, для каждого объекта, созданного из унаследованного от `Control` типа, потребуется огромный объем памяти. Кстати, типы `System.Web.UI.Control` (из `ASP.NET`) и `System.Windows.UIElement` (из `Windows Presentation Foundation, WPF`) также предлагают множество событий, которые большинство программистов не использует.

В этом разделе рассказано о том, каким образом компилятор `C#` позволяет разработчикам явно управлять регистрацией событий, контролировать добавление и удаление методов, манипулировать делегатами обратных вызовов.

Для эффективного использования делегатов события каждый объект, применяющий события, содержится в главной коллекции (обычно это словарь) с несколькими типами идентификаторов событий в качестве ключей и списком делегатов в качестве значений. При создании нового объекта эта коллекция пуста. При регистрации события идентификатор события ищется в коллекции. Если идентификатор события найден, то новый делегат добавляется в список делегатов для этого события. Если идентификатор события не найден, то он добавляется к делегатам. Когда наступает событие, идентификатор события ищется в коллекции. Если в коллекции нет записи для него, то оно не регистрируется, и делегаты не вызываются. Если идентификатор события находится в коллекции, то вызывается список делегатов, ассоциированных с этим идентификатором события.

Реализация этого эталона проектирования находится в зоне ответственности разработчика, проектирующего типы, определяющие события. Разработчик, использующий тип, обычно не знает о том, как события реализованы. Приведу пример того, как вы можете усовершенствовать данный эталон. Я реализовал класс `EventSet`, представляющий коллекцию событий и каждого делегата события следующим образом:

```
using System;
using System.Collections.Generic;

// Этот класс нужен для поддержания безопасности типа
// и кода при использовании EventSet
public sealed class EventKey : Object { }

public sealed class EventSet {
    // Закрытый словарь служит для отображения EventKey -> Delegate
    private readonly Dictionary<EventKey, Delegate> m_events =
        new Dictionary<EventKey, Delegate>();

    // Добавление отображения EventKey -> Delegate, если его не существует
    // И компоновка делегата с существующим ключом EventKey
    public void Add(EventKey eventKey, Delegate handler) {
        Monitor.Enter(m_events);
        Delegate d;
        m_events.TryGetValue(eventKey, out d);
        m_events[eventKey] = Delegate.Combine(d, handler);
        Monitor.Exit(m_events);
    }

    // Удаление делегата из EventKey (если он существует)
    // и ликвидация отображения EventKey -> Delegate.
    // когда удален последний делегат
    public void Remove(EventKey eventKey, Delegate handler) {
```

```

Monitor.Enter(m_events);
// Вызов TryGetValue, чтобы исключение не было
// выброшено при попытке удалить делегата
// из неустановленного ключа EventKey
Delegate d;
if (m_events.TryGetValue(eventKey, out d)) {
d = Delegate.Remove(d, handler);
// Если делегат остается, то установить новый ключ EventKey,
// иначе - удалить EventKey
if (d != null) m_events[eventKey] = d;
else m_events.Remove(eventKey);
}
Monitor.Exit(m_events);
}

// Поднятие события для обозначенного ключа EventKey
public void Raise(EventKey eventKey, Object sender, EventArgs e) {
// Не вбрасывать исключение, если ключ EventKey не назначен
Delegate d;
Monitor.Enter(m_events);
m_events.TryGetValue(eventKey, out d);
Monitor.Exit(m_events);

if (d != null) {
// Из-за того что каталог может содержать несколько разных типов
// делегатов, невозможно создать вызов защищенного типа делегата
// во время компиляции. Я вызвал метод DynamicInvoke типа
// System.Delegate, передав в метод обратного вызова параметры в виде
// массива объектов. DynamicInvoke будет контролировать безопасность
// типов параметров метода обратного вызова и вызов этого метода. Если
// будет найдено несоответствие типов, вбрасывается исключение
d.DynamicInvoke(new Object[] { sender, e });
}
}

```

ПРИМЕЧАНИЕ

FCL определяет тип `System.ComponentModel.EventHandlerList`, который, по существу, делает то же самое, что и класс `EventSet`. Типы `System.Windows.Forms.Control` и `System.Web.UI.Control` используют тип `EventHandlerList` для поддержания набора редких событий. Конечно, вы можете задействовать тип `EventHandlerList` из библиотеки FCL. Разница между типом `EventHandlerList` и моим типом `EventSet` заключается в том, что `EventHandlerList` применяет связанный список вместо хэш-таблицы. Это означает, что доступ к элементам, управляемый при помощи `EventHandlerList`, медленнее, чем доступ при помощи `EventSet`. К тому же `EventHandlerList` не предлагает безопасный в отношении потоков способ для доступа к событиям, при необходимости вы можете реализовать собственный безопасный в отношении потоков обработчик при помощи коллекции `EventHandlerList`.

Я продемонстрирую класс, который использует класс `EventSet`. Этот класс имеет поле, ссылающееся на объект `EventSet`, и каждое событие из этого класса реализуется явно таким образом, что каждый метод `add` содержит специального делегата обратного вызова в объекте `EventSet`, а каждый метод `remove` уничтожает специального делегата обратного вызова (если найдет его).

```
using System;
```

```
// Определение типа, унаследованного от EventArgs для этого события
public class FooEventArgs : EventArgs { }

public class TypeWithLotsOfEvents {

    // Определение закрытого экземплярного поля, ссылающегося на коллекцию
    // Коллекция управляет множеством пар "Event/Delegate"
    // Примечание: Тип EventSet не является частью FCL,
    // это мой собственный тип
    private readonly EventSet m_eventSet = newEventSet();

    // Защищенное свойство позволяет унаследовать тип доступа к коллекции
    protected EventSet EventSet { get { return m_eventSet; } }

    #region Code to support the Foo event (
        repeat this pattern for additional events)
    // Определение членов необходимо для события Foo
    // 2a. Создайте статический неизменяемый (только для чтения) объект
    // для идентификации события
    // Каждый объект имеет свой хэш-код для нахождения связанного списка
    // делегата события в коллекции объекта
    protected static readonly EventKey s_fooEventKey = newEventKey();

    // 2d. Определите для события метод-аксессор, который добавляет
    // делегата в коллекцию и удаляет его из коллекции
    public event EventHandler<FooEventArgs> Foo {
        add { m_eventSet.Add(s_fooEventKey, value); }
        remove { m_eventSet.Remove(s_fooEventKey, value); }
    }

    // 2e. Определите защищенный виртуальный метод On для этого события
    protected virtual void OnFoo(FooEventArgs e) {
        m_eventSet.Raise(s_fooEventKey, this, e);
    }

    // 2f. Определите метод, осуществляющий ввод этого события
    public void SimulateFoo() {OnFoo(newFooEventArgs());}
    #endregion
}
```

Программный код, использующий тип `upewWithLotsOfEvents`, не может сказать, было ли событие реализовано неявно компилятором или явно разработчиком. Он просто регистрирует события при помощи обычного синтаксиса. Вот программный код, демонстрирующий это:

```
public sealed class Program {  
    public static void Main() {  
        TypeWithLotsOfEvents twle = newTypeWithLotsOfEvents();  
  
        // Добавьте обратный вызов  
        twle.Foo += HandleFooEvent;  
  
        // Проверьте, что это работает  
        twle.SimulateFoo();  
    }  
  
    private static void HandleFooEvent(object sender, FooEventArgs e) {  
        Console.WriteLine("Handling Foo Event here...");  
    }  
}
```

Глава 12. Обобщения

Разработчикам хорошо известны его достоинства объектно-ориентированного программирования. Одно из ключевых преимуществ — возможность многократно использовать код, то есть создавать производные классы, наследующие все возможности базового класса. В производном классе можно просто переопределить виртуальные методы или добавить новые методы, чтобы изменить унаследованные от базового класса характеристики для решения новых задач.

Обобщения (generics) — еще один механизм, поддерживаемый общезыковой исполняющей средой (CLR) и языками программирования. Этот механизм является новой формой многократного использования кода, а точнее — многократного использования алгоритма. По сути, разработчик описывает алгоритм, например, сортировки, поиска, замены, сравнения или преобразования, но не указывает типы данных, с которыми тот работает. Поэтому алгоритм может обобщенно применяться к объектам разных типов. Применяя готовый алгоритм, другой разработчик должен указать конкретные типы данных, например для алгоритма сортировки — `Int32s`, `String` и т. д., а для алгоритма сравнения — `DateTime`, `Versions` и т. д.

Большинство алгоритмов инкапсулировано в типе. CLR поддерживает создание как обобщенных ссылочных, так и обобщенных значимых типов, однако обобщенные перечислимые типы не поддерживаются. Кроме того, CLR позволяет создавать обобщенные интерфейсы и обобщенные делегаты. Иногда полезный алгоритм инкапсулирован в одном методе, поэтому CLR поддерживает создание обобщенных методов, определенных в ссылочном типе, в значимом типе или в интерфейсе.

В частности, в библиотеке FCL определен обобщенный алгоритм управления списками, работающий с набором объектов. Тип объектов в обобщенном алгоритме не указан. Используя такой алгоритм, нужно указывать конкретные типы данных.

FCL-класс, инкапсулирующий обобщенный алгоритм управления списками, называется `List<T>` и определен в пространстве имен `System.Collections.Generic`.

Исходный текст определения этого класса выглядит следующим образом (это сильно сокращенный вариант):

```
[Serializable]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
    IList, ICollection, IEnumerable {

    public List();
```

продолжение ➤

```
public void Add(T item);
public Int32 BinarySearch(T item);
public void Clear();
public Boolean Contains(T item);
public Int32 IndexOf(T item);
public Boolean Remove(T item);
public void Sort();
public void Sort(IComparer<T> comparer);
public void Sort(Comparison<T> comparison);
public T[] ToArray();
public Int32 Count { get; }
public T this[Int32 index] { get; set; }
}
```

Символами <T> сразу после имени класса автор обобщенного класса List указал, что класс работает с неопределенным типом данных. При определении обобщенного типа или метода переменные, указывающие на типы (например, T), называются *параметрами типа* (type parameters). T — это имя переменной, которое применяется в исходном тексте во всех местах, где используется соответствующий тип данных. Например, в определении класса List переменная T служит параметром (метод Add принимает параметр типа T) и возвращаемым значением (метод ToArray возвращает одномерный массив типа T) метода. Другой пример — метод-индексатор (в C# он называется this). У индексатора есть метод-аксессор get, возвращающий значение типа T, и метод-аксессор set, принимающий параметр типа T. Переменную T можно использовать в любом месте, где указывается тип данных, а значит, и при определении локальных переменных внутри метода или полей внутри типа.

ПРИМЕЧАНИЕ

Рекомендации проектирования Microsoft утверждают, что переменные параметров должны называться T или, в крайнем случае, начинаться с T (как, например, TKey или TValue). T означает тип (type), а I означает интерфейс (например, IComparable).

Итак, после определения обобщенного типа List<T> готовый обобщенный алгоритм могут использовать другие разработчики, просто указав точный тип данных, с которым должен работать этот алгоритм. В случае обобщенного типа или метода указанные типы данных называют *аргументами типа* (type arguments). Например, разработчик может использовать алгоритм List, указав тип DateTime в качестве аргумента типа:

```
private static void SomeMethod() {
    // Создание списка (List), работающего с объектами DateTime
    List<DateTime> dtList = new List<DateTime>();

    // Добавление объекта DateTime в список
    dtList.Add(DateTime.Now); // Без упаковки
}
```

```
// Добавление еще одного объекта DateTime в список
dtList.Add(DateTime.MinValue); // Без упаковки

// Попытка добавить объект типа String в список
dtList.Add("1/1/2004"); // Ошибка компиляции
// Извлечение объекта DateTime из списка
DateTime dt = dtList[0]; // Приведение типов не требуется
}
```

На примере этого кода видны главные преимущества обобщений для разработчиков.

- ❑ **Защита исходного кода.** Разработчику, использующему обобщенный алгоритм, не нужен доступ к исходному тексту алгоритма (при работе с шаблонами C++ или обобщениями Java разработчику требуется исходный текст алгоритма).
- ❑ **Безопасность типов.** Когда обобщенный алгоритм применяется с конкретным типом, компилятор и CLR понимают это и обеспечивают, чтобы в алгоритме использовались лишь объекты, совместимые с этим типом данных. Попытка использования объекта, не совместимого с указанным типом, приведет к ошибке на этапе компиляции или выполнения. В данном примере передача объекта String методу Add вызывает ошибку компиляции.
- ❑ **Более простой и понятный код.** Поскольку компилятор обеспечивает безопасность типов, в исходном тексте требуется меньше операция приведения типов, а такой код проще писать и поддерживать. В последней строке `SomeMethod` разработчику не нужно использовать приведение (`DateTime`), чтобы присвоить переменной `dt` результат вызова индексатора (при запросе элемента с индексом 0).
- ❑ **Повышение производительности.** До появления обобщений одним из способов задания обобщенного алгоритма было такое описание всех его членов, чтобы они по определению «умели» работать с типом данных `Object`. Чтобы этот алгоритм работал с экземплярами значимого типа, перед вызовом членов алгоритма среда CLR должна была упаковать этот экземпляр. Как показано в главе 5, упаковка требует выделения памяти в управляемой куче, что приводит к более частым процедурам сборки мусора, а это, в свою очередь, снижает производительность приложения. Поскольку отныне обобщенный алгоритм можно создать для работы с конкретным значимым типом, экземпляры значимого типа могут передаваться по значению и CLR не нужно выполнять упаковку. Операции приведения типа также не нужны (см. предыдущий пункт), поэтому CLR не нужно контролировать безопасность типов при их преобразовании, что также ускоряет работу кода.

Чтобы убедить вас в том, что обобщения повышают производительность, я написал программу для сравнения производительности необобщенного алгоритма `ArrayList` из библиотеки классов FCL и обобщенного алгоритма `List`.

Программа позволяет тестировать работу алгоритмов с объектами значимого и ссылочного типов:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

public static class Program {
    public static void Main() {
        ValueTypePerfTest();
        ReferenceTypePerfTest();
    }

    private static void ValueTypePerfTest() {
        const Int32 count = 10000000;
        using (new OperationTimer("List<Int32>")) {
            List<Int32> l = new List<Int32>(count);
            for (Int32 n = 0; n < count; n++) {
                l.Add(n);
                Int32 x = l[n];
            }
            l = null; // Это должно удаляться в процессе сборки мусора
        }

        using (new OperationTimer("ArrayList of Int32")) {
            ArrayList a = new ArrayList();
            for (Int32 n = 0; n < count; n++) {
                a.Add(n);
                Int32 x = (Int32) a[n];
            }
            a = null; // Это должно удаляться в процессе сборки мусора
        }
    }

    private static void ReferenceTypePerfTest() {
        const Int32 count = 10000000;
        using (new OperationTimer("List<String>")) {
            List<String> l = new List<String>();
            for (Int32 n = 0; n < count; n++) {
                l.Add("X");
                String x = l[n];
            }
            l = null; // Это должно удаляться в процессе сборки мусора
        }

        using (new OperationTimer("ArrayList of String")) {
            ArrayList a = new ArrayList();
```

```

    for (Int32 n = 0; n < count; n++) {
        a.Add("X");
        String x = (String) a[n];
    }
    a = null; // Это должно удаляться в процессе сборки мусора
}
}
}

```

```

// Это полезный способ оценки времени выполнения алгоритма
internal sealed class OperationTimer : IDisposable {
    private Int64 m_startTime;
    private String m_text;
    private Int32 m_collectionCount;

    public OperationTimer(String text) {
        PrepareForOperation();

        m_text = text;
        m_collectionCount = GC.CollectionCount(0);

        // Это выражение должно быть последним в этом методе.
        // чтобы обеспечить максимально точную оценку быстродействия
        m_startTime = Stopwatch.GetTimestamp();
    }

    public void Dispose() {
        Console.WriteLine("{0,6:###.00} seconds (GCs={1,3}) {2}",
            (Stopwatch.GetTimestamp() - m_startTime) /
            (Double) Stopwatch.Frequency,
            GC.CollectionCount(0) - m_collectionCount, m_text);
    }

    private static void PrepareForOperation() {
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();
    }
}

```

Скомпилировав эту программу в режиме **Release** (с включенной оптимизацией) и выполнив ее на своем компьютере, я получил следующий результат:

```

.10 seconds (GCs= 0) List<Int32>
2.02 seconds (GCs= 30) ArrayList of Int32
.52 seconds (GCs= 6) List<String>
.53 seconds (GCs= 6) ArrayList of String

```

Как видите, с типом `Int32` обобщенный алгоритм `List` работает гораздо быстрее, чем необобщенный алгоритм `ArrayList`. А ведь разница огромная: десятая доля секунды против целых двух секунд, то есть в 20 раз быстрее! Кроме того, использование значимого типа (`Int32`) с алгоритмом `ArrayList` требует множества операций упаковки, и, как результат, 30 процедур сборки мусора, а в алгоритме `List` сборка мусора вообще не нужна.

Результаты проверки ссылочного типа не столь впечатляющие: временные показатели и число операций сборки мусора здесь примерно одинаковы. Поэтому в данном случае у обобщенного алгоритма `List` реальных преимуществ нет. Тем не менее помните, что применение обобщенного алгоритма значительно упрощает код и контроль типов при компиляции. Таким образом, хотя выигрыша в производительности практически нет, обобщенный алгоритм берет свое в другом.

ПРИМЕЧАНИЕ

Необходимо понимать, что CLR генерирует машинный код для любого метода при первом его вызове в применении к конкретному типу данных. Это увеличивает размер рабочего набора приложения и снижает производительность. Подробнее об этом мы поговорим чуть позже в разделе «Инфраструктура обобщений».

Обобщения в библиотеке FCL

Разумеется, обобщения применяются с классами коллекций, и в FCL определено несколько таких обобщенных классов. Большинство этих классов можно найти в пространствах имен `System.Collections.Generic` и `System.Collections.ObjectModel`. Есть также безопасные в отношении потоков классы коллекций в пространстве имен `System.Collections.Concurrent`. Microsoft рекомендует программистам отказаться от необобщенных классов коллекций в пользу их обобщенных аналогов по нескольким причинам. Во-первых, необобщенные классы коллекций, в отличие от обобщенных, не обеспечивают безопасность типов, простоту и понятность кода и повышение производительности. Во-вторых, объектная модель у обобщенных классов лучше, чем у необобщенных. Например, у них меньше виртуальных методов, что повышает производительность, а новые члены, добавленные в обобщенные коллекции, предоставляют массу новых возможностей.

Классы коллекций реализуют множество интерфейсов, а объекты, добавляемые в коллекции, могут реализовывать интерфейсы, используемые классами коллекций для таких операций, как сортировка и поиск. В составе FCL поставляется множество определений обобщенных интерфейсов, поэтому при работе с интерфейсами также доступны преимущества обобщений. Большинство используемых интерфейсов содержится в пространстве имен `System.Collections.Generic`.

Новые обобщенные интерфейсы не заменяют необобщенные: во многих ситуациях приходится задействовать оба вида интерфейсов. Причина — необходимость поддержания обратной совместимости. Например, если бы класс `List<T>` реализовывал только интерфейс `ICollection<T>`, в коде нельзя было бы рассматривать объект `List<DateTime>` как `ICollection`.

Также отмечу, что класс `System.Array`, базовый для всех типов массивов, поддерживает множество статических обобщенных методов, в том числе `AsReadOnly`, `BinarySearch`, `ConvertAll`, `Exists`, `Find`, `FindAll`, `FindIndex`, `FindLast`, `FindLastIndex`, `ForEach`, `IndexOf`, `LastIndexOf`, `Resize`, `Sort` и `TrueForAll`. Вот как выглядят некоторые из них:

```
public abstract class Array : ICollection, IList, IEnumerable {
    public static void Sort<T>(T[] array);

    public static void Sort<T>(T[] array, IComparer<T> comparer);
    public static int BinarySearch<T>(T[] array, T value);

    public static int BinarySearch<T>(T[] array, T value,
        IComparer<T> comparer);
    ...
}
```

Следующий код иллюстрирует применение нескольких из этих методов:

```
public static void Main() {
    // Создание и инициализация массива байтов
    byte[] byteArray = new byte[] { 5, 1, 4, 2, 3 };

    // Вызов алгоритма сортировки byte[]
    Array.Sort<byte>(byteArray);

    // Вызов алгоритма двоичного поиска byte[]
    int i = Array.BinarySearch<byte>(byteArray, 1);
    Console.WriteLine(i); // Выводит "0"
}
```

Библиотека Power Collections производства Wintellect

По заказу Microsoft компания Wintellect разработала библиотеку Power Collections, основное назначение которой — сделать некоторые классы коллекций из STL-библиотеки C++ доступными для CLR. Классы библиотеки Power Collections распространяются бесплатно (детали см. на <http://Wintellect.com>). Эти классы коллекций сами по себе являются обобщенными и в них широко

используются обобщения. В табл. 12.1 приведен неполный список классов коллекций из библиотеки Power Collections.

Таблица 12.1. Обобщенные классы коллекций из библиотеки Power Collections от Wintellect

Класс набора	Описание
BigList<T>	Коллекция упорядоченных объектов T. Очень эффективна, когда объектов больше 100
Bag<T>	Коллекция неупорядоченных объектов T. Эта коллекция хэшируется и допускает дублирование объектов
OrderedBag<T>	Коллекция упорядоченных объектов T. Допускается дублирование объектов
Set<T>	Коллекция неупорядоченных элементов T. Дублирование не поддерживается
OrderedSet<T>	Коллекция упорядоченных элементов T. Дублирование не поддерживается
Deque<T>	Очередь с двусторонним доступом. Похожа на список, но более эффективна при добавлении/удалении элементов в начало
OrderedDictionary<TKey,TValue>	Словарь с упорядоченными однозначными ключами <TKey,TValue>
MultiDictionary<TKey,TValue>	Словарь с многозначными ключами. Ключи хэшируются, допускается дублирование, элементы не упорядочены
OrderedMultiDictionary<TKey,TValue>	Словарь с упорядоченными многозначными ключами (также в отсортированном порядке). Допускается дублирование ключей

Инфраструктура обобщений

Обобщения были добавлены в версию 2.0 CLR, над реализацией обобщений долго трудилось множество специалистов. Для поддержания работы обобщений Microsoft нужно было сделать следующее.

- ❑ Создать новые IL-команды, работающие с аргументами типа.
- ❑ Изменить формат существующих таблиц метаданных для выражения имен типов и методов с обобщенными параметрами.
- ❑ Обновить многие языки программирования (в том числе C#, Microsoft Visual Basic .NET и др.), чтобы обеспечить поддержку нового синтаксиса и позволить разработчикам определять и ссылаться на новые обобщенные типы и методы.

- ❑ Изменить компиляторы для генерации новых IL-команд и измененного формата метаданных.
- ❑ Изменить JIT-компилятор, чтобы он обрабатывал новые IL-команды, работающие с аргументами типа, и создавал корректный машинный код.
- ❑ Создать новых членов отражения, для того чтобы разработчики с помощью запроса могли проверять наличие обобщенных параметров у типов и членов.
- ❑ Определить новые предоставляющие информацию отражения члены, что позволило разработчикам создавать обобщенные определения типов и методов во время исполнения.
- ❑ Изменить отладчик, чтобы он поддерживал обобщенные типы, члены, поля и локальные переменные.
- ❑ Изменить механизм IntelliSense в Microsoft Visual Studio для вывода конкретных прототипов членов при использовании обобщенного типа или метода с указанием типа данных.

А теперь обсудим, как работают с обобщениями внутренние механизмы CLR. Эта информация пригодится вам как при проектировании и создании обобщенных алгоритмов, так и при выборе готовых обобщенных алгоритмов.

Открытые и закрытые типы

Я уже рассказывал, как CLR создает внутреннюю структуру данных для каждого типа, применяемого в приложении. Эти структуры данных называют *объектами типа* (type objects). Тип с обобщенными параметрами типа также считается типом, причем для такого типа CLR создает внутренний объект типа. Это справедливо для ссылочных типов (классов), значимых типов (структур), интерфейсов и делегатов. Тем не менее тип с обобщенными параметрами типа называют *открытым типом* (open type), а в CLR запрещено конструирование экземпляров открытых типов (как и экземпляров интерфейсных типов).

При ссылке на обобщенный тип в коде можно определить набор обобщенных аргументов типа. Если всем аргументам определенного типа переданы действительные типы данных, то он становится *закрытым типом* (closed type). CLR разрешает создание экземпляров закрытых типов. Тем не менее в коде, ссылающемся на обобщенный тип, можно не определять все обобщенные аргументы типа. Таким образом, в CLR создается новый объект открытого типа, экземпляры которого создавать нельзя. Следующий код проясняет ситуацию:

```
using System;  
using System.Collections.Generic;
```

```
// Частично определенный открытый тип  
internal sealed class DictionaryStringKey<TValue> :  
    Dictionary<String, TValue> {  
}
```

продолжение ➤

```

public static class Program {
    public static void Main() {
        Object o = null;

        // Dictionary<,> – это открытый тип с двумя параметрами типа
        Type t = typeof(Dictionary<,>);

        // Попытка создания экземпляра этого типа (неудачная)
        o = CreateInstance(t);
        Console.WriteLine();

        // DictionaryStringKey<> – это открытый тип с одним параметром типа
        t = typeof(DictionaryStringKey<>);

        // Попытка создания экземпляра этого типа (неудачная)
        o = CreateInstance(t);
        Console.WriteLine();

        // DictionaryStringKey<Guid> – это закрытый тип
        t = typeof(DictionaryStringKey<Guid>);

        // Попытка создания экземпляра этого типа (удачная)
        o = CreateInstance(t);

        // Проверка успешности попытки
        Console.WriteLine("Object type=" + o.GetType());
    }

    private static Object CreateInstance(Type t) {
        Object o = null;
        try {
            o = Activator.CreateInstance(t);
            Console.WriteLine("Created instance of {0}", t.ToString());
        }
        catch (ArgumentException e) {
            Console.WriteLine(e.Message);
        }
        return o;
    }
}

```

Скомпилировав и выполнив этот код, вы увидите следующее:

```

Cannot create an instance of System.Collections.Generic.
Dictionary`2[TKey,TValue] because Type.ContainsGenericParameters is true.

```

```

Cannot create an instance of DictionaryStringKey`1[TValue] because
Type.ContainsGenericParameters is true.

```

```

Created instance of DictionaryStringKey`1[System.Guid]
Object type=DictionaryStringKey`1[System.Guid]

```

Итак, при попытке создания экземпляра открытого типа метод `CreateInstance` объекта `Activator` вбрасывает исключение `ArgumentException`. На самом деле, сообщение об исключении означает, что тип все же содержит несколько обобщенных параметров.

В выводимой программой информации видно, что имена типов заканчиваются левой одиночной кавычкой (```), за которой следует число, означающее *арность* (arity) типа, то есть число необходимых для него параметров типа. Например, арность класса `Dictionary` равна 2, потому что для него требуется определить типы `TKey` и `TValue`. Арность класса `DictionaryStringKey` равна 1, так как требуется указать лишь один тип — `TValue`. Необходимо отметить, что CLR размещает статические поля типа в самом объекте типа (см. главу 4). Поэтому у каждого закрытого типа есть свои статические поля. Иначе говоря, статические поля, определенные в объекте `List<T>`, не будут совместно использоваться объектами `List<DateTime>` и `List<String>`, потому что у каждого объекта закрытого типа есть свои статические поля. Если же в обобщенном типе определен статический конструктор (см. главу 8), то последний выполняется для закрытого типа лишь раз. Иногда разработчики определяют статический конструктор для обобщенного типа, чтобы аргументы типа соответствовали определенным критериям. Например, обобщенный тип, используемый только с перечислимыми типами, определяется следующим образом:

```
internal sealed class GenericTypeThatRequiresAnEnum<T> {
    static GenericTypeThatRequiresAnEnum() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enumerated type");
        }
    }
}
```

В CLR есть механизм *ограничений* (constraints), предлагающий более удачный инструмент определения обобщенного типа с указанием допустимых для него аргументов типа. Но об ограничениях — чуть позже. К сожалению, этот механизм не позволяет ограничить аргументы типа только перечислимыми типами. Поэтому в предыдущем примере необходим статический конструктор для проверки того, что используемый тип является перечислимым.

Обобщенные типы и наследование

Обобщенный тип, как и всякий другой, может быть производным от других типов. При использовании обобщенного типа с указанием аргументов типа в CLR определяется новый объект типа, производный от того же типа, что и обобщенный тип.

Например, тип `List<T>` является производным от `Object`, поэтому типы `List<String>` и `List<Guid>` тоже производные от `Object`. Аналогично, тип `DictionaryStringKey<TValue>` — производный от `Dictionary<String, TValue>`, поэтому

тип `DictionaryStringKey<Guid>` также производный от `Dictionary<String, Guid>`. Понимание того, что определение аргументов типа не имеет ничего общего с иерархиями наследования, позволяет разобраться, какие приведения типов допустимы, а какие нет.

Например, пусть класс `Node` связанного списка определяется следующим образом.

```
internal sealed class Node<T> {
    public T m_data;
    public Node<T> m_next;

    public Node(T data) : this(data, null) {
    }

    public Node(T data, Node<T> next) {
        m_data = data; m_next = next;
    }

    public override String ToString() {
        return m_data.ToString() +
            ((m_next != null) ? m_next.ToString() : null);
    }
}
```

Тогда код для создания связанного списка будет примерно таким:

```
private static void SameDataLinkedList() {
    Node<Char> head = new Node<Char>('C');
    head = new Node<Char>('B', head);
    head = new Node<Char>('A', head);
    Console.WriteLine(head.ToString());
}
```

В приведенном классе `Node` поле `m_next` должно ссылаться на другой узел, поле `m_data` которого содержит тот же тип данных. Это значит, что узлы связанного списка должны иметь одинаковый (или производный) тип данных. Например, нельзя использовать класс `Node` для создания связанного списка, в котором тип данных одного элемента — `Char`, другого — `DateTime`, а третьего — `String`.

Однако определив необобщенный базовый класс `Node`, а затем — обобщенный класс `TypedNode` (используя класс `Node` как базовый), можно создать связанный список с произвольным типом данных у каждого узла. Вот определения новых классов:

```
internal class Node {
    protected Node m_next;
    public Node(Node next) {
        m_next = next;
    }
}
```

```
internal sealed class TypedNode<T> : Node {
    public T m_data;
    public TypedNode(T data) : this(data, null) {
    }
    public TypedNode(T data, Node next) : base(next) {
        m_data = data;
    }

    public override String ToString() {
        return m_data.ToString() +
            ((m_next != null) ? m_next.ToString() : null);
    }
}
```

Теперь можно написать код для создания связанного списка с разными типами данных у разных узлов. Код будет примерно таким:

```
private static void DifferentDataLinkedList() {
    Node head = new TypedNode<Char>('.');
    head = new TypedNode<DateTime>(DateTime.Now, head);
    head = new TypedNode<String>("Today is ", head);
    Console.WriteLine(head.ToString());
}
```

Идентификация обобщенных типов

Синтаксис обобщенных типов часто приводит разработчиков в замешательство. В исходном коде часто оказывается слишком много знаков «меньше» (<) и «больше» (>), и это сильно затрудняет его чтение. Для упрощения синтаксиса некоторые разработчики определяют новый необобщенный тип класса, производный от обобщенного типа и определяющий все необходимые аргументы типа. Например, пусть нужно упростить следующий код:

```
List<DateTime> dt = new List<DateTime>();
```

Некоторые разработчики сначала определяют класс:

```
internal sealed class DateTimeList : List<DateTime> {
    // Здесь никакой код добавлять не нужно!
}
```

Теперь код, создающий список, можно написать проще (без знаков «меньше» и «больше»):

```
DateTimeList dt = new DateTimeList();
```

Этот вариант особенно удобен при использовании нового типа для параметров, локальных переменных и полей. И все же ни в коем случае нельзя явно определять новый класс лишь затем, чтобы сделать исходный текст более читабельным.

Причина проста: пропадает тождественность и эквивалентность типов, как видно из следующего кода:

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

При выполнении этого кода `sameType` инициализируется значением `false`, потому что сравниваются два объекта разных типов. Это также значит, что методу, в прототипе которого определено, что он принимает значение типа `DateTimeList`, нельзя передать `List<DateTime>`. Тем не менее методу, который должен принимать `List<DateTime>`, можно передать `DateTimeList`, потому что тип `DateTimeList` является производным от `List<DateTime>`. Запутаться в этом очень просто.

К счастью, C# позволяет использовать упрощенный синтаксис для ссылки на обобщенный закрытый тип, не влияющий на эквивалентность типов. Для этого в начало файла с исходным текстом нужно добавить старую добрую директиву `using`:

```
using DateTimeList = System.Collections.Generic.List<System.DateTime>;
```

Здесь директива `using` просто определяет символ `DateTimeList`. При компиляции кода компилятор заменяет все вхождения `DateTimeList` типом `System.Collections.Generic.List<System.DateTime>`. Таким образом, разработчики могут использовать упрощенный синтаксис, не меняя смысл кода и тем самым сохраняя идентификацию и тождество типов. Теперь при выполнении следующей строки кода `sameType` инициализируется значением `true`:

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

Для удобства вы можете использовать свойство локальной переменной неявного типа языка C#, для которой компилятор обозначает тип локальной переменной метода из типа вашего выражения:

```
using System;
using System.Collections.Generic;
...
internal sealed class SomeType {
    private static void SomeMethod () {

        // Компилятор полагает, что DateTimeList имеет тип
        // System.Collections.Generic.List<System.DateTime>
        var dtl = List<DateTime>();

        ...
    }
}
```

«Распухание» кода

При JIT-компиляции метода, в котором используются обобщенные параметры типа, CLR подставляет в IL-код метода указанные аргументы типа, а затем соз-

дает машинный код для данного метода, работающего с конкретными типами данных. Это именно то, что нужно, и это одна из основных функций обобщения. Но в таком подходе есть один недостаток: CLR генерирует машинный код для каждого сочетания «метод + тип», что приводит к *распуханию кода* (code explosion), в итоге существенно увеличивается рабочий набор приложения, снижая производительность.

К счастью, в CLR есть несколько оптимизационных алгоритмов, призванных предотвратить разрастание кода. Во-первых, если метод вызывается для конкретного аргумента типа и позже он вызывается опять с тем же аргументом типа, CLR компилирует код для такого сочетания «метод + тип» только один раз. Поэтому, если `List<DateTime>` используется в двух совершенно разных сборках (загруженных в один домен приложений), CLR компилирует методы для `List<DateTime>` всего один раз. Это существенно сокращает степень распухания кода.

При использовании другого алгоритма оптимизации CLR считает все аргументы ссылочного типа тождественными, что опять же обеспечивает совместное использование кода. Например, код, скомпилированный в CLR для методов `List<String>`, может применяться для методов `List<Stream>`, потому что `String` и `Stream` — ссылочные типы. По сути, для всех ссылочных типов используется одинаковый код. CLR выполняет эту оптимизацию, потому что все аргументы и переменные ссылочного типа — это просто указатели на объекты в куче (32-разрядное значение в 32-разрядной и 64-разрядное значение в 64-разрядной версии Windows), а все указатели на объекты обрабатываются одинаково.

Но если аргументы типа имеют значимый тип, среда CLR должна сгенерировать машинный код именно для этого значимого типа. Это объясняется тем, что у значимых типов может быть разный размер. И даже если у двух значимых типов одинаковый размер (например, `Int32` и `UInt32` — это 32-разрядные значения), CLR все равно не может использовать для них единый код, потому что для обработки этих значений могут применяться разные машинные команды.

Обобщенные интерфейсы

Конечно же, основное преимущество обобщений — их способность определять обобщенные ссылочные и значимые типы. Но для CLR также исключительно важна поддержка обобщенных интерфейсов. Без них любая попытка работы со значимым типом через необобщенный интерфейс (например, `IComparable`) всякий раз будет приводить к необходимости упаковки и потере безопасности типов в процессе компиляции. Это сильно сузило бы область применения обобщенных типов. Вот почему CLR поддерживает обобщенные интерфейсы. Ссылочный и значимый типы реализуют обобщенный интерфейс путем зада-

ния аргументов типа, или же любой тип реализует обобщенный интерфейс, не определяя аргументы типа. Рассмотрим несколько примеров.

Определение обобщенного интерфейса из библиотеки FCL (из пространства имен `System.Collections.Generic`) выглядит следующим образом:

```
public interface IEnumerator<T> : IDisposable, IEnumerator {
    T Current { get; }
}
```

А этот тип реализует данный обобщенный интерфейс и задает аргументы типа.

Обратите внимание, что объект `Triangle` может перечислять набор объектов `Point`, а типом свойства `Current` является `Point`:

```
internal sealed class Triangle : IEnumerator<Point> {
    private Point[] m_vertices;

    // Тип свойства Current в IEnumerator<Point> - это Point
    Point Current { get { ... } }
    ...
}
```

Теперь рассмотрим пример типа, реализующего тот же обобщенный интерфейс, но без задания аргументов типа:

```
internal sealed class ArrayEnumerator<T> : IEnumerator<T> {
    private T[] m_array;

    // Тип свойства Current в IEnumerator<T> - T
    T Current { get { ... } }
    ...
}
```

Обратите внимание, что объект `ArrayEnumerator` перечисляет набор объектов `T` (где `T` не задано, поэтому код, использующий обобщенный тип `ArrayEnumerator`, может задать тип `T` позже). Также отмечу, что в этом примере свойство `Current` имеет неопределенный тип данных `T`. Подробнее обобщенные интерфейсы обсуждаются в главе 13.

Обобщенные делегаты

Поддержка обобщенных делегатов в CLR позволяет передавать методам обратного вызова любые типы объектов, обеспечивая при этом безопасность типов. Более того, благодаря обобщенным делегатам экземпляры значимого типа могут передаваться методам обратного вызова без упаковки. Как уже отмечалось в главе 17, делегат — это просто определение класса с помощью четырех мето-

дов: конструктора и методов `Invoke`, `BeginInvoke` и `EndInvoke`. При определении типа делегата с параметрами типа компилятор задает методы класса делегата, а параметры типа применяются ко всем методам, параметры и возвращаемые значения которых относятся к указанному параметру типа.

Например, обобщенный делегат определяется следующим образом:

```
public delegate TReturn CallMe<TReturn, TKey, TValue>(
    TKey key, TValue value);
```

Компилятор превращает его в класс, который логически выглядит так:

```
public sealed class CallMe<TReturn, TKey, TValue> : MulticastDelegate {
    public CallMe(Object object, IntPtr method);
    public TReturn Invoke(TKey key, TValue value);
    public IAsyncResult BeginInvoke(TKey key, TValue value,
        AsyncCallback callback, Object object);
    public TReturn EndInvoke(IAsyncResult result);
}
```

ПРИМЕЧАНИЕ

Рекомендовано использовать обобщения `Action` и `Func`, которые поставляются предопределенными в библиотеке `FCL`. Я описал эти типы делегатов в главе 17.

Контравариантные и ковариантные аргументы обобщенного типа делегатов и интерфейсов

Каждый из параметров обобщенного типа делегата должен быть помечен как ковариантный или контравариантный. Это функциональная возможность позволяет вам осуществлять приведение типа переменной обобщенного типа делегата к тому же типу, что и делегат, причем параметры обобщенных типов будут отличаться. Параметры обобщенного типа могут быть следующими.

- ❑ **Инвариантными.** Параметр обобщенного типа не может изменяться. Далее в этой главе показаны только инвариантные параметры обобщенного типа.
- ❑ **Контравариантными.** Параметр обобщенного типа может изменяться в наследуемом классе. В языке `C#` вы обозначаете контравариантный тип при помощи ключевого слова `in`. Контравариантный параметр обобщенного типа может появляться только во входной позиции, например, в качестве аргументов метода.
- ❑ **Ковариантными.** Аргумент обобщенного типа может быть изменен в базовом классе. В языке `C#` вы обозначаете ковариантный тип при помощи ключевого слова `out`. Ковариантный параметр обобщенного типа может появляться только в выходной позиции, например, в качестве возвращаемого значения метода.

Предположим, что существует следующий тип делегата:

```
public delegate TResult Func<in T, out TResult>(T arg);
```

Здесь параметр обобщенного типа `T` помечен словом `in`, делающим его контравариантным, а параметр обобщенного типа `TResult` помечен словом `out`, делающим его ковариантным.

Пусть объявлена следующая переменная:

```
Func<Object, ArgumentException> fn1 = null;
```

В этом случае я могу привести ее к типу `Func` с отличающимися параметрами обобщенного типа:

```
Func<String, Exception>fn2 = fn1; // Явного приведения типа не требуется
Exception e = fn2("");
```

Это говорит о том, что `fn1` ссылается на функцию, доступную в `Object`, и возвращает `ArgumentException`.

Переменная `fn2` пытается сослаться на метод, в который передается переменная строкового типа и который возвращает `Exception`. Если вы передадите строковую переменную в метод, который требует тип `Object` (`String` наследуется от `Object`), и если вы вернете в качестве возвращаемого значения `ArgumentException` и пометите его как `Exception` (`ArgumentException` наследуется от `Exception`), представленный здесь программный код скомпилируется и на этапе компиляции будет сохраняться безопасность типов.

ПРИМЕЧАНИЕ

Расхождение имеет место, только если компилятор сможет установить, что между двумя типами существует обращение по ссылке. Другими словами, расхождение невозможно для значимых типов из-за того, что требуется упаковка (`boxing`). Я считаю, что это ограничение, которое делает расхождение бесполезным. Например:

```
void ProcessCollection(IEnumerable<Object> collection) { ... }
```

Я не смогу вызвать этот метод путем передачи ссылки объекту `List<DateTime>`, если не существует ссылочного перехода между значимым типом `DateTime` и объектом `Object`, даже если `DateTime` унаследован от объекта `Object`. Можно решить эту проблему следующим образом:

```
void ProcessCollection<T>(IEnumerable<T> collection) { ... }
```

Плюс большая выгода от `ProcessCollection(IEnumerable<Object> collection)` в том, что здесь используется только одна версия JIT-кода. Однако есть только одна версия JIT-кода для `ProcessCollection<T> (IEnumerable<T> collection)`, поддерживаемая всеми `T`, являющимися ссылочными типами, но вы не сможете вызвать метод путем передачи ему значимого типа.

Расхождение также недопустимо для параметра обобщенного типа, если аргумент того типа передается в метод при помощи ключевых слов `out` и `ref`. Например:

```
delegate void SomeDelegate<in T>(ref T t);
```

Эта строка кода заставляет компилятор выдать следующее сообщение об ошибке (недействительное расхождение: Тип параметра 'T' должен быть инвариантно действительным для 'SomeDelegate<T>.Invoke(ref T)'. 'T' — контрвариантно):

```
Invalid variance: The type parameter 'T' must be invariantly valid on  
'SomeDelegate<T>.Invoke(ref T)'. 'T' is contravariant
```

При использовании делегатов, получающих обобщенные аргументы и возвращающих значения, рекомендуется всегда использоваться ключевые слова `in` и `out` для обозначения контрвариантности и ковариантности везде, где это возможно. Это не дает никакого негативного эффекта, но делает ваших делегатов применимыми в большем количестве сценариев.

Подобно делегатам, интерфейсы с параметрами обобщенного типа могут иметь параметры типа либо контрвариантными, либо ковариантными. Приведу пример интерфейса с контрвариантным параметром обобщенного типа:

```
public interface IEnumerator<out T> : IEnumerator {  
    Boolean MoveNext();  
    T Current { get; }  
}
```

По причине того, что `T` является контрвариантным, можно успешно скомпилировать и выполнить следующий программный код:

```
// Этот метод допускает интерфейс IEnumerable любого ссылочного типа  
Int32 Count(IEnumerable<Object> collection) { ... }  
  
...  
// Этот вызов передает IEnumerable<String> в Count  
Int32 c = Count(new[] { "Grant" });
```

ВНИМАНИЕ

Иногда разработчики спрашивают, почему они должны явно указывать слово `in` или `out` в параметрах обобщенного типа. Они думают, что компилятор может самостоятельно проверить объявление делегатов или интерфейсов и автоматически определить, являются ли параметры обобщенного типа контрвариантными или ковариантными. Несмотря на то что компилятор действительно может это определять автоматически, разработчики языка `C#` надеются, что вы будете указывать эти слова в явном виде. Например, было бы плохо, если компилятор определил, что параметр обобщенного типа контрвариантен и затем, в будущем, вы бы добавили в выходной позиции интерфейса член, имеющий параметр типа. В следующий раз при

компиляции компилятор определит, что этот параметр типа инвариантен, но весь программный код располагает ссылками на другие члены, а это может стать причиной ошибок, если они предполагают, что данный параметр типа контравариантный.

В силу этой причины разработчики компилятора рекомендуют указывать параметр обобщенного типа в явном виде. И потом, когда вы попытаетесь использовать этот параметр типа в контексте, вами необъявленном, компилятор выдаст ошибку, позволяющую узнать, в чем дело, и исправить код. Если потом вы решите исправить код путем добавления параметров обобщенного типа `in` или `out`, вам придется внести изменения в программный код.

Обобщенные методы

При определении обобщенного ссылочного и значимого типа или интерфейса все методы, определенные в этих типах, могут ссылаться на любой параметр типа, заданный этим типом. Параметр типа может использоваться как параметр метода, возвращаемое значение метода или как заданная внутри него локальная переменная. Но CLR также позволяет методу иметь собственные параметры типа, которые могут применяться в качестве параметров, возвращаемых значений или локальных переменных. Вот немного искусственный пример типа, в котором определяются параметр типа и метод с собственным параметром типа:

```
internal sealed class GenericType<T> {
    private T m_value;

    public GenericType(T value) { m_value = value; }

    public TOutput Converter<TOutput>() {
        TOutput result = (TOutput) Convert.ChangeType(m_value, typeof(TOutput));
        return result;
    }
}
```

Здесь в классе `GenericType` определяется свой параметр типа (`T`), а в методе `Converter` — свой (`TOutput`). Благодаря этому можно создать класс `GenericType`, работающий с любым типом. Метод `Converter` преобразует объект, на который ссылается поле `m_value`, в другие типы в зависимости от аргумента типа, переданного ему при его вызове. Возможность иметь параметры типа и параметры метода дает небывалую гибкость.

Удачный пример обобщенного метода — метод `Swap`:

```
private static void Swap<T>(ref T o1, ref T o2) {
    T temp = o1;
    o1 = o2;
    o2 = temp;
}
```

Теперь вызывать Swap из кода можно следующим образом:

```
private static void CallingSwap() {  
    Int32 n1 = 1, n2 = 2;  
    Console.WriteLine("n1={0}, n2={1}", n1, n2);  
    Swap<Int32>(ref n1, ref n2);  
    Console.WriteLine("n1={0}, n2={1}", n1, n2);  
  
    String s1 = "Aidan", s2 = "Kristin";  
    Console.WriteLine("s1={0}, s2={1}", s1, s2);  
    Swap<String>(ref s1, ref s2);  
    Console.WriteLine("s1={0}, s2={1}", s1, s2);  
}
```

Использование обобщенных типов с методами, принимающими параметры `out` и `ref`, особенно интересно тем, что переменные, передаваемые в качестве аргумента `out/ref`, должны быть того же типа, что и параметр метода, чтобы избежать возможного нарушения безопасности типов. Эта особенность параметров `out/ref` обсуждается в главе 9. В сущности, именно поэтому методы `Exchange` и `CompareExchange` класса `Interlocked` поддерживают обобщенную перегрузку¹:

```
public static class Interlocked {  
    public static T Exchange<T>(ref T location1, T value) where T: class;  
    public static T CompareExchange<T>(  
        ref T location1, T value, T comparand) where T: class;  
}
```

Интерфейсы обобщенных методов и типов

Синтаксис обобщений в C# со всеми его знаками «меньше» и «больше» приводит в замешательство многих разработчиков. С целью упростить создание, чтение и работу с кодом компилятор C# предлагает *логический вывод типов* (type inference) при вызове обобщенных методов. Это значит, что компилятор пытается определить (или логически вывести) тип, который будет автоматически использоваться при вызове обобщенного метода. Логический вывод типов иллюстрирует следующий код:

```
private static void CallingSwapUsingInference() {  
    Int32 n1 = 1, n2 = 2;  
    Swap(ref n1, ref n2); // Вызывает Swap<Int32>  
  
    String s1 = "Aidan";  
    Object s2 = "Kristin";  
    Swap(ref s1, ref s2); // Ошибка, невозможно вывести тип  
}
```

¹ Ключевое слово `where` описывается в разделе «Верификация и ограничения» этой главы.

Обратите внимание, что в этом коде при вызове `Swap` аргументы типа не задаются с помощью знаков «меньше» и «больше». В первом вызове `Swap` компилятор C# сумел установить, что типом переменных `n1` и `n2` является `Int32`, поэтому вызвал `Swap`, используя аргумент типа `Int32`.

При выполнении логического вывода типа в C# используется тип данных переменной, а не объекта, на который ссылается эта переменная. Поэтому во втором вызове `Swap` компилятор C# «видит», что `s1` имеет тип `String`, а `s2` — `Object` (хотя `s2` ссылается на `String`). Поскольку у переменных `s1` и `s2` разный тип данных, компилятор не может с точностью вывести тип для аргумента типа метода `Swap` и выдает ошибку (ошибка CS0411: аргументы типа для метода `Program.Swap<T>(ref T, ref T)` не могут быть выведены. Попробуйте явно задать аргументы типа):

```
error CS0411: The type arguments for method 'Program.Swap<T>(ref T, ref T)'
cannot be inferred from the usage. Try specifying the type arguments explicitly
```

В типе могут определяться несколько методов таким образом, что один из них будет принимать конкретный тип данных, а другой — обобщенный параметр типа, как в этом примере:

```
private static void Display(String s) {
    Console.WriteLine(s);
}

private static void Display<T>(T o) {
    Display(o.ToString()); // Вызывает Display(String)
}
```

Метод `Display` можно вызвать несколькими способами:

```
Display("Jeff");           // Вызывает Display(String)
Display(123);              // Вызывает Display<T>(T)
Display<String>("Aidan");  // Вызывает Display<T>(T)
```

В первом случае компилятор может вызвать либо метод `Display`, принимающий `String`, либо обобщенный метод `Display` (заменяя `T` типом `String`). Но компилятор C# всегда выбирает явное, а не обобщенное соответствие, поэтому генерирует вызов необобщенного метода `Display`, принимающего `String`. Во втором случае компилятор не может вызвать необобщенный метод `Display`, принимающий `String`, поэтому он вызывает обобщенный метод `Display`. Кстати, очень удачно, что компилятор всегда выбирает более явное соответствие. Ведь если бы компилятор выбрал обобщенный метод `Display`, тот вызвал бы метод `ToString`, возвращающий `String`, что привело бы к бесконечной рекурсии.

В третьем вызове метода `Display` задается обобщенный аргумент типа `String`. Для компилятора это означает, что не нужно пытаться логически вывести аргументы типа, а нужно использовать указанные аргументы типа. В данном случае компилятор также считает, что непременно нужно вызвать обобщенный метод `Display`, поэтому он его и вызывает. Внутренний код обобщенного метода

Display вызывает ToString для переданной ему строки, а полученная в результате строка затем передается необобщенному методу Display.

Обобщения и другие члены

В языке C# у свойств, индексаторов, событий, операторных методов, конструкторов и деструкторов не может быть параметров типа. Однако их можно определить в обобщенном типе с тем, чтобы в их коде использовать параметры этого типа.

C# не поддерживает задание собственных обобщенных параметров типа у этих членов, поскольку создатели языка C# из компании Microsoft считают, что разработчикам вряд ли потребуется задействовать эти члены в качестве обобщенных. К тому же, чтобы они могли применяться как обобщенные, для C# пришлось бы разработать специальный синтаксис, что довольно затратно. Например, при использовании в коде оператора + компилятор может вызвать перегруженный операторный метод. И нет способа, позволяющего указать в коде, где есть оператор +, какие бы то ни было аргументы типа.

Верификация и ограничения

В процессе компиляции обобщенного кода компилятор C# анализирует его, убеждаясь, что он сможет работать с любыми типами данных — существующими и теми, которые будут определены в будущем. Рассмотрим следующий метод:

```
private static Boolean MethodTakingAnyType<T>(T o) {  
    T temp = o;  
    Console.WriteLine(o.ToString());  
    Boolean b = temp.Equals(o);  
    return b;  
}
```

Здесь объявляется временная переменная (temp) типа T, а затем выполняется несколько операций присваивания переменных и несколько вызовов методов. Представленный метод работает с любым типом T — ссылочным, значимым, перечислимым, типом интерфейса или типом делегата, существующим типом или типом, который определят в будущем, потому что любой тип поддерживает присваивание и вызовы методов, определенных в Object (например, ToString и Equals).

Вот еще метод:

```
private static T Min<T>(T o1, T o2) {  
    if (o1.CompareTo(o2) < 0) return o1;  
    return o2;  
}
```

Метод `Min` пытается через переменную `o1` вызвать метод `CompareTo`. Но многие типы не поддерживают метод `CompareTo`, поэтому компилятор C# не в состоянии скомпилировать этот код и обеспечить, чтобы после компиляции метод смог работать со всеми типами. При попытке скомпилировать приведенный код появится сообщение об ошибке (ошибка CS0117: `T` не содержит определение метода `CompareTo`):

```
error CS0117: 'T' does not contain a definition for 'CompareTo'
```

Поэтому может показаться, что при использовании обобщений можно лишь объявлять переменные обобщенного типа, назначать переменные, вызывать методы, определенные в `Object`, — и все! Но ведь в таком случае от обобщений пользы мало.

К счастью, компиляторы и CLR поддерживают уже упоминавшийся механизм *ограничений* (constraints), благодаря которому обобщения успешно «реабилитируются».

Ограничение позволяет сузить перечень типов, которые можно передать в обобщенном аргументе, и расширяет возможности по работе с этими типами. Вот новый вариант метода `Min`, который задает ограничение (выделено полужирным шрифтом):

```
public static T Min<T>(T o1, T o2) where T : IComparable<T> {
    if (o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```

Маркер `where` в C# сообщает компилятору, что указанный в `T` тип должен реализовывать обобщенный интерфейс `IComparable` того же типа (`T`). Благодаря этому ограничению компилятор разрешает методу вызвать метод `CompareTo`, потому что последний определен в интерфейсе `IComparable<T>`.

Теперь, когда код ссылается на обобщенный тип или метод, компилятор должен убедиться, что в коде указан аргумент типа, удовлетворяющий этим ограничениям. Пример:

```
private static void CallMin() {
    Object o1 = "Jeff", o2 = "Richter";
    Object oMin = Min<Object>(o1, o2); // Ошибка CS0309
}
```

При компиляции этого кода появляется сообщение (ошибка CS0309: тип `object` необходимо преобразовать в `System.IComparable<object>`, чтобы его можно было использовать в качестве параметра `T` в обобщенном типе или методе `Program.Min<T>(T, T)`):

```
error CS0309: The type 'object' must be convertible to 'System.
IComparable<object>' in order to use it as a parameter 'T' in the generic type
or method 'Program.Min<T>(T, T)'
```

Компилятор выдает эту ошибку, потому что `System.Object` не реализует интерфейс `Comparable<Object>`. На самом деле, `System.Object` вообще не реализует никаких интерфейсов.

На данном этапе вы получили общее представление об ограничениях и их работе. Давайте познакомимся с ними поближе. Ограничения можно применять к параметрам типа как обобщенных типов, так и обобщенных методов (как показано в методе `Min`). Среда CLR не поддерживает перегрузку по именам параметров типа или по именам ограничений. Перегрузка типов и методов выполняется только по арности. Покажу это на примере:

```
// Можно определить следующие типы:
internal sealed class AType {}
internal sealed class AType<T> {}
internal sealed class AType<T1, T2> {}

// Ошибка: конфликт с типом AType<T>, у которого нет ограничений.
internal sealed class AType<T> where T : Comparable<T> {}

// Ошибка: конфликт с типом AType<T1, T2>
internal sealed class AType<T3, T4> {}

internal sealed class AnotherType {
    // Можно определить следующие методы:
    private static void M() {}
    private static void M<T>() {}
    private static void M<T1, T2>() {}

    // Ошибка: конфликт с типом M<T>, у которого нет ограничений
    private static void M<T>() where T : Comparable<T> {}

    // Ошибка: конфликт с типом M<T1, T2>.
    private static void M<T3, T4>() {}
}
```

При переопределении виртуального обобщенного метода в переопределяющем методе нужно задавать то же число параметров типа, а они, в свою очередь, наследуют ограничения, заданные для них методом базового класса. Честно говоря, переопределяемый метод вообще не вправе задавать ограничения для своих параметров типа, но может переименовывать параметры типа. Аналогично, при реализации интерфейсного метода в нем должно задаваться то же число параметров типа, что и в интерфейсном методе, причем эти параметры типа наследуют ограничения, заданные для них методом интерфейса. Следующий пример демонстрирует это правило с помощью виртуальных методов:

```
internal class Base {
    public virtual void M<T1, T2>()
```

продолжение ➤

```

where T1 : struct
where T2 : class {
}
}

internal sealed class Derived : Base {
    public override void M<T3, T4>()
    where T3 : EventArgs // Ошибка
    where T4 : class      // Ошибка
    { }
}

```

При компиляции этого кода появится сообщение об ошибке (ошибка CS0460: ограничения для методов интерфейсов с переопределением и явной реализацией наследуются от базового метода и поэтому не могут быть заданы явно):

Error CS0460: Constraints for override and explicit interface implementation methods are inherited from the base method so cannot be specified directly

Если из метода `M<T3, T4>` класса `Derived` убрать две строки `where`, код успешно компилируется. Заметьте: разрешается переименовывать параметры типа (в этом примере имя `T1` изменено на `T3`, а `T2` — на `T4`), но изменять (и даже задавать) ограничения нельзя.

Теперь поговорим о различных типах ограничений, которые компилятор и CLR позволяют применять к параметрам типа. К параметру типа могут применяться следующие ограничения: *основное* (primary), *дополнительное* (secondary) и/или *ограничение конструктора* (constructor constraint). Речь о них идет в следующих трех разделах.

Основные ограничения

В параметре типа можно задать не более одного основного ограничения. Основным ограничением может быть ссылочный тип, указывающий на изолированный класс. Нельзя использовать для этой цели следующие ссылочные типы: `System.Object`, `System.Array`, `System.Delegate`, `System.MulticastDelegate`, `System.ValueType`, `System.Enum` и `System.Void`.

При задании ограничения ссылочного типа вы обязуетесь перед компилятором, что любой аргумент типа будет иметь либо тот же тип, что и ограничение, либо производный от него тип. Например, как в этом обобщенном классе:

```

internal sealed class PrimaryConstraintOfStream<T> where T : Stream {
    public void M(T stream) {
        stream.Close();// OK
    }
}

```

В этом определении класса на параметр типа `T` наложено основное ограничение `Stream` (из пространства имен `System.IO`), сообщаящее компилятору, что код, использующий `PrimaryConstraintOfStream`, должен задавать аргумент типа `Stream` или производного от него типа (например, `FileStream`). Когда параметр типа не задает основное ограничение, автоматически задается тип `System.Object`. Однако если в исходном тексте явно указать `System.Object`, компилятор `C#` выдаст ошибку (ошибка `CS0702`: ограничение не может быть конкретным классом `object`):

```
error CS0702: Constraint cannot be special class 'object'
```

Есть два особых основных ограничения: `class` и `struct`. Ограничение `class` гарантирует компилятору, что указанный аргумент типа будет иметь ссылочный тип. Этому ограничению удовлетворяют все типы-классы, типы-интерфейсы, типы-делегаты и типы-массивы, как в следующем обобщенном классе:

```
internal sealed class PrimaryConstraintOfClass<T> where T : class {  
    public void M() {  
        T temp = null; // Допустимо, потому что тип T должен быть ссылочным  
    }  
}
```

В этом примере присвоение `temp` значения `null` допустимо, потому что известно, что `T` имеет ссылочный тип, а любая переменная ссылочного типа может быть равна `null`. При отсутствии у `T` ограничений этот код бы не скомпилировался, потому что тип `T` мог бы быть значимым, а переменные значимого типа нельзя приравнивать к `null`.

Ограничение `struct` гарантирует компилятору, что указанный аргумент типа будет иметь значимый тип. Этому ограничению удовлетворяют все значимые типы, а также перечисления. Однако компилятор и CLR рассматривают любой значимый тип `System.Nullable<T>` как особый, и значимые типы с поддержкой значения `null` не подходят под это ограничение. Это объясняется тем, что для параметра типа `Nullable<T>` действует ограничение `struct`, а среда CLR запрещает такие рекурсивные типы, как `Nullable<Nullable<T>>`. Значимые типы с поддержкой значения `null` обсуждаются в главе 19.

Вот пример класса, где параметр типа ограничивается с помощью ключевого слова `struct`:

```
internal sealed class PrimaryConstraintOfStruct<T> where T : struct {  
    public static T Factory() {  
        // Допускается, потому что у каждого значимого типа неявно  
        // есть открытый конструктор без параметров  
        return new T();  
    }  
}
```


В этом примере применение к `T` оператора `new` правомерно, потому что известно, что `T` имеет значимый тип, а у всех значимых типов неявно есть открытый конструктор без параметров. Если бы тип `T` был неограниченным, ограниченным ссылочным типом или ограниченным классом, этот код не скомпилировался бы, потому что у некоторых ссылочных типов нет открытых конструкторов без параметров.

Дополнительные ограничения

В параметре типа можно задать несколько дополнительных ограничений или не задавать их вообще. При задании ограничения типа-интерфейса компилятору гарантируется, что указанный аргумент типа будет иметь тип, реализующий этот интерфейс. А так как можно задать несколько ограничений интерфейса, в аргументе типа должен указываться тип, реализующий все ограничения интерфейса (и все основные ограничения, если они есть). Подробнее об ограничениях интерфейса см. главу 13.

Другой тип дополнительных ограничений называют *ограничением параметра типа* (type parameter constraint). Оно используется гораздо реже, чем ограничение интерфейса, и позволяет обобщенному типу или методу указать, что между указанными аргументами типа должны быть определенные отношения. К параметру типа может быть применено несколько или ни одного ограничения типа. В следующем обобщенном методе показано использование ограничения параметра типа:

```
private static List<TBase> ConvertIList<T, TBase>(IList<T> list)
    where T : TBase {
    List<TBase> baseList = new List<TBase>(list.Count);
    for (Int32 index = 0; index < list.Count; index++) {
        baseList.Add(list[index]);
    }
    return baseList;
}
```

В методе `ConvertIList` определены два параметра типа, из которых параметр `T` ограничен параметром типа `TBase`. Это значит, что какой бы аргумент типа ни был задан для `T`, он должен быть совместим с аргументом типа, заданным для `TBase`.

В следующем методе показаны допустимые и недопустимые вызовы `ConvertIList`:

```
private static void CallingConvertIList() {
    // Создает и инициализирует тип List<String> (реализующий IList<String>)
    IList<String> ls = new List<String>();
    ls.Add("A String");

    // Преобразует IList<String> в IList<Object>
    IList<Object> lo = ConvertIList<String, Object>(ls);
}
```

```
// Преобразует IList<String> в IList<IComparable>
IList<IComparable> lc = ConvertIList<String, IComparable>(ls);

// Преобразует IList<String> в IList<IComparable<String>>
IList<IComparable<String>> lcs =
    ConvertIList<String, IComparable<String>>(ls);

// Преобразует IList<String> в IList<String>
IList<String> ls2 = ConvertIList<String, String>(ls);

// Преобразует IList<String> в IList<Exception>
IList<Exception> le = ConvertIList<String, Exception>(ls); // Ошибка
}
```

В первом вызове `ConvertIList` компилятор проверяет, чтобы тип `String` был совместим с `Object`. Поскольку тип `String` является производным от `Object`, первый вызов удовлетворяет ограничению параметра типа. Во втором вызове `ConvertIList` компилятор проверяет, чтобы тип `String` был совместим с `IComparable`. Поскольку `String` реализует интерфейс `IComparable`, второй вызов соответствует ограничению параметра типа. В третьем вызове `ConvertIList` компилятор проверяет, чтобы тип `String` был совместим с `IComparable<String>`. Так как `String` реализует интерфейс `IComparable<String>`, третий вызов соответствует ограничению параметра типа. В четвертом вызове `ConvertIList` компилятор знает, что тип `String` совместим сам с собой. В пятом вызове `ConvertIList` компилятор проверяет, чтобы тип `String` был совместим с `Exception`. Однако так как тип `String` несовместим с `Exception`, пятый вызов не соответствует ограничению параметра типа, и компилятор возвращает ошибку (ошибка CS0309: тип `string` должен быть приведен к `System.Exception`, чтобы использоваться в качестве параметра `T` в обобщенном коде или в методе `SomeType.ConvertIList<T, TBase>(System.Collections.Generic.IList<T>)`, а неявное приведение типа `string` к `System.Exception` отсутствует):

```
error CS0309: The type 'string' must be converted to 'System.Exception' in
order to use it as parameter 'T' in the generic type or method SomeType.
ConvertIList<T, TBase>(System.Collections.Generic.IList<T>). There is no
implicit reference conversion from 'string' to 'System.Exception'
```

Ограничения конструктора

В параметре типа можно задать не более одного ограничения конструктора. Ограничение конструктора указывает компилятору, что указанный аргумент типа будет иметь неабстрактный тип, реализующий открытый конструктор без параметров. Обратите внимание, что компилятор C# считает за ошибку одновременное задание ограничения конструктора и ограничения `struct`, потому что это избыточно. У всех значимых типов неявно присутствует открытый

конструктор без параметров. В следующем классе для параметров типа использовано ограничение конструктора:

```
internal sealed class ConstructorConstraint<T> where T : new() {
    public static T Factory() {
        // Допустимо, потому что у всех значимых типов неявно
        // есть открытый конструктор без параметров и потому что
        // это ограничение требует, чтобы у всех указанных ссылочных типов
        // также был открытый конструктор без параметров
        return new T();
    }
}
```

В этом примере применение оператора `new` к `T` допустимо, потому что известно, что `T` — это тип с открытым конструктором без параметров. Разумеется, это справедливо и для всех значимых типов, а ограничение конструктора требует, чтобы это условие выполнялось и для всех ссылочных типов, заданных как аргументы типа.

Иногда разработчики предпочитают объявлять параметр типа через ограничение конструктора, при котором сам конструктор принимает различные параметры. На сегодняшний день CLR (и, как следствие, компилятор C#) поддерживает только конструкторы без параметров. По мнению специалистов компании Microsoft, в большинстве случаев этого вполне достаточно, и я с ними полностью согласен.

Другие проблемы верификации

В оставшейся части этой главы я представлю несколько кодов, которые из-за проблем с верификацией ведут себя непредсказуемо при использовании с обобщениями, и покажу, как с помощью ограничений сделать их верифицируемыми.

Приведение переменной обобщенного типа

Приведение переменной обобщенного типа к другому типу допускается, лишь если она приводится к типу, разрешенному ограничением:

```
private static void CastingAGenericTypeVariable1<T>(T obj) {
    Int32 x = (Int32) obj; // Ошибка
    String s = (String) obj; // Ошибка
}
```

Компилятор вернет ошибку для обеих строк, потому что `T` может иметь любой тип и успех приведения типа не гарантирован. Чтобы этот код скомпилировался, его нужно изменить, добавив в начале приведение к `Object`:

```
private static void CastingAGenericTypeVariable2<T>(T obj) {
    Int32 x = (Int32) (Object) obj; // Ошибки нет
    String s = (String) (Object) obj; // Ошибки нет
}
```

Теперь этот код скомпилируется, но во время выполнения CLR все равно может сгенерировать исключение `InvalidCastException`.

Для приведения к ссылочному типу также применяют оператор `as` языка C#. В следующем коде он используется с типом `String` (поскольку `Int32` — значимый тип):

```
private static void CastingAGenericTypeVariable3<T>(T obj) {  
    String s = obj as String; // Ошибки нет  
}
```

Присваивание переменной обобщенного типа значения по умолчанию

Приравнивание переменной обобщенного типа к `null` допустимо, только если обобщенный тип ограничен ссылочным типом:

```
private static void SettingAGenericTypeVariableToNull<T>() {  
    T temp = null; // CS0403: нельзя привести null к параметру типа T  
    // because it could be a value type...  
    // (Ошибка CS0403: нельзя привести null к параметру типа T,  
    // поскольку T может иметь значимый тип...)  
}
```

Так как параметр типа `T` не ограничен, он может иметь значимый тип, а приравнять переменную значимого типа к `null` нельзя. Если бы параметр типа `T` был ограничен ссылочным типом, `temp` можно было бы приравнять к `null`, и код скомпилировался бы и работал. При создании C# в Microsoft посчитали, что разработчикам может понадобиться присвоить переменной значение по умолчанию. Для этого в компиляторе C# есть ключевое слово `default`:

```
private static void SettingAGenericTypeVariableToDefaultValue<T>() {  
    T temp = default(T); // Работает  
}
```

В этом примере ключевое слово `default` дает команду компилятору C# и JIT-компилятору CLR создать код, приравнивающий `temp` к `null`, если `T` имеет ссылочный тип, и обнуляющий все биты переменной `temp`, если `T` имеет значимый тип.

Сравнение переменной обобщенного типа с null

Сравнение переменной обобщенного типа с `null` с помощью операторов `==` и `!=` допустимо независимо от того, ограничен обобщенный тип или нет:

```
private static void ComparingAGenericTypeVariableWithNull<T>(T obj) {  
    if (obj == null)  
    { /* Этот код никогда не исполнится, если тип значимый */ }  
}
```

Так как тип `T` не ограничен, он может быть ссылочный или значимый. Во втором случае `obj` нельзя приравнять `null`. Обычно в этом случае компилятор `C#` должен выдать ошибку. Но этого не происходит — код успешно компилируется. При вызове этого метода с аргументом значимого типа JIT-компилятор, обнаружив, что результат выполнения инструкции `if` никогда не равен `true`, просто не создаст машинный код для инструкции `if` и кода в фигурных скобках. Если бы я использовал оператор `!=`, JIT-компилятор также не сгенерировал бы код для инструкции `if` (поскольку ее значение всегда `true`), но сгенерировал бы код внутри фигурных скобок после `if`.

Кстати, если к `T` применить ограничение `struct`, компилятор `C#` не вернет ошибку, потому что не нужно создавать код, сравнивающий значимый тип с `null`, — результат всегда один.

Сравнение двух переменных обобщенного типа

Сравнение двух переменных одинакового обобщенного типа допустимо только в том случае, если обобщенный параметр типа имеет ссылочный тип:

```
private static void ComparingTwoGenericTypeVariables<T>(T o1, T o2) {
    if (o1 == o2) { } // Ошибка
}
```

В этом примере у `T` нет ограничений, и хотя можно сравнивать две переменные ссылочного типа, сравнивать две переменные значимого типа допустимо лишь в том случае, когда значимый тип перегружает оператор `==`. Если у `T` есть ограничение `class`, этот код скомпилируется, а оператор `==` вернет значение `true`, если переменные ссылаются на один объект и полностью тождественны. Заметьте: если параметр `T` ограничен ссылочным типом, перегружающим метод `operator==`, компилятор сгенерирует вызовы этого метода при виде оператора `==`. Ясно, что все сказанное относится и к оператору `!=`.

При написании кода для сравнения элементарных значимых типов (`Byte`, `Int32`, `Single`, `Decimal` и т. д.) компилятор `C#` сгенерирует код правильно, но для других значимых типов генерировать код сравнения он не умеет. Поэтому если у параметра `T` метода `ComparingTwoGenericTypeVariables` есть ограничение `struct`, компилятор выдаст ошибку. А ограничивать параметр типа значимым типом нельзя, потому что они неявно являются изолированными. Теоретически этот метод можно скомпилировать, задав в качестве ограничения конкретный значимый тип, но в таком случае метод перестанет быть обобщенным. Он будет привязан к конкретному типу данных, и, конечно, компилятор не скомпилирует обобщенный метод, ограниченный одним типом.

Использование переменных обобщенного типа в качестве операндов

И, наконец, замечу, что немало трудностей несет в себе использование операторов с операндами обобщенного типа. В главе 5 я показал, как `C#` обрабатывает элементарные типы — `Byte`, `Int16`, `Int32`, `Int64`, `Decimal` и др. В частности, я отметил, что `C#` умеет интерпретировать операторы, применяемые к элементар-

ным типам (например +, -, * и /). Однако эти операторы нельзя использовать с переменными обобщенного типа, потому что во время компиляции компилятор не знает их тип. Получается, что нельзя спроектировать математический алгоритм для произвольных числовых типов данных. Я написал следующий обобщенный метод:

```
private static T Sum<T>(T num) where T : struct {  
    T sum = default(T) ;  
    for (T n = default(T) ; n < num ; n++)  
        sum += n;  
    return sum;  
}
```

Я также сделал все возможное, чтобы он скомпилировался: определил ограничение struct для T и использовал конструкцию default(T), чтобы sum и n инициализировались нулем. Но при компиляции кода появились три сообщения об ошибках:

- ❑ ошибка CS0019: оператор < нельзя применять к операндам типа T и T:
error CS0019: Operator '<' cannot be applied to operands
of type 'T' and 'T'
- ❑ ошибка CS0023: оператор ++ нельзя применять к операнду типа T:
error CS0023: Operator '++' cannot be applied to operand of type 'T'
- ❑ ошибка CS0019: оператор += нельзя применять к операндам типа T и T:
error CS0019: Operator '+=' cannot be applied to operands
of type 'T' and 'T'

Это существенно ограничивает поддержку обобщений в среде CLR, и многие разработчики (особенно из научных и математических кругов) испытали глубокое разочарование. Многие пытались создать методы, призванные обойти это ограничение, прибегая к отражению (см. главу 23), перегрузке операторов и т. п. Однако все эти решения сильно снижают производительность или ухудшают читаемость кода. Остается надеяться, что в следующих версиях CLR и компиляторов Microsoft устранил этот недостаток.

Глава 13. Интерфейсы

Многие программисты знакомы с принципами *множественного наследования* (multiple inheritance) — возможности определять класс, производный от двух или более базовых классов. Представьте себе один класс TransmitData, назначением которого является передача данных, и второй класс ReceiveData, обеспечивающий получение данных. Допустим, нужно создать класс SocketPort, который может и получать, и передавать данные. Чтобы добиться этого, надо сделать класс SocketPort наследником одновременно обоих классов: TransmitData и ReceiveData.

Некоторые языки программирования разрешают множественное наследование, позволяя создать класс SocketPort, наследующий от двух базовых классов. Однако CLR, а значит, и все основанные на этой среде языки программирования, множественное наследование не поддерживают. Вместе с тем CLR позволяет реализовать ограниченное множественное наследование через *интерфейсы* (interfaces). В этой главе рассказывается об определении и применении интерфейсов, а также приводятся основные правила, позволяющие понять, когда нужно использовать интерфейсы, а не базовые классы.

Наследование в классах и интерфейсах

В .NET Framework есть класс System.Object, в котором определено 4 открытых экземплярных метода: ToString, Equals, GetHashCode и GetType. Этот класс является корневым, или основным, базовым классом для всех остальных классов, поэтому все классы наследуют эти четыре метода класса Object. Это также означает, что код, оперирующий экземпляром класса Object, в действительности может выполнять операции с экземпляром любого класса.

Любой производный от Object класс наследует следующее:

- **Сигнатуры методов.** Это позволяет коду считать, что он оперирует экземпляром класса Object, тогда как на самом деле он работает с экземпляром какого-либо другого класса.
- **Реализацию этих методов.** Разработчик может определить класс, производный от Object, не реализуя методы класса Object вручную.

В CLR у класса может быть один и только один прямой «родитель» (который прямо или косвенно, но в конечном итоге наследует от класса Object). Базовый класс предоставляет набор сигнатур и реализации этих методов. И, что интересно в определении нового класса, — он может стать базовым классом для другого класса, который будет определен другим разработчиком, и при этом новый производный класс унаследует все сигнатуры методов и их реализации.

CLR также позволяет определить *интерфейс*, который, в сущности, представляет собой средство задания имени набору сигнатур методов. Интерфейс не содержит реализаций методов. Класс наследует интерфейс через указание имени последнего, причем этот класс должен явно содержать реализации интерфейсных методов — только в этом случае CLR посчитает определение типа надлежащим. Конечно, реализация интерфейсных методов — обычно довольно утомительное занятие, поэтому я и назвал наследование интерфейсов ограниченным механизмом реализации множественного наследования. Компилятор C# и CLR позволяют классу наследовать несколько интерфейсов, и, конечно же, класс должен реализовать все наследуемые через интерфейс методы.

Одна из замечательных особенностей наследования классов — возможность подставлять экземпляры производного типа в любые контексты, в которых выступают экземпляры базового типа. Аналогично, наследование интерфейсов позволяет подставлять экземпляры типа, реализующего интерфейс, во все контексты, где требуются экземпляры указанного интерфейсного типа. Обратимся к конкретике — практическому определению интерфейсов.

Определение интерфейсов

Как уже отмечалось, интерфейс — это именованный набор сигнатур методов. Обратите внимание, что в интерфейсах можно также определять события, свойства — без параметров или с ними (последние в C# называют индексаторами), поскольку все это — просто средства упрощения синтаксиса, который обеспечивает сопоставление методов. Однако в интерфейсе нельзя определять методы-конструкторы и экземплярные поля.

Хотя CLR допускает наличие в интерфейсах статических методов, полей и конструкторов, а также констант, CLS-совместимый интерфейс не может иметь подобных статических членов, поскольку некоторые языки не поддерживают их определение или обращение к ним. На самом деле, C# не позволяет определить в интерфейсе статические члены.

В C# для определения интерфейса, назначения ему имени и набора сигнатур экземплярных методов используется ключевое слово `interface`. Вот определения некоторых интерфейсов из библиотеки классов Framework Class Library:

```
public interface IDisposable {  
    void Dispose();  
}
```

```
public interface IEnumerable {  
    IEnumerator GetEnumerator();  
}
```



```

public interface IEnumerable<T> : IEnumerable {
    IEnumerator<T> GetEnumerator();
}

public interface ICollection<T> : IEnumerable<T>, IEnumerable {
    void Add(T item);
    void Clear();
    Boolean Contains(T item);
    void CopyTo(T[] array, Int32 arrayIndex);
    Boolean Remove(T item);
    Int32 Count { get; } // Свойство только для чтения
    Boolean IsReadOnly { get; } // Свойство только для чтения
}

```

С точки зрения CLR, определение интерфейса — почти то же, что и определение типа. То есть CLR определяет внутреннюю структуру данных для объекта интерфейсного типа, а для обращения к различным членам интерфейса может использовать отражение. Как и типы, интерфейс может определяться на уровне файлов или быть вложенным в другой тип. При определении интерфейсного типа можно указать требуемую область видимости и доступа (`public`, `protected`, `internal` и т. п.).

В соответствии с соглашением имени интерфейсных типов начинаются с прописной буквы `I`, что облегчает их поиск в исходном коде. CLR поддерживает обобщенные интерфейсы (как показано в некоторых предыдущих примерах) и интерфейсные методы. В этой главе я лишь слегка касаюсь некоторых возможностей обобщенных интерфейсов, детали см. в главе 12.

Определение интерфейса может «наследовать» другие интерфейсы. Однако слово «наследовать» не совсем точное, поскольку в интерфейсах наследование работает иначе, чем в классах. Я предпочитаю рассматривать наследование интерфейсов как включение контрактов других интерфейсов. Например, определение интерфейса `ICollection<T>` включает контракт интерфейсов `IEnumerable<T>` и `IEnumerable`. Это означает следующее:

- ❑ любой класс, наследующий интерфейс `ICollection<T>`, должен реализовать все методы, определенные в интерфейсах `ICollection<T>`, `IEnumerable<T>` и `IEnumerable`;
- ❑ любой код, ожидающий объект, тип которого реализует интерфейс `ICollection<T>`, вправе ожидать, что тип объекта также реализует методы интерфейсов `IEnumerable<T>` и `IEnumerable`.

Наследование интерфейсов

Сейчас я покажу, как определить тип, реализующий интерфейс, создать экземпляр этого типа и использовать полученный объект для вызова интерфейсных

методов. На самом деле, в C# это очень просто, но происходящее за кулисами чуть сложнее, но об этом — немного позже.

Интерфейс `System.IComparable<T>` определяется так (в `mscorlib.dll`):

```
public interface IComparable<T> {  
    Int32 CompareTo(T other);  
}
```

Следующий код демонстрирует, как определить тип, реализующий этот интерфейс, и код, сравнивающий два объекта `Point`:

using System;

```
// Объект Point является производным от System.Object  
// и реализует IComparable<T> в Point  
public sealed class Point : IComparable<Point> {  
    private Int32 m_x, m_y;  
    public Point(Int32 x, Int32 y) {  
        m_x = x;  
        m_y = y;  
    }  
  
    // Этот метод реализует IComparable<T> в Point  
    public Int32 CompareTo(Point other) {  
        return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)  
            - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));  
    }  
  
    public override String ToString() {  
        return String.Format("({0}, {1})", m_x, m_y);  
    }  
}  
  
public static class Program {  
    public static void Main() {  
        Point[] points = new Point[] {  
            new Point(3, 3),  
            new Point(1, 2),  
        };  
  
        // Вызов метода CompareTo интерфейса IComparable<T> объекта Point  
        if (points[0].CompareTo(points[1]) > 0) {  
            Point tempPoint = points[0];  
            points[0] = points[1];  
            points[1] = tempPoint;  
        }  
  
        Console.WriteLine("Points from closest to (0, 0) to farthest:");  
        foreach (Point p in points)  
            Console.WriteLine(p);  
    }  
}
```

Компилятор C# требует, чтобы метод, реализующий интерфейс, отмечался модификатором `public`. CLR требует, чтобы интерфейсные методы были виртуальными. Если метод явно не определен в коде как виртуальный, компилятор сделает его таковым и, вдобавок, изолированным. Это не позволяет производному классу переопределять интерфейсные методы. Если явно задать метод как виртуальный, компилятор сделает его таковым и оставит неизолированным, что предоставит производному классу возможность переопределять интерфейсные методы.

Производный класс не в состоянии переопределять интерфейсные методы, объявленные изолированными, но может повторно унаследовать тот же интерфейс и предоставить собственную реализацию его методов. При вызове интерфейсного метода объекта вызывается реализация, связанная с типом самого объекта. Вот пример, демонстрирующий это:

```
using System;
```

```
public static class Program {
    public static void Main() {
        /***** Первый пример *****/
        Base b = new Base();

        // Вызов реализации Dispose в типе b: "Dispose класса Base"
        b.Dispose();

        // Вызов реализации Dispose в типе объекта b: "Dispose класса Base"
        ((IDisposable)b).Dispose();

        /***** Второй пример *****/
        Derived d = new Derived();

        // Вызов реализации Dispose в типе d: "Dispose класса Derived"
        d.Dispose();

        // Вызов реализации Dispose в типе объекта d: "Dispose класса Derived"
        ((IDisposable)d).Dispose();

        /***** Третий пример *****/
        b = new Derived();
        // Вызов реализации Dispose в типе b: "Dispose класса Base"
        b.Dispose();

        // Вызов реализации Dispose в типе объекта b: "Dispose класса Derived"
        ((IDisposable)b).Dispose();
    }
}
```

```
// Этот класс наследует от класса Object и реализует IDisposable
internal class Base : IDisposable {
    // Этот метод неявно изолирован и его нельзя переопределить
```

```
public void Dispose() {
    Console.WriteLine("Base's Dispose");
}
}

// Этот класс наследует от базового и повторно реализует IDisposable
internal class Derived : Base, IDisposable {
    // Этот метод не может переопределить базовый метод Dispose
    // Ключевое слово 'new' указывает на то, что этот метод
    // повторно реализует метод Dispose интерфейса IDisposable
    new public void Dispose() {
        Console.WriteLine("Derived's Dispose");

        // ПРИМЕЧАНИЕ: следующая строка кода показывает,
        // как вызвать реализацию базового класса (если нужно)
        // base.Dispose();
    }
}
```

Подробнее о вызовах интерфейсных методов

Тип `System.String` из библиотеки FCL наследует сигнатуры и реализации методов `System.Object`. Кроме того, тип `String` реализует несколько интерфейсов: `Comparable`, `Cloneable`, `Convertible`, `Enumerable`, `Comparable<String>`, `Enumerable<Char>` и `IEnumerable<String>`. Это значит, что типу `String` не требуется реализовывать (или переопределять) методы, имеющиеся в его базовом типе `Object`. Однако тип `String` должен реализовывать методы, объявленные во всех интерфейсах.

CLR допускает определение поля, параметра или локальных переменных, имеющих интерфейсный тип. Используя переменную интерфейсного типа, можно вызывать методы, определенные этим интерфейсом. К тому же CLR позволяет вызывать методы, определенные в типе `Object`, поскольку все классы наследуют его методы, как продемонстрировано в следующем коде:

```
// Переменная s ссылается на объект String
String s = "Jeffrey";
// Используя s, я могу вызывать любой метод,
// определенный в String, Object, Comparable, Cloneable,
// Convertible, Enumerable и т. д.

// Переменная cloneable ссылается на тот же объект String
ICloneable cloneable = s;
// Используя переменную cloneable, я могу вызвать любой метод.
```

продолжение ➤

```
// объявленный только в интерфейсе ICloneable (или любой метод,
// определенный в типе Object)

// Переменная comparable ссылается на тот же объект String
IComparable comparable = s;
// Используя переменную comparable, я могу вызвать любой метод,
// объявленный только в интерфейсе IComparable (или любой метод,
// определенный в типе Object)

// Переменная enumerable ссылается на тот же объект String
// Во время выполнения можно приводить интерфейсную переменную
// к интерфейсу другого типа, если тип объекта реализует оба интерфейса
IEnumerable enumerable = (IEnumerable) comparable;
// Используя переменную enumerable, я могу вызывать любой метод,
// объявленный только в интерфейсе IEnumerable (или любой метод,
// определенный только в типе Object)
```

Все переменные в этом коде ссылаются на один объект String в управляемой куче, а значит, любой метод, который я вызываю с использованием любой из этих переменных, задействует один объект String, хранящий строку "Jeffrey". Однако тип переменной определяет действие, которое я могу выполнить с объектом. Переменная *s* имеет тип String, значит, она позволяет вызвать любой член, определенный в типе String (например, свойство Length). Переменную *s* можно также использовать для вызова любых методов, унаследованных от типа Object (например, GetType).

Переменная *cloneable* имеет тип интерфейса ICloneable, а значит, позволяет вызывать метод Clone, определенный в этом интерфейсе. Кроме того, можно вызвать любой метод, определенный в типе Object (например, GetType), поскольку CLR «знает», что все типы порождаются типом Object. Однако переменная *cloneable* не позволяет вызывать открытые методы, определенные в любом другом интерфейсе, реализованном типом String. Аналогично этому, используя переменную *comparable*, можно вызвать CompareTo или любой метод, определенный в типе Object, но не другие методы.

ВНИМАНИЕ

Как и ссылочный тип, значимый тип может реализовать несколько (или нуль) интерфейсов. Но при приведении экземпляра значимого типа к интерфейсному типу этот экземпляр надо упаковать, потому что переменная интерфейса является ссылкой, которая должна указывать на объект в куче, чтобы среда CLR могла проверить указатель и точно выяснить тип объекта. Затем при вызове метода интерфейса с упакованным значимым типом CLR использует указатель, чтобы найти таблицу методов типа объекта и вызвать нужный метод.

Явные и неявные реализации интерфейсных методов (что происходит за кулисами)

Когда тип загружается в CLR, для типа создается и инициализируется таблица методов (см. главу 1). Она содержит по одной записи для каждого нового, представляемого только этим типом метода, а также записи для всех методов, унаследованных типом. Унаследованные методы включают методы, определенные в базовых типах иерархии наследования, а также все методы, определенные интерфейсными типами. Пусть простой тип определен так:

```
internal sealed class SimpleType : IDisposable {  
    public void Dispose() { Console.WriteLine("Dispose"); }  
}
```

Тогда таблица методов типа содержит записи, в которых представлены:

- ❑ все экземплярные методы, определенные в типе `Object` и неявно унаследованные от этого базового класса;
- ❑ все интерфейсные методы, определенные в явно унаследованном интерфейсе `IDisposable` (в нашем примере в интерфейсе `IDisposable` определен только один метод — `Dispose`);
- ❑ новый метод, `Dispose`, появившийся в типе `SimpleType`.

Чтобы упростить жизнь программиста, компилятор C# считает, что появившийся в типе `SimpleType` метод `Dispose` является реализацией метода `Dispose` из интерфейса `IDisposable`. Компилятор C# вправе сделать такое предположение, потому что метод открытый, а сигнатуры интерфейсного метода и нового метода совпадают. Значит, методы принимают и возвращают одинаковые типы. Кстати, если бы новый метод `Dispose` был помечен как виртуальный, компилятор C# все равно сопоставил бы этот метод одноименному интерфейсному методу.

Сопоставляя новый метод интерфейсному методу, компилятор C# генерирует метаданные, указывающие на то, что обе записи в таблице методов типа `SimpleType` должны ссылаться на одну реализацию. Поясним это в коде, демонстрирующем вызов открытого метода `Dispose` класса, а также вызов реализации класса для метода `Dispose` интерфейса `IDisposable`.

```
public sealed class Program {  
    public static void Main() {  
        SimpleType st = new SimpleType();  
  
        // Вызов реализации открытого метода Dispose  
        st.Dispose();  
    }  
}
```

продолжение ➤

```
// Вызов реализации метода Dispose интерфейса IDisposable
IDisposable d = st;
d.Dispose();
}
}
```

В первом вызове выполняется обращение к методу `Dispose`, определенному в типе `SimpleType`. Затем я определяю переменную `d` интерфейсного типа `IDisposable`. Я инициализирую переменную `d` ссылкой на объект `SimpleType`. Теперь при вызове `d.Dispose()` выполняется обращение к методу `Dispose` интерфейса `IDisposable`. Так как `C#` требует, чтобы открытый метод `Dispose` тоже был реализацией для метода `Dispose` интерфейса `IDisposable`, будет выполнен тот же код, и в этом примере вы не заметите какой-либо разницы. На выходе получим следующее:

```
Dispose
Dispose
```

Теперь я перепишу `SimpleType`, чтобы можно было увидеть разницу:

```
internal sealed class SimpleType : IDisposable {
    public void Dispose() { Console.WriteLine("public Dispose"); }
    void IDisposable.Dispose() { Console.WriteLine("IDisposable Dispose"); }
}
```

Не вызывая метод `Main`, мы можем просто перекомпилировать и запустить заново программу, и на выходе получим следующее:

```
public Dispose
IDisposable Dispose
```

Если в `C#` перед именем метода вы ставите имя интерфейса, в котором определен этот метод (в нашем примере — `IDisposable.Dispose`), то вы создаете *явную реализацию интерфейсного метода* (Explicit Interface Method Implementation, EIMI). Заметьте: когда в `C#` определяется явный интерфейсный метод, нельзя указывать область доступа (открытый или закрытый). Однако когда компилятор создает метаданные для метода, его область доступа оказывается закрытой (`private`), что запрещает любому коду использовать экземпляр класса, просто вызвав интерфейсный метод. Единственный способ вызвать интерфейсный метод — обратиться через переменную этого интерфейсного типа.

Также обратите внимание на то, что EIMI-метод не может быть виртуальным, а значит, его нельзя переопределить. Это происходит потому, что EIMI-метод в действительности не является частью объектной модели типа; это средство подключения интерфейса (набора вариантов поведения, или методов) к типу, которое не делает эти варианты поведения/методы очевидными. Если вам все это кажется немного несуразным, значит, вы все поняли *правильно*. Это и правда несуразно. Далее в этой главе я расскажу о причинах, способных побудить вас использовать EIMI.

Обобщенные интерфейсы

C# и CLR поддерживают обобщенные интерфейсы, что открывает разработчикам доступ ко многим замечательным возможностям. В этом разделе рассказывается о преимуществах обобщенных интерфейсов.

Во-первых, обобщенные интерфейсы значительно снижают время компиляции. Некоторые интерфейсы (такие как необобщенный `Comparable`) определяют методы, которые принимают или возвращают параметры типа `Object`. При вызове в коде методов таких интерфейсов можно передать ссылку на экземпляр любого типа. Однако обычно это не требуется. Приведем пример:

```
private void SomeMethod1() {
    Int32 x = 1, y = 2;
    Comparable c = x;

    // CompareTo ожидает Object.
    // но вполне допустимо передать переменную y типа Int32
    c.CompareTo(y); // Выполняется упаковка

    // CompareTo ожидает Object.
    // при передаче "2" (тип String) компиляция выполняется нормально,
    // но во время выполнения вбрасывается исключение ArgumentException
    c.CompareTo("2");
}
```

Ясно, что предпочтительнее обеспечить более строгий контроль типов в интерфейсном методе, поэтому FCL содержит обобщенный интерфейс `Comparable<T>`. Вот новая версия кода, измененная с учетом использования обобщенного интерфейса:

```
private void SomeMethod2() {
    Int32 x = 1, y = 2;
    Comparable<Int32> c = x;

    // CompareTo ожидает Object.
    // но вполне допустимо передать переменную y типа Int32
    c.CompareTo(y); // Выполняется упаковка

    // CompareTo ожидает Int32.
    // передача "2" (тип String) приводит к ошибке компиляции
    // с сообщением о невозможности привести тип String к Int32
    c.CompareTo("2"); // Ошибка
}
```

Второе преимущество обобщенных интерфейсов заключается в том, что при работе со значимыми типами требуется меньше операций упаковки. Заметьте:

в `SomeMethod1` необобщенный метод `CompareTo` интерфейса `IComparable` ожидает переменную типа `Object`; передача переменной `y` (значимый тип `Int32`) приводит к упаковке значения `y`. В `SomeMethod2` метод `CompareTo` обобщенного интерфейса `IComparable<T>` ожидает `Int32`; передача `y` выполняется по значению, поэтому упаковка не требуется.

ПРИМЕЧАНИЕ

В FCL определены необобщенные и обобщенные версии интерфейсов `IComparable`, `ICollection`, `IList`, `IDictionary` и некоторых других. Если вы определяете тип и хотите реализовать любой из этих интерфейсов, обычно нужно выбирать обобщенные версии. Необобщенные версии оставлены в FCL для обратной совместимости с кодом, написанным до того, как в .NET Framework появилась поддержка обобщений. Необобщенные версии также предоставляют пользователям механизм работы с данными более универсальным, но и менее безопасным образом.

Некоторые обобщенные интерфейсы наследуют необобщенные версии, так что в классе приходится реализовывать как обобщенную, так и необобщенную версии. Например, обобщенный интерфейс `IEnumerable<T>` наследует необобщенный интерфейс `IEnumerable`. Так что если класс реализует `IEnumerable<T>`, он должен также реализовать `IEnumerable`.

Иногда, при необходимости интеграции с другим кодом, вам придется реализовывать необобщенный интерфейс просто потому, что необобщенной версии не существует. В этом случае, если любой из интерфейсных методов принимает или возвращает тип `Object`, теряется контроль типов при компиляции, и значимые типы должны упаковываться. Можно в некоторой степени исправить эту ситуацию, действуя так, как описано далее в разделе «Совершенствование контроля типов за счет явной реализации интерфейсных методов».

Третьим преимуществом обобщенных интерфейсов является то, что класс может реализовать один интерфейс многократно, просто используя параметры различного типа. Вот пример, показывающий, насколько полезным это может быть:

```
using System;
```

```
// Этот класс реализует обобщенный интерфейс IComparable<T> дважды
public sealed class Number: IComparable<Int32>, IComparable<String> {
    private Int32 m_val = 5;
    // Этот метод реализует метод CompareTo интерфейса IComparable<Int32>
    public Int32 CompareTo(Int32 n) {
        return m_val.CompareTo(n);
    }

    // Этот метод реализует метод CompareTo интерфейса IComparable<String>
    public Int32 CompareTo(String s) {
```

```
        return m_val.CompareTo(Int32.Parse(s));
    }
}

public static class Program {
    public static void Main() {
        Number n = new Number();

        // Здесь я сравниваю значение n со значением 5 типа Int32
        IComparable<Int32> cInt32 = n;
        Int32 result = cInt32.CompareTo(5);

        // Здесь я сравниваю значение n со значением "5" типа String
        IComparable<String> cString = n;
        result = cString.CompareTo("5");
    }
}
```

Параметры интерфейса обобщенного типа могут быть также помечены как контравариантные или ковариантные, что позволяет более гибко использовать интерфейсы. Подробнее о контравариантности и ковариантности рассказывается в главе 12.

Обобщения и ограничения интерфейса

Я уже отмечал преимущества обобщенных интерфейсов. В этом разделе я рассказываю о пользе ограничения параметров обобщенных типов отдельными интерфейсами.

Первое преимущество состоит в том, что можно ограничить параметр обобщенного типа несколькими интерфейсами. В этом случае тип передаваемого параметра должен реализовывать *все* ограничения интерфейса. Вот пример:

```
public static class SomeType {
    private static void Test() {
        Int32 x = 5;
        Guid g = new Guid();

        // Компиляция этого вызова М выполняется без проблем,
        // поскольку Int32 реализует и IComparable, и IConvertible
        M(x);
    }
}
```

продолжение ➤

```

    // Компиляция этого вызова М приводит к ошибке, поскольку
    // Guid реализует IComparable, но не реализует IConvertible
    M(g);
}

// Параметр Т типа М ограничен для работы только с теми типами,
// которые реализованы в интерфейсах IComparable и IConvertible
private static Int32 M<T>(T t) where T : IComparable, IConvertible {
    ...
}
}

```

Это замечательно! При определении параметров метода каждый тип параметра указывает, что передаваемый аргумент должен иметь заданный тип параметра или быть производным от него. Если типом параметра является интерфейс, аргумент может быть любого типа класса, если только этот класс реализует заданный интерфейс. Использование нескольких ограничений интерфейса позволяет методу указывать, что передаваемый аргумент должен реализовывать несколько интерфейсов.

На самом деле, если мы ограничим Т до класса и двух интерфейсов, это будет означать, что типом передаваемого аргумента должен быть указанный базовый класс (или быть производный от него), а также что он должен реализовывать оба интерфейса.

Такая гибкость позволяет методу диктовать условия вызывающему коду, а в случае несоответствия ограничениям возникают ошибки компиляции.

Второе преимущество ограничений интерфейса — избавление от упаковки при передаче экземпляров значимых типов. В предыдущем фрагменте кода методу М передавался аргумент х (экземпляр типа Int32, то есть значимого типа). При передаче х в М упаковка не выполнялась. Если код метода М вызовет t.CompareTo(...), упаковка при вызове также не будет выполняться (упаковка может выполняться для аргументов, передаваемых в CompareTo).

В то же время если М объявить следующим образом, то для передачи х в М придется выполнять упаковку:

```

private static Int32 M(IComparable t) {
    ...
}

```

Для ограничений интерфейсов компилятор C# генерирует определенные IL-инструкции, которые вызывают интерфейсный метод для значимого типа напрямую, без упаковки. Кроме использования ограничений интерфейса нет другого способа заставить компилятор C# генерировать такие IL-инструкции, а, следовательно, во всех других случаях вызов интерфейсного метода для значимого типа всегда приводит к упаковке.

Реализация нескольких интерфейсов с одинаковыми сигнатурами и именами методов

Иногда нужно определить тип, реализующий несколько интерфейсов с методами, у которых совпадают имена и сигнатуры. Допустим, два интерфейса определены следующим образом:

```
public interface IWindow {  
    Object GetMenu();  
}
```

```
public interface IRestaurant {  
    Object GetMenu();  
}
```

Требуется определить тип, реализующий оба этих интерфейса. В этом случае нужно реализовать члены типа путем явной реализации методов:

```
// Этот тип является производным от System.Object  
// и реализует интерфейсы IWindow и IRestaurant  
public sealed class MarioPizzeria : IWindow, IRestaurant {  
    // Это реализация метода GetMenu интерфейса IWindow  
    Object IWindow.GetMenu() { ... }  
  
    // Это реализация метода GetMenu интерфейса IRestaurant  
    Object IRestaurant.GetMenu() { ... }  
  
    // Это метод GetMenu (необязательный).  
    // не имеющий с интерфейсом ничего общего  
    public Object GetMenu() { ... }  
}
```

Так как этот тип должен реализовывать несколько различных методов GetMenu, нужно сообщить компилятору C#, какой из методов GetMenu реализацию какого интерфейса содержит.

Код, в котором используется объект MarioPizzeria, должен выполнять приведение типа к определенному интерфейсу, чтобы вызвать желаемый метод:

```
MarioPizzeria mp = new MarioPizzeria();
```

```
// Эта строка вызывает открытый метод GetMenu класса MarioPizzeria  
mp.GetMenu();
```

продолжение ➤

```
// Эти строки вызывают метод IWindow.GetMenu
IWindow window = mp;
window.GetMenu();
```

```
// Эти строки вызывают метод IRestaurant.GetMenu
IRestaurant restaurant = mp;
restaurant.GetMenu();
```

Совершенствование контроля типов за счет явной реализации интерфейсных методов

Интерфейсы очень удобны, так как позволяют задать стандартный способ взаимодействия между типами. Ранее я говорил об обобщенных интерфейсах и о том, как они повышают безопасность типов при компиляции и позволяют избавиться от упаковки. К сожалению, иногда приходится реализовывать необобщенные интерфейсы, поскольку обобщенной версии попросту не существует. Если какой-либо из интерфейсных методов принимает параметры типа `System.Object` или возвращает значение типа `System.Object`, безопасность типов при компиляции нарушается и выполняется упаковка. В этом разделе я показываю, как за счет явной реализации интерфейсных методов (EIMI) можно несколько смягчить ситуацию.

Вот очень типичный интерфейс `Comparable`:

```
public interface Comparable {
    Int32 CompareTo(Object other);
}
```

В этом интерфейсе определяется единственный метод, который принимает параметр типа `System.Object`. Если я определяю собственный тип, реализующий этот интерфейс, определение типа будет выглядеть примерно так:

```
internal struct SomeValueType : Comparable {
    private Int32 m_x;
    public SomeValueType(Int32 x) { m_x = x; }
    public Int32 CompareTo(Object other) {
        return(m_x - ((SomeValueType) other).m_x);
    }
}
```

Используя `SomeValueType`, я могу написать следующий код:

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    Object o = new Object();
```

```

Int32 n = v.CompareTo(v); // Нежелательная упаковка
n = v.CompareTo(o);       // Исключение InvalidCastException
}

```

Стать «идеальным» этому коду не дают две проблемы:

- ❑ **нежелательная упаковка** — когда переменная *v* передается в качестве аргумента методу *CompareTo*, она должна упаковываться, поскольку *CompareTo* ожидает параметр типа *Object*;
- ❑ **отсутствие безопасности типов** — компиляция кода выполняется без проблем, но когда метод *CompareTo* пытается привести *other* к типу *SomeValueType*, вбрасывается исключение *InvalidCastException*.

Оба недостатка можно исправить средствами EIMI. Вот модифицированная версия типа *SomeValueType*, в которой имеет место явная реализация интерфейсных методов:

```

internal struct SomeValueType : IComparable {
    private Int32 m_x;
    public SomeValueType(Int32 x) { m_x = x; }
    public Int32 CompareTo(SomeValueType other) {
        return(m_x - other.m_x);
    }
}

// ПРИМЕЧАНИЕ: в следующей строке не используется public/private
Int32 IComparable.CompareTo(Object other) {
    return CompareTo((SomeValueType) other);
}
}

```

Обратите внимание на некоторые изменения в новой версии. Во-первых, здесь два метода *CompareTo*. Первый больше не принимает параметр типа *Object*, а принимает параметр типа *SomeValueType*. Поскольку параметр изменился, код, выполняющий приведение *other* к типу *SomeValueType*, стал ненужным и был удален. Во-вторых, изменение первого метода *CompareTo* для обеспечения безопасности типов приводит к тому, что *SomeValueType* больше не придерживается контракта, присутствующего по причине реализации интерфейса *IComparable*. Поэтому в *SomeValueType* нужно реализовать метод *CompareTo*, удовлетворяющий контракту *IComparable*. Этим занимается второй метод *CompareTo*, который реализован за счет явной реализации интерфейсных методов.

Эти два изменения обеспечили безопасность типов при компиляции и избавили от упаковки:

```

public static void Main() {
    SomeValueType v = new SomeValueType(0);
    Object o = new Object();
    Int32 n = v.CompareTo(v); // Без упаковки
    n = v.CompareTo(o);       // Ошибка компиляции
}

```

Однако если мы определим переменную интерфейсного типа, то потеряем безопасность типов при компиляции и опять вернемся к упаковке:

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    IComparable c = v; // Упаковка!

    Object o = new Object();
    Int32 n = c.CompareTo(v); // Нежелательная упаковка
    n = c.CompareTo(o);       // Исключение InvalidCastException
}
```

Как уже отмечалось, при приведении экземплярного типа к интерфейсному среда CLR должна упаковывать экземпляр значимого типа. Поэтому в приведенном методе `Main` выполняются две упаковки.

К EIMI часто прибегают при реализации таких интерфейсов, как `IConvertible`, `ICollection`, `IList` и `IDictionary`. Это позволяет обеспечить в интерфейсных методах безопасность типов при компиляции и избавиться от упаковки значимых типов.

Опасности явной реализации интерфейсных методов

Исключительно важно понимать некоторые особенности EIMI, из-за которых следует избегать явной реализации интерфейсных методов везде, где это возможно. К счастью, в некоторых случаях вместо EIMI можно обойтись обобщенными интерфейсами. Но все равно остаются ситуации, когда без EIMI не обойтись (например, при реализации двух интерфейсных методов с одинаковыми именами и сигнатурами). С явной реализацией интерфейсных методов связаны некоторые серьезные проблемы (далее я расскажу о них подробнее):

- ❑ отсутствие документации, объясняющей, как именно тип реализует EIMI-метод, а также отсутствие IntelliSense-поддержки в Microsoft Visual Studio;
- ❑ при приведении к интерфейсному типу экземпляры значимого типа упаковываются;
- ❑ EIMI нельзя вызвать из производного типа.

В части документации .NET Framework, касающейся методов типов, можно найти сведения о явной реализации методов интерфейсов, но справка по конкретным типам отсутствует — доступна только общая информация об интерфейсных методах. Например, о типе `Int32` говорится, что он реализует все интерфейсные методы `IConvertible`.

И это хорошо, потому что позволяет разработчикам узнать, что такие методы существуют, однако это же может смутить, потому что вызвать метод интерфейса `IConvertible` для `Int32` напрямую нельзя. Например, следующий метод не скомпилируется.

```
public static void Main() {  
    Int32 x = 5;  
    Single s = x.ToSingle(null); // Попытка вызвать метод  
                                // интерфейса IConvertible  
}
```

При компиляции этого метода компилятор C# вернет следующую ошибку (ошибка CS0117: int не содержит определения для ToSingle):

```
error CS0117: 'int' does not contain a definition for 'ToSingle'
```

Это сообщение об ошибке лишь запутывает разработчика, поскольку ясно говорит, что в типе Int32 метод ToSingle не определен, хотя на самом деле это неправда.

Чтобы вызвать метод ToSingle типа Int32, сначала следует привести его к типу IConvertible:

```
public static void Main() {  
    Int32 x = 5;  
    Single s = ((IConvertible) x).ToSingle(null);  
}
```

Требование приводить тип далеко не очевидно, многие разработчики не могут самостоятельно до этого додуматься. Но на этом проблемы не заканчиваются — при приведении значимого типа Int32 к интерфейсному типу IConvertible значимый тип упаковывается, на что тратится память и из-за чего снижается производительность. Это вторая серьезная проблема.

Третья и, наверное, самая серьезная проблема с EIMI состоит в том, что явная реализация интерфейсного метода не может вызываться из производного класса. Вот пример:

```
internal class Base : IComparable {  
    // Явная реализация интерфейсного метода (EIMI)  
    Int32 IComparable.CompareTo(Object o) {  
        Console.WriteLine("Base's CompareTo");  
        return 0;  
    }  
}  
  
internal sealed class Derived : Base, IComparable {  
    // Открытый метод, также являющийся реализацией интерфейса  
    public Int32 CompareTo(Object o) {  
        Console.WriteLine("Derived's CompareTo");  
        // Эта попытка вызвать EIMI базового класса приводит к ошибке:  
        // "error CS0117: 'Base' does not contain a definition for 'CompareTo'"  
        base.CompareTo(o);  
        return 0;  
    }  
}
```


В методе CompareTo типа Derived я попытался вызвать base.CompareTo, но это привело к ошибке компилятора C#. Проблема заключается в том, что в классе Base нет открытого или защищенного метода CompareTo, который он мог бы вызвать. Есть метод CompareTo, который можно вызвать только через переменную типа IComparable. Я мог бы изменить метод CompareTo класса Derived следующим образом:

// Открытый метод, который также является реализацией интерфейса

```
public Int32 CompareTo(Object o) {
    Console.WriteLine("Derived's CompareTo");

    // Эта попытка вызова EIMI базового класса приводит
    // к бесконечной рекурсии
    IComparable c = this;
    c.CompareTo(o);
    return 0;
}
```

В этой версии я привожу this к типу переменной c (типу IComparable), а затем использую c для вызова CompareTo. Однако открытый метод CompareTo класса Derived является реализацией метода CompareTo интерфейса IComparable класса Derived, поэтому возникает бесконечная рекурсия. Ситуацию можно исправить, объявив класс Derived без интерфейса IComparable:

```
internal sealed class Derived : Base /*, IComparable */ { ... }
```

Теперь предыдущий метод CompareTo вызовет метод CompareTo класса Base. Однако не всегда можно просто удалить интерфейс из типа, поскольку производный тип должен реализовывать интерфейсный метод. Лучший способ исправить ситуацию — в дополнение к явно реализованному интерфейсному методу создать в базовом классе виртуальный метод, который будет реализовываться явно. Затем в классе Derived можно переопределить виртуальный метод. Вот как правильно определять классы Base и Derived:

```
internal class Base : IComparable {
    // Явная реализация интерфейсного метода (EIMI)
    Int32 IComparable.CompareTo(Object o) {
        Console.WriteLine("Base's IComparable CompareTo");
        return CompareTo(o); // Теперь здесь вызывается виртуальный метод
    }

    // Виртуальный метод для производных классов
    // (этот метод может иметь любое имя)
    public virtual Int32 CompareTo(Object o) {
        Console.WriteLine("Base's virtual CompareTo");
        return 0;
    }
}

internal sealed class Derived : Base, IComparable {
    // Открытый метод, который также является реализацией интерфейса
```

```
public override Int32 CompareTo(Object o) {  
    Console.WriteLine("Derived's CompareTo");  
  
    // Теперь можно вызвать виртуальный метод класса Base  
    return base.CompareTo(o);  
}  
}
```

Заметьте: я определил виртуальный метод как открытый, но в некоторых случаях лучше сделать его защищенным. Это вполне возможно, хотя и потребует небольших изменений.

Как видите, к явной реализации интерфейсных методов нужно прибегать с осторожностью. Когда разработчики впервые узнали о EIMI, многие посчитали это отличной новостью и стали пытаться выполнять явную реализацию интерфейсных методов везде, где только можно. Не попадайтесь на эту удочку! Явная реализация интерфейсных методов полезна в лишь некоторых случаях, но ее следует избегать везде, где можно обойтись другими средствами.

Дилемма разработчика: базовый класс или интерфейс?

Меня часто спрашивают, что выбрать для реализации — базовый тип или интерфейс? Ответ не всегда очевиден. Вот несколько правил, которые могут помочь вам сделать выбор.

- ❑ **Связь потомка с предком.** Любой тип может наследовать только одну реализацию. Если производный тип не может ограничиваться функциональностью базового, нужно применять интерфейс, а не базовый тип. Например, тип может преобразовывать экземпляры самого себя в другой тип (IConvertible), может создать набор экземпляров самого себя (ISerializable) и т. д. Заметьте, что значимые типы должны наследовать от типа System.ValueType и поэтому не могут наследовать от произвольного базового класса. В этом случае нужно определять интерфейс.
- ❑ **Простота использования.** Разработчику проще определить новый тип, производный от базового, чем создать интерфейс. Базовый тип может предоставлять массу функций, и в производном типе потребуется внести лишь незначительные изменения, чтобы изменить его поведение. При создании интерфейса в новом типе придется реализовывать все члены.
- ❑ **Четкая реализация.** Как бы хорошо ни был документирован контракт, маловероятно, что он будет реализован абсолютно корректно. По сути, проблемы COM связаны именно с этим — вот почему некоторые COM-объекты нормально работают только с Microsoft Word или Microsoft Internet Explorer. Базовый тип с хорошей реализацией основных функций — прекрасная отправная точка, вам останется изменить лишь отдельные части.

- **Управление версиями.** Когда вы добавляете метод к базовому типу, производный тип наследует стандартную реализацию этого метода без всяких затрат. Пользовательский исходный код даже не нужно перекомпилировать. Добавление нового члена к интерфейсу требует изменения пользовательского исходного кода и его перекомпиляции.

В FCL классы, связанные с потоками данных, построены по принципу наследования реализации. `System.IO.Stream` — это абстрактный базовый класс, предоставляющий множество методов, в том числе `Read` и `Write`. Другие классы (`System.IO.FileStream`, `System.IO.MemoryStream` и `System.Net.Sockets.NetworkStream`) являются производными от `Stream`. В Microsoft выбрали такой вид отношений между этими тремя классами и `Stream` по той причине, что так проще реализовывать конкретные классы. Так, производные классы должны реализовать только операции синхронного ввода-вывода, а способность выполнять асинхронные операции они наследуют от базового класса `Stream`.

Возможно, выбор наследования реализации для классов, работающих с потоками, не совсем очевиден: ведь базовый класс `Stream` на самом деле предоставляет лишь ограниченную готовую функциональность. Однако если взглянуть на классы элементов управления Windows Forms, где `Button`, `CheckBox`, `ListBox` и все прочие элементы управления порождаются от `System.Windows.Forms.Control`, легко представить объем кода, реализованного в `Control`.

Что касается *коллекций* (collections), то их специалисты Microsoft реализовали в FCL на основе интерфейсов. В пространстве имен `System.Collections.Generic` определено несколько интерфейсов для работы с коллекциями: `IEnumerable<T>`, `ICollection<T>`, `IList<T>` и `IDictionary<TKey, TValue>`. Кроме того, Microsoft предлагает несколько конкретных классов (таких как `List<T>`, `Dictionary<TKey, TValue>`, `Queue<T>`, `Stack<T>` и пр.), которые реализуют комбинации этих интерфейсов. Такой подход объясняется тем, что реализация всех классов-коллекций существенно различается. Иначе говоря, у `List<T>`, `Dictionary<TKey, TValue>` и `Queue<T>` не так много совместно используемого кода.

И все же операции, предлагаемые всеми этими классами, вполне согласованы. Например, все они поддерживают подмножество элементов, которые поддаются перечислению, все они позволяют добавлять и удалять элементы. Если есть ссылка на объект, тип которого реализует интерфейс `IList<T>`, можно написать код, способный добавлять, удалять и искать элементы, не зная конкретный тип коллекции. Это очень мощный механизм.

Наконец, нужно сказать, что на самом деле можно определить интерфейс и создать базовый класс, который реализует интерфейс. Например, в FCL определен интерфейс `IComparer<T>`, и любой тип может реализовать этот интерфейс. Кроме того, FCL предоставляет абстрактный базовый класс `Comparer<T>`, который реализует этот интерфейс (абстрактно) и предлагает некоторые дополнительные методы. Применение обеих возможностей дает большую гибкость, поскольку разработчики теперь могут выбрать из двух вариантов наиболее предпочтительный.

ЧАСТЬ III

Основные типы данных

Глава 14. Символы, строки и обработка текста	344
Глава 15. Перечислимые типы и битовые флаги	393
Глава 16. Массивы	406
Глава 17. Делегаты	426
Глава 18. Настраиваемые атрибуты	458
Глава 19. Null-совместимые значимые типы	481

Глава 14. Символы, строки и обработка текста

Эта глава посвящена приемам обработки отдельных символов, а также целых строк в Microsoft .NET Framework. Вначале рассматриваются структура `System.Char` и способы работы с символами. Потом мы перейдем к весьма полезному классу `System.String`, предназначенному для работы с неизменяемыми строками (такую строку можно создать, но не изменить). Затем рассказывается о динамическом создании строк с помощью класса `System.Text.StringBuilder`. Выходя за рамки основной темы главы, мы обсудим вопросы форматирования объектов в строки и эффективного сохранения и передачи строк различными способами. В конце главы рассказывается о классе `System.Security.SecureString`, который может использоваться для защиты конфиденциальных строк данных, таких как пароли и информация кредитной карты.

Символы

Символы в .NET Framework всегда представлены 16-разрядными кодами стандарта Unicode, что облегчает разработку многоязыковых приложений. Символ представляет собой экземпляр структуры `System.Char` (значимый тип). Тип `System.Char` довольно прост, у него лишь два открытых неизменяемых поля: константа `MinValue`, определенная как `\0`, и константа `MaxValue`, определенная как `\uffff`.

Для экземпляра `Char` можно вызывать статический метод `GetUnicodeCategory`, который возвращает значение перечислимого типа `System.Globalization.UnicodeCategory`, показывающее категорию символа: управляющий символ, символ валюты, буква в нижнем или верхнем регистре, знак пунктуации, математический символ и т. д. (в соответствии со стандартом Unicode).

Для облегчения работы с типом `Char` имеется несколько статических методов, например: `IsDigit`, `IsLetter`, `IsWhiteSpace`, `IsUpper`, `IsLower`, `IsPunctuation`, `IsLetterOrDigit`, `IsControl`, `IsNumber`, `IsSeparator`, `IsSurrogate`, `IsLowSurrogate`, `IsHighSurrogate` и `IsSymbol`. Большинство этих методов обращается к `GetUnicodeCategory` и возвращает `true` или `false`. Обратите внимание: в качестве параметров они принимают либо одиночный символ, либо тип `String` с указанием индекса символа в строке.

Кроме того, статические методы `ToLowerInvariant` и `ToUpperInvariant` позволяют преобразовать символ в его эквивалент в нижнем или верхнем регистре без учета региональных стандартов. Для преобразования символа с учетом

региональных стандартов (culture), относящихся к вызывающему потоку (эти сведения методы получают, запрашивая статическое свойство `CurrentCulture` типа `System.Threading.Thread`), служат методы `ToLower` и `ToUpper`. Чтобы задать определенный набор региональных стандартов, передайте этим методам экземпляр класса `CultureInfo`. Данные о региональных стандартах нужны методам `ToLower` и `ToUpper`, поскольку от них зависит результат операции изменения регистра буквы. Например, в турецком языке символ `U+0069` (латинская строчная буква `i`) при переводе в верхний регистр становится символом `U+0130` (латинская прописная буква `I` с надстрочной точкой), тогда как в других языках — это символ `U+0049` (латинская прописная буква `I`).

Помимо перечисленных статических методов, у типа `Char` есть также несколько собственных экземплярных методов. Метод `Equals` возвращает `true`, если два экземпляра `Char` представляют один и тот же 16-разрядный Unicode-символ. Метод `CompareTo` (определенный в интерфейсах `IComparable` и `IComparable<Char>`) сравнивает два кодовых значения без учета региональных стандартов. Метод `ToString` возвращает строку, состоящую из одного символа, тогда как `Parse` и `TryParse` получают односимвольную строку `String` и возвращают соответствующую кодовую позицию UTF-16.

Наконец, метод `GetNumericValue` возвращает числовой эквивалент символа. Это можно продемонстрировать на следующем примере:

using System;

```
public static class Program {
    public static void Main() {
        Double d;
        d = Char.GetNumericValue('\u0033'); // '\u0033' - это "цифра 3"
                                           // Параметр '3'
                                           // даст тот же результат
        Console.WriteLine(d.ToString());    // Выводится "3"

        // '\u00bc' - это "простая дробь одна четвертая ('1/4')"
        d = Char.GetNumericValue('\u00bc');
        Console.WriteLine(d.ToString());    // Выводится "0.25"

        // 'A' - это "Латинская прописная буква A"
        d = Char.GetNumericValue('A');
        Console.WriteLine(d.ToString());    // Выводится "-1"
    }
}
```

А теперь представлю в порядке предпочтения три способа преобразования различных числовых типов в экземпляры типа `Char`, и наоборот.

- ❑ **Приведение типа.** Самый эффективный способ, так как компилятор генерирует IL-команды преобразования без вызовов каких-либо методов. Для преобразования типа `Char` в числовое значение, такое как `Int32`, приведение

подходит лучше всего. Кроме того, в некоторых языках (например, в C#) допускается указывать, какой код выполняет преобразование: проверяемый или непроверяемый (см. главу 5).

- ❑ **Использование типа `Convert`.** У типа `System.Convert` есть несколько статических методов, корректно преобразующих `Char` в числовой тип и обратно. Все эти методы выполняют преобразование как проверяемую операцию, чтобы в случае потери данных при преобразовании вбрасывалось исключение `OverflowException`.
- ❑ **Использование интерфейса `IConvertible`.** В типе `Char` и во всех числовых типах библиотеки .NET Framework Class Library (FCL) реализован интерфейс `IConvertible`, в котором определены такие методы, как `ToUInt16` и `ToChar`. Этот способ наименее эффективен, так как вызов интерфейсных методов для числовых типов приводит к упаковке экземпляра: `Char` и все числовые типы являются значимыми типами. Методы `IConvertible` вбрасывают исключение `System.InvalidCastException`, если преобразование невозможно (например, преобразование типа `Char` в `Boolean`) или грозит потерей данных. Во многих типах (в том числе `Char` и числовых типах FCL) методы `IConvertible` являются EIM-методами (см. главу 13), а значит, перед вызовом какого-либо метода этого интерфейса нужно выполнить явное приведение экземпляра к `IConvertible`. Все методы `IConvertible` за исключением `GetTypeCode` принимают ссылку на объект, реализующий интерфейс `IFormatProvider`. Этот параметр полезен, когда по какой-либо причине при преобразовании требуется учитывать региональные стандарты. В большинстве операций преобразования в этом параметре передается `null`, потому что он все равно игнорируется.

Применение всех трех способов продемонстрировано в следующем примере:

```
using System;
```

```
public static class Program {  
    public static void Main() {  
        Char c;  
        Int32 n;  
  
        // Преобразование "число - символ" посредством приведения типов C#  
        c = (Char) 65;  
        Console.WriteLine(c); // Выводится "A"  
  
        n = (Int32) c;  
        Console.WriteLine(n); // Выводится "65"  
        c = unchecked((Char) (65536 + 65));  
        Console.WriteLine(c); // Выводится "A"  
  
        // Преобразование "число - символ" с помощью типа Convert  
        c = Convert.ToChar(65);  
        Console.WriteLine(c); // Выводится "A"
```

```
n = Convert.ToInt32(c);
Console.WriteLine(n); // Выводится "65"

// Этот код демонстрирует проверку допустимых значений для Convert
try {
    c = Convert.ToChar(70000); // Слишком много для 16 разрядов
    Console.WriteLine(c);      // Этот вызов выполняться НЕ будет
}
catch (OverflowException) {
    Console.WriteLine("Can't convert 70000 to a Char.");
}

// Преобразование "число - символ" с помощью интерфейса IConvertible
c = ((IConvertible) 65).ToChar(null);
Console.WriteLine(c); // Выводится "A"
n = ((IConvertible) c).ToInt32(null);
Console.WriteLine(n); // Выводится "65"
}
}
```

Тип System.String

В любом приложении мы, несомненно, встретимся с типом `System.String`, представляющим собой неизменяемый упорядоченный набор символов. Будучи прямым потомком `Object`, он является ссылочным типом, и по этой причине строки всегда размещаются в куче и никогда — в стеке потока. В типе `String` реализовано также несколько интерфейсов (`IComparable/IComparable<String>`, `ICloneable`, `IConvertible`, `IEnumerable/IEnumerable<Char>` и `IEquatable<String>`).

Создание строк

Во многих языках (включая C#) `String` считают элементарным типом, то есть компилятор разрешает вставлять литеральные строки непосредственно в исходный код. Компилятор помещает эти литеральные строки в метаданные модуля, где они часто загружаются и на них ссылаются во время выполнения.

В C# с помощью оператора `new` нельзя создавать объекты типа `String` из литеральных строк:

```
using System;

public static class Program {
    public static void Main() {
        String s = new String("Hi there."); // Ошибка
        Console.WriteLine(s);
    }
}
```


Вместо этого используется более простой синтаксис:

using System;

```
public static class Program {
    public static void Main() {
        String s = "Hi there.";
        Console.WriteLine(s);
    }
}
```

Результат компиляции этого кода можно посмотреть с помощью утилиты **ILDasm.exe**:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 13 (0xd)
    .maxstack 1
    .locals init (string V_0)
    IL_0000: ldstr "Hi there."
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: call void [mscorlib]System.Console::WriteLine(string)
    IL_000c: ret
} // end of method Program::Main
```

За создание нового экземпляра объекта отвечает IL-команда **newobj**. Однако здесь этой команды нет. Вместо нее вы видите специальную IL-команду **ldstr** (загрузка строки), которая создает объект **String** на основе литеральной строки, полученной из метаданных. Отсюда следует, что для создания объектов **String** в CLR применяет специальный подход.

Используя небезопасный код, можно создать объект **String** с помощью **Char*** и **SByte***. Тогда следует применить оператор **new** и вызвать один из конструкторов, предоставляемых типом **String** и принимающим параметры **Char*** и **SByte***. Эти конструкторы создают объект **String** и заполняют его строкой, состоящей из указанного массива экземпляров **Char** или байтов со знаком. У других конструкторов нет параметров-указателей, их можно вызвать из любого языка, создающего управляемый код.

В **C#** имеется специальный синтаксис для ввода литеральных строк в исходный код. Для вставки специальных символов, таких как конец строки, возврат каретки, забой, в **C#** используются управляющие последовательности, знакомые разработчикам на **C/C++**:

```
// String содержит символы конца строки и перевода каретки
String s = "Hi\r\nthere.";
```

ВНИМАНИЕ

Задавать в коде последовательность символов конца строки и перевода каретки напрямую, как это сделано в представленном примере, не рекомендуется. У типа System.Environment определено неизменяемое свойство NewLine, которое при выполнении приложения в Windows возвращает строку, состоящую из этих символов. Однако свойство NewLine зависит от платформы и возвращает ту строку, которая обеспечивает создание разрыва строк на конкретной платформе. Скажем, при переносе CLI в UNIX свойство NewLine должно возвращать строку, состоящую только из символа «\n». Чтобы приведенный код работал на любой платформе, перепишите его следующим образом:

```
String s = "Hi" + Environment.NewLine + "there.";
```

Несколько строк можно объединить в одну строку с помощью оператора + языка C#:

```
// Объединение трех литеральных строк образует одну литеральную строку
String s = "Hi" + " " + "there.";
```

Поскольку все строки в этом коде литеральные, компилятор выполняет их конкатенацию на этапе компиляции, в результате в метаданных модуля оказывается лишь строка "Hi there.". Конкатенация нелитеральных строк с помощью оператора + происходит на этапе выполнения. Для конкатенации нескольких строк на этапе выполнения оператор + применять нежелательно, так как он создает в куче несколько строковых объектов. Вместо него используйте тип System.Text.StringBuilder (о нем рассказано далее).

И наконец, в C# есть особый вариант объявления строки, в которой все символы между кавычками трактуются как часть строки. Эти специальные объявления — *буквальные строки* (verbatim strings) — обычно используют при задании пути к файлу или каталогу и при работе с регулярными выражениями. Например:

```
// Задание пути к приложению
String file = "C:\\Windows\\System32\\Notepad.exe";
```

```
// Задание пути к приложению с помощью буквальной строки
String file = @"C:\Windows\System32\Notepad.exe";
```

Оба фрагмента кода дают одинаковый результат. Однако символ @ перед строкой во втором случае сообщает компилятору, что перед ним буквальная строка и он должен рассматривать символ обратного слэша (\) как таковой, а не как признак управляющей последовательности, благодаря чему путь выглядит привычнее.

Теперь, познакомившись с формированием строк, рассмотрим операции, выполняемые над объектами типа String.

Неизменяемые строки

Самое важное, что нужно помнить об объекте `String` — то, что он неизменяем; то есть созданную однажды строку нельзя сделать длиннее или короче, в ней нельзя изменить ни одного символа. Неизменность строк дает определенные преимущества. Для начала можно выполнять операции над строками, не изменяя их:

```
if (s.ToUpperInvariant().Substring(10, 21).EndsWith("EXE")) {  
    ...  
}
```

Здесь `ToUpperInvariant` возвращает новую строку; символы в строке `s` не изменяются. `SubString` обрабатывает строку, возвращенную `ToUpperInvariant`, и тоже возвращает новую строку, которая затем передается методу `EndsWith`. В программном коде приложения нет ссылок на две временные строки, созданные `ToUpperInvariant` и `SubString`, поэтому занятая ими память освободится при очередной сборке мусора. Если выполняется много операций со строками, в куче создается много объектов `String` — это заставляет чаще прибегать к помощи сборщика мусора, что отрицательно сказывается на производительности приложения. Если требуется эффективно выполнять много операций со строками, следует использовать класс `StringBuilder`.

Благодаря неизменности строк отпадает проблема синхронизации потоков при работе со строками. Кроме того, в CLR несколько ссылок `String` могут указывать на один, а не на несколько разных строковых объектов, если строки идентичны. А значит, можно сократить количество строк в системе и уменьшить расход памяти — это именно то, что непосредственно относится к *интернированию строк* (*string interning*), о котором речь пойдет дальше.

По соображениям производительности тип `String` тесно интегрирован с CLR. В частности, CLR «знает» о размещении полей в этом типе и обращается к ним напрямую. За повышение производительности и прямой доступ приходится платить небольшую цену: класс `String` является изолированным. Иначе, имея возможность описать собственный тип, производный от `String`, можно было бы добавлять свои поля, противоречащие структуре `String` и нарушающие работу CLR. Кроме того, ваши действия могли бы нарушить представления CLR об объекте `String`, которые вытекают из его неизменности.

Сравнение строк

Сравнение — пожалуй, наиболее часто выполняемая со строками операция. Есть две причины, по которым приходится сравнивать строки. Мы сравниваем две строки для выяснения, равны ли они, и для сортировки (прежде всего, для представления их пользователю программы).

Для проверки, равны ли строки, и для сравнения строк при сортировке я настоятельно рекомендую использовать один из перечисленных далее методов, реализованных в классе String:

```
Boolean Equals(String value, StringComparison comparisonType)
static Boolean Equals(String a, String b,
    StringComparison comparisonType)
static Int32 Compare(String strA, String strB,
    StringComparison comparisonType)
static Int32 Compare(string strA, string strB,
    Boolean ignoreCase, CultureInfo culture)
static Int32 Compare(String strA, Int32 indexA,
    String strB, Int32 indexB, Int32 length, StringComparison comparisonType)
static Int32 Compare(String strA, Int32 indexA, String strB,
    Int32 indexB, Int32 length, Boolean ignoreCase, CultureInfo culture)

Boolean StartsWith(String value, StringComparison comparisonType)
Boolean StartsWith(String value,
    Boolean ignoreCase, CultureInfo culture)

Boolean EndsWith(String value, StringComparison comparisonType)
Boolean EndsWith(String value, Boolean ignoreCase, CultureInfo culture)
```

При сортировке всегда нужно учитывать регистр по той простой причине, что две строки, отличающиеся лишь регистром символов, будут считаться одинаковыми и поэтому при каждой сортировке упорядочиваться в произвольном порядке, что может приводить пользователя в замешательство.

В аргументе `comparisonType` (он есть в большинстве перечисленных методов) передается одно из значений, определенных в перечислимом типе `StringComparison`, который описан следующим образом:

```
public enum StringComparison {
    CurrentCulture = 0,
    CurrentCultureIgnoreCase = 1,
    InvariantCulture = 2,
    InvariantCultureIgnoreCase = 3,
    Ordinal = 4,
    OrdinalIgnoreCase = 5
}
```

Аргумент `CompareOptions` является одним из значений, определенным перечислимым типом `CompareOptions`:

```
[Flags]
public enum CompareOptions {
    None = 0,
    IgnoreCase = 1,
    IgnoreNonSpace = 2,
```

продолжение ➞

```
IgnoreSymbols = 4,  
IgnoreKanaType = 8,  
IgnoreWidth = 0x00000010,  
Ordinal = 0x40000000,  
OrdinalIgnoreCase = 0x10000000,  
StringSort = 0x20000000  
}
```

Методы, работающие с аргументом `CompareOptions`, также поддерживают явную передачу информации о языке и региональных стандартах. Если установлен флаг `Ordinal` или `OrdinalIgnoreCase`, тогда методы `Compare` игнорируют определенный язык и региональные стандарты.

Во многих программах строки применяют для решения внутренних задач, таких как поддержка имен путей и файлов, URL-адресов, параметров и разделов реестра, переменных окружения, отражения, XML-тегов, XML-атрибутов и т. п. Для сравнения строк внутри программы следует всегда использовать флаг `StringComparison.Ordinal` или `StringComparison.OrdinalIgnoreCase`. Это дает самый быстрый инструмент сравнения, так как он игнорирует лингвистические особенности и региональные стандарты.

В то же время, если требуется корректно сравнить строки с точки зрения лингвистических особенностей (обычно перед выводом их на экран для пользователя), следует использовать флаг `StringComparison.CurrentCulture` или `StringComparison.CurrentCultureIgnoreCase`.

ВНИМАНИЕ

Обычно следует избегать использования флагов `StringComparison.InvariantCulture` и `StringComparison.InvariantCultureIgnoreCase`. Хотя эти значения и позволяют выполнить лингвистически корректное сравнение, применение их для сравнения строк в программе занимает больше времени, чем флага `StringComparison.Ordinal` или `StringComparison.OrdinalIgnoreCase`. Кроме того, игнорирование региональных стандартов — совсем неудачный выбор для сортировки строк, которые планируется показывать пользователю.

ВНИМАНИЕ

Если вы хотите изменить регистр символов строки перед выполнением простого сравнения, следует использовать предоставляемый `String` метод `ToUpperInvariant` или `ToLowerInvariant`. При нормализации строк настоятельно рекомендуется использовать метод `ToUpperInvariant`, а не `ToLowerInvariant` из-за того, что в `Microsoft` сравнение строк в верхнем регистре оптимизировано. На самом деле, в `FCL` перед не зависящим от регистра сравнением строки нормализуют путем приведения их к верхнему регистру.

Иногда для лингвистически корректного сравнения строк используют региональные стандарты, отличные от таковых у вызывающего потока. В таком случае можно задействовать перегруженные версии показанных ранее методов `StartsWith`, `EndsWith` и `Compare` — все они принимают аргументы `Boolean` и `CultureInfo`.

ВНИМАНИЕ

В типе `String` определено несколько вариантов перегрузки методов `Equals`, `StartsWith`, `EndsWith` и `Compare` помимо тех, что приведены ранее. Microsoft рекомендует избегать других версий (не представленных в этой книге). Кроме того, других имеющихся в `String` методов сравнения — `CompareTo` (необходимый для интерфейса `IComparable`), `CompareOrdinal` и операторов `==` и `!=` следует также избегать. Причина в том, что вызывающий код не определяет явно, как должно выполняться сравнение строк, а на основании метода нельзя узнать, какой способ сравнения выбран по умолчанию. Например, по умолчанию метод `CompareTo` выполняет сравнение с учетом региональных стандартов, а `Equals` — без учета. Код будет легче читать и поддерживать, если всегда явно описывать, как следует выполнять сравнение строк.

А теперь поговорим о лингвистически корректных сравнениях. Для представления пары «язык-страна» (как описано в спецификации RFC 1766) в .NET Framework используется тип `System.Globalization.CultureInfo`. В частности, `en-US` означает американскую (США) версию английского языка, `en-AU` — австралийскую версию английского языка, а `de-DE` — германскую версию немецкого языка. В CLR у каждого потока есть два свойства, относящиеся к этой паре и ссылающиеся на объект `CultureInfo`.

- ❑ `CurrentUICulture` служит для получения ресурсов, видимых конечному пользователю. Это свойство наиболее полезно для графического интерфейса пользователя или приложений Web Forms, так как указывает на язык, который следует выбрать для вывода элементов пользовательского интерфейса, таких как метки и кнопки. При извлечении ресурсов применяется только «языковая» часть объекта `CultureInfo`, а информация о стране игнорируется. По умолчанию при создании потока это свойство потока задается Win32-функцией `GetUserDefaultUILanguage` на основании объекта `CultureInfo`, который указывает на язык текущей версии Windows. При использовании MUI-версии (Multilingual User Interface) Windows это свойство можно задать с помощью утилиты Regional and Language Options (Язык и региональные стандарты) панели управления.
- ❑ `CurrentCulture` используется во всех случаях, в которых не используется свойство `CurrentUICulture`, в том числе для форматирования чисел и дат, приведения и сравнения строк. При форматировании требуются обе части объекта `CultureInfo` — информация о языке и стране. По умолчанию при созда-

нии потока это свойство потока задается Win32-функцией `GetUserDefaultLCID` на основании объекта `CultureInfo`. Его можно задать на вкладке **Regional Options** (Региональные параметры) утилиты **Regional and Language Options** (Язык и региональные стандарты) панели управления.

Во многих приложениях свойства `CurrentUICulture` и `CurrentCulture` потока извлекают из одного объекта `CultureInfo`, то есть в них содержится одинаковая информация о языке и стране. Однако она может различаться. Например, в приложении, работающем в США, все элементы интерфейса могут представляться на испанском языке, а валюта и формат дата — в соответствии с принятыми в США стандартами. Для этого свойство `CurrentUICulture` потока должно приравниваться объекту `CultureInfo`, инициализированному парой `en-US`.

Внутренняя реализация объекта `CultureInfo` ссылается на объект `System.Globalization.CompareInfo`, инкапсулирующий принятые в данных региональных стандартах таблицы сортировки в соответствии с правилами `Unicode`. Эти таблицы являются частью самой инфраструктуры `.NET Framework`, поэтому все версии `.NET Framework` (независимо от ОС) также будут сравнивать и сортировать строки.

Значение региональных стандартов при сортировке строк демонстрирует следующий пример:

```
using System;
using System.Globalization;

public static class Program {
    public static void Main() {
        String s1 = "Strasse";
        String s2 = "Straße";
        Boolean eq;

        // CompareOrdinal возвращает ненулевое значение
        eq = String.Compare(s1, s2, StringComparison.Ordinal) == 0;
        Console.WriteLine("Ordinal comparison: '{0}' {2} '{1}'", s1, s2,
            eq ? "==" : "!=");

        // Сортировка строк для немецкого языка (de) в Германии (DE)
        CultureInfo ci = new CultureInfo("de-DE");

        // Compare возвращает нуль
        eq = String.Compare(s1, s2, true, ci) == 0;
        Console.WriteLine("Cultural comparison: '{0}' {2} '{1}'", s1, s2,
            eq ? "==" : "!=");
    }
}
```

В результате компоновки и выполнения кода получим следующее:

```
Ordinal comparison: 'Strasse' != 'Straße'  
Cultural comparison: 'Strasse' == 'Straße'
```

ПРИМЕЧАНИЕ

Если метод Compare не выполняет простое сравнение, то он производит расширение символов (character expansions), то есть разбивает сложные символы на несколько символов, игнорируя региональные стандарты. В предыдущем случае немецкий символ эсцет Я всегда расширяется до ss. Аналогично лигатурный символ Ж всегда расширяется до AE. Поэтому в приведенном примере вызов Compare будет всегда возвращать 0 независимо от выбранных региональных стандартов.

В некоторых редких случаях требуется более тонкий контроль при сравнении строк для проверки на равенство и для сортировки. Это может потребоваться при сравнении строк с японскими иероглифами. Дополнительный контроль получают через объект CultureInfo свойства CompareInfo. Как отмечалось ранее, объект CultureInfo инкапсулирует таблицы сравнения символов для различных региональных стандартов, причем для каждого регионального стандарта существует только один объект CompareInfo.

При вызове метода Compare класса String используются указанные вызывающим потоком региональные стандарты. Если региональные стандарты не указаны, используются значения свойства CurrentCulture вызывающего потока. Код, реализующий метод Compare, получает ссылку на объект CompareInfo соответствующего регионального стандарта и вызывает метод Compare объекта CompareInfo, передавая соответствующие параметры (например, признак игнорирования регистра символов). Естественно, если требуется дополнительный контроль, вы должны самостоятельно вызывать метод Compare конкретного объекта CompareInfo.

Метод Compare класса CompareInfo принимает в качестве параметра значение перечислимого типа CompareOptions, в котором определены символы IgnoreCase, IgnoreKanaType, IgnoreNonSpace, IgnoreSymbols, IgnoreWidth, None, Ordinal и StringSort. Эти символы представляют собой битовые флаги, которые можно объединять посредством оператора «или» для большего контроля над сравнением строк. Полное описание этих символов см. в документации на .NET Framework.

Следующий пример демонстрирует значение региональных стандартов при сортировке строк и различные варианты сравнения строк:

```
using System;  
using System.Text;  
using System.Windows.Forms;  
using System.Globalization;  
using System.Threading;
```



```

public sealed class Program {
    public static void Main() {
        String output = String.Empty;
        String[] symbol = new String[] { "<", "=", ">" };
        Int32 x;
        CultureInfo ci;

        // Следующий код демонстрирует, насколько отличается результат
        // сравнения строк для различных региональных стандартов
        String s1 = "coté";
        String s2 = "côte";

        // Сортировка строк для французского языка во Франции
        ci = new CultureInfo("fr-FR");
        x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
        output += String.Format("{0} Compare: {1} {3} {2}",
            ci.Name, s1, s2, symbol[x + 1]);
        output += Environment.NewLine;

        // Сортировка строк для японского языка в Японии
        ci = new CultureInfo("ja-JP");
        x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
        output += String.Format("{0} Compare: {1} {3} {2}",
            ci.Name, s1, s2, symbol[x + 1]);
        output += Environment.NewLine;

        // Сортировка строк с учетом региональных стандартов потока
        ci = Thread.CurrentThread.CurrentCulture;
        x = Math.Sign(ci.CompareInfo.Compare(s1, s2));
        output += String.Format("{0} Compare: {1} {3} {2}",
            ci.Name, s1, s2, symbol[x + 1]);
        output += Environment.NewLine + Environment.NewLine;

        // Следующий код демонстрирует использование дополнительных возможностей
        // метода CompareInfo.Compare при работе с двумя строками
        // на японском языке
        // Эти строки представляют слово "shinkansen" (название
        // высокоскоростного поезда) в разных вариантах письма:
        // хирагане и катакане
        s1 = " "; // ("\u3057\u3093\u304b\u3093\u305b\u3093")
        s2 = " "; // ("\u30b7\u30f3\u30ab\u30f3\u30bb\u30f3")

        // Здесь результат сравнения по умолчанию
        ci = new CultureInfo("ja-JP");
        x = Math.Sign(String.Compare(s1, s2, true, ci));
        output += String.Format("Simple {0} Compare: {1} {3} {2}",
            ci.Name, s1, s2, symbol[x + 1]);
        output += Environment.NewLine;
    }
}

```

```
// Здесь результат сравнения, который игнорирует тип каны
CompareInfo compareInfo = CompareInfo.GetCompareInfo("ja-JP");
x = Math.Sign(compareInfo.Compare(s1, s2,
    CompareOptions.IgnoreKanaType));
output += String.Format("Advanced {0} Compare: {1} {3} {2}",
    ci.Name, s1, s2, symbol[x + 1]);
MessageBox.Show(output, "Comparing Strings For Sorting");
```

ПРИМЕЧАНИЕ

Подобные файлы с исходным кодом нельзя сохранить в кодировке ANSI, поскольку иначе японские символы будут потеряны. Для того чтобы сохранить этот файл при помощи Microsoft Visual Studio, откройте диалоговое окно **Save File As**, раскройте список форматов сохранения и выберите вариант **Save With Encoding**. Я выбрал Unicode (UTF-8 with signature) – Codepage 65001. Компилятор C# успешно анализирует программный код, использующий эту кодовую страницу.

После компоновки и выполнения кода получим результат, показанный на рис. 14.1.



Рис. 14.1. Результат сортировки строк

Помимо `Compare`, класс `CompareInfo` предлагает методы `IndexOf`, `IsLastIndexOf`, `IsPrefix` и `IsSuffix`. Благодаря имеющейся у каждого из этих методов перегруженной версии, которой в качестве параметра передается значение перечислимого типа `CompareOptions`, вы получаете дополнительные возможности по сравнению с методами `Compare`, `IndexOf`, `LastIndexOf`, `StartsWith` и `EndsWith` класса `String`. Кроме того, следует иметь в виду, что в FCL есть класс `System.StringComparer`, который также можно использовать для сравнения строк. Он оказывается кстати в тех случаях, когда необходимо многократно выполнять однотипные сравнения множества строк.

Интернирование строк

Как я уже отмечал, сравнение строк требуется во многих приложениях. Но эта операция может ощутимо сказаться на производительности. При *порядковом*

сравнении (ordinal comparison) CLR быстро проверяет, равно ли количество символов в строках. При отрицательном результате строки точно не равны, но если длина одинакова, приходится кропотливо сравнивать их символ за символом. При сравнении с учетом региональных стандартов среде CLR тоже приходится посимвольно сравнить строки, потому что две строки разной длины могут оказаться равными.

К тому же, если в памяти содержится еще несколько экземпляров строки, потребуется дополнительная память, ведь строки неизменяемы. Эффективного использования памяти можно добиться, если держать в ней одну строку, на которую будут указывать соответствующие ссылки.

Если в приложении строки сравниваются часто методом порядкового сравнения с учетом регистра или если в приложении ожидается появление множества одинаковых строковых объектов, то для повышения производительности надо применить поддерживаемый CLR механизм *интернирования строк* (string interning). При инициализации CLR создает внутреннюю хэш-таблицу, в которой ключами являются строки, а значениями — ссылки на строковые объекты в управляемой куче. Вначале таблица пуста (это ясно). В классе String есть два метода, предоставляющие доступ к внутренней хэш-таблице:

```
public static String Intern(String str);  
public static String IsInterned(String str);
```

Первый из них, Intern, ищет String во внутренней хэш-таблице. Если строка обнаруживается, возвращается ссылка на соответствующий объект String. Иначе создается копия строки, она добавляется во внутреннюю хэш-таблицу и возвращается ссылка на копию. Если приложение больше не удерживает ссылку на исходный объект String, сборщик мусора вправе освободить память, занимаемую этой строкой. Обратите внимание, что сборщик мусора не вправе освободить строки, на которые ссылается внутренняя хэш-таблица, поскольку в ней самой есть ссылки на эти String. Объекты String, на которые ссылается внутренняя хэш-таблица, нельзя освободить, пока не выгружены соответствующие домены или не закрыт поток.

Как и Intern, метод IsInterned получает параметр String и ищет его во внутренней хэш-таблице. Если поиск удачен, IsInterned возвращает ссылку на интернированную строку. В противном случае он возвращает null, а саму строку не вставляет в хэш-таблицу.

По умолчанию при загрузке сборки CLR интернирует все литеральные строки, описанные в метаданных сборки. В Microsoft выяснили, что это отрицательно сказывается на производительности из-за необходимости дополнительного поиска в хэш-таблицах, поэтому теперь можно отключить эту «функцию». Если сборка отмечена атрибутом System.Runtime.CompilerServices.CompilationRelaxationsAttribute, определяющим значение флага System.Runtime.CompilerServices.CompilationRelaxations.NoStringInterning, то в соответствии со спецификацией ECMA среда CLR *может* не интернировать все строки, определенные в мета-

данных сборки. Обратите внимание, что в целях повышения производительности работы приложения компилятор C# всегда при компиляции сборки определяет этот атрибут/флаг.

Даже если в сборке определен этот атрибут/флаг, CLR может предпочесть интернировать строки, но на это не стоит рассчитывать. В действительности, никогда не стоит писать код в расчете на интернирование строк, если только вы сами в своем коде явно не вызываете метод Intern типа String. Следующий код демонстрирует интернирование строк:

```
String s1 = "Hello";  
String s2 = "Hello";  
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // Должно быть 'False'  
  
s1 = String.Intern(s1);  
s2 = String.Intern(s2);  
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // 'True'
```

При первом вызове метода ReferenceEquals переменная s1 ссылается на объект-строку "Hello" в куче, а s2 — на другую объект-строку "Hello". Поскольку ссылки разные, выводится значение False. Однако если выполнить этот код в CLR версии 2.0, вы увидите True. Причина в том, что эта версия CLR предпочитает игнорировать атрибут/флаг, созданный компилятором C#, и интернирует литеральную строку "Hello" при загрузке сборки в домен приложений. Это означает, что s1 и s2 ссылаются на одну строку в куче. Однако, как уже отмечалось, никогда не стоит писать код с расчетом на такое поведение, потому что в последующих версиях этот атрибут/флаг может приниматься во внимание, и строка "Hello" интернироваться не будет. В действительности, CLR версии 2.0 учитывает этот атрибут/флаг, но только если код сборки создан с помощью утилиты NGen.exe.

Перед вторым вызовом метода ReferenceEquals строка "Hello" явно интернируется, в результате s1 ссылается на интернированную строку "Hello". Затем при повторном вызове Intern переменной s2 присваивается ссылка на ту же самую строку "Hello", на которую ссылается s1. Теперь при втором вызове ReferenceEquals мы гарантировано получаем результат True независимо от того, была ли сборка скомпилирована с этим атрибутом/флагом.

Теперь на примере посмотрим, как использовать интернирование строки для повышения производительности и снижения нагрузки на память. Показанный далее метод NumTimesWordAppearsEquals принимает два аргумента: строку-слово word и массив строк, в котором каждый элемент массива ссылается на одно слово. Метод определяет, сколько раз указанное слово содержится в списке слов, и возвращает число:

```
private static Int32 NumTimesWordAppearsEquals(String word, String[]  
    wordlist) {  
    Int32 count = 0;
```

продолжение ➤

```
for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {  
    if (word.Equals(wordlist[wordnum], StringComparison.Ordinal))  
        count++;  
}  
return count;  
}
```

Как видите, этот метод вызывает метод `Equals` типа `String`, который сравнивает отдельные символы строк и проверяет, все ли символы совпадают. Это сравнение может выполняться медленно. Кроме того, массив `wordlist` может иметь много элементов, которые ссылаются на многие объекты `String`, содержащие тот же набор символов. Это означает, что в куче может существовать множество идентичных строк, не подлежащих сборке в качестве «мусора».

А теперь посмотрим на версию этого метода, который написан с интернированием строк:

```
private static Int32 NumTimesWordAppearsIntern(String word, String[]  
    wordlist) {  
    // В этом методе предполагается, что все элементы в wordlist  
    // ссылаются на интернированные строки  
    word = String.Intern(word);  
    Int32 count = 0;  
    for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {  
        if (Object.ReferenceEquals(word, wordlist[wordnum]))  
            count++;  
    }  
    return count;  
}
```

Этот метод интернирует слово и предполагает, что `wordlist` содержит ссылки на интернированные строки. Во-первых, в этой версии экономится память, если слово повторяется в списке слов, потому что теперь `wordlist` содержит многочисленные ссылки на единственный объект `String` в куче. Во-вторых, эта версия работает быстрее, потому что для выяснения, есть ли указанное слово в массиве, достаточно сравнить указатели.

Хотя метод `NumTimesWordAppearsIntern` работает быстрее, чем `NumTimesWordAppearsEquals`, общая производительность приложения может оказаться ниже, чем при использовании метода `NumTimesWordAppearsIntern` из-за времени, которое требуется на интернирование всех строк по мере добавления их в массив `wordlist` (соответствующий код не показан). Преимущества метода `NumTimesWordAppearsIntern` — ускорение работы и снижение потребления памяти — будут заметны, если приложению нужно множество раз вызывать метод, передавая один и тот же массив `wordlist`. Этим обсуждением я хотел донести до вас, что интернирование строк件но, но использовать его нужно с осторожностью. Собственно, именно по этой причине компилятор C# указывает, что не следует разрешать интернирование строк.

Создание пулов строк

При обработке исходного кода компилятор должен каждую литеральную строку поместить в метаданные управляемого модуля. Если одна строка встречается в исходном коде много раз, размещение всех таких строк в метаданных приведет к увеличению размера результирующего файла.

Чтобы не допустить роста объема кода, многие компиляторы (в том числе C#) хранят литеральную строку в метаданных модуля только в одном экземпляре. Все упоминания этой строки в исходном коде компилятор заменяет ссылками на ее экземпляры в метаданных. Благодаря этому заметно уменьшается размер модуля. Способ не нов — в компиляторах Microsoft C/C++ этот механизм реализован уже давно и называется *созданием пула строк* (string pooling). Это еще одно средство, позволяющее ускорить обработку строк. Полагаю, знание о нем может пригодиться.

Работа с символами и текстовыми элементами в строке

Сравнение строк полезно при сортировке и поиске одинаковых строк, однако иногда требуется видеть отдельные символы в пределах строки. С подобными задачами призваны справляться несколько методов и свойств типа String, в числе которых Length, Chars (индексатор в C#), GetEnumerator, ToCharArray, Contains, IndexOf, LastIndexOf, IndexOfAny и LastIndexOfAny.

На самом деле System.Char представляет одно 16-разрядное кодовое значение в кодировке Unicode, которое необязательно соответствует абстрактному Unicode-символу. Так, некоторые абстрактные Unicode-символы являются комбинацией двух кодовых значений. Например, сочетание символов U+0625 (арабская буква «алеф» с подстрочной «хамза») и U+0650 (арабская «казра») образует один арабский символ, или текстовый элемент.

Кроме того, представление некоторых абстрактных Unicode-символов требует не одного, а двух 16-разрядных кодовых значений. Первое называют *старшим* (high surrogate), а второе — *младшим заменителем* (low surrogate). Значения старшего находятся в диапазоне от U+D800 до U+DBFF, младшего — от U+DC00 до U+DFFF. Такой способ кодировки позволяет представить в Unicode более миллиона различных символов.

Символы-заменители востребованы в основном в странах Восточной Азии и гораздо меньше в США и Европе. Для корректной работы с абстрактными Unicode-символами предназначен тип System.Globalization.StringInfo. Самый простой способ воспользоваться этим типом — создать его экземпляр, передав его конструктору строку. Чтобы затем узнать, сколько текстовых элементов есть в строке, достаточно считать свойство LengthInTextElements объекта StringInfo. Позже можно вызвать метод SubstringByTextElements объекта StringInfo, чтобы извлечь один или несколько последовательных текстовых элементов.

Кроме того, в классе `StringInfo` есть статический метод `GetTextElementEnumerator`, возвращающий объект `System.Globalization.TextElementEnumerator`, который, в свою очередь, позволяет просмотреть в строке все абстрактные Unicode-символы. Наконец, можно воспользоваться статическим методом `ParseCombiningCharacters` типа `StringInfo`, чтобы получить массив значений типа `Int32`, по длине которого можно судить о количестве текстовых элементов в строке. Каждый элемент массива содержит индекс первого кодового значения соответствующего текстового элемента.

Очередной пример демонстрирует различные способы использования класса `StringInfo` для управления текстовыми элементами строки:

```
using System;
using System.Text;
using System.Globalization;
using System.Windows.Forms;

public sealed class Program {
    public static void Main() {
        // Следующая строка содержит комбинированные символы
        String s = "a\u0304\u0308bc\u0327";
        SubstringByTextElements(s);
        EnumTextElements(s);
        EnumTextElementIndexes(s);
    }

    private static void SubstringByTextElements(String s) {
        String output = String.Empty;
        StringInfo si = new StringInfo(s);
        for (Int32 element = 0; element < si.LengthInTextElements; element++) {
            output += String.Format(
                "Text element {0} is '{1}'{2}",
                element, si.SubstringByTextElements(element, 1),
                Environment.NewLine);
        }
        MessageBox.Show(output, "Result of SubstringByTextElements");
    }

    private static void EnumTextElements(String s) {
        String output = String.Empty;
        TextElementEnumerator charEnum =
            StringInfo.GetTextElementEnumerator(s);
        while (charEnum.MoveNext()) {
            output += String.Format(
                "Character at index {0} is '{1}'{2}",
                charEnum.ElementIndex, charEnum.GetTextElement(),
                Environment.NewLine);
        }
    }
}
```

```
        MessageBox.Show(output, "Result of GetTextElementEnumerator");  
    }  
  
    private static void EnumTextElementIndexes(String s) {  
        String output = String.Empty;  
        Int32[] textElemIndex = StringInfo.ParseCombiningCharacters(s);  
        for (Int32 i = 0; i < textElemIndex.Length; i++) {  
            output += String.Format(  
                "Character {0} starts at index {1}{2}".  
                i, textElemIndex[i], Environment.NewLine);  
        }  
        MessageBox.Show(output, "Result of ParseCombiningCharacters");  
    }  
}
```

После компоновки и последующего запуска этого кода на экране появятся информационные окна (рис. 14.2–14.4).

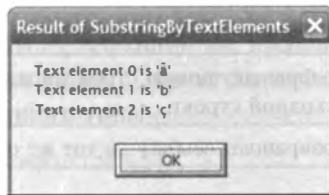


Рис. 14.2. Результат работы метода SubstringByTextElements

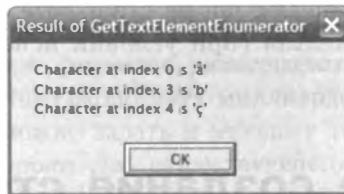


Рис. 14.3. Результат работы метода GetTextElementEnumerator

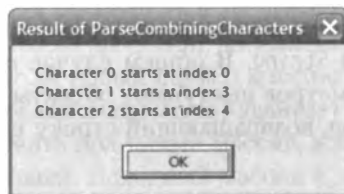


Рис. 14.4. Результат работы метода ParseCombiningCharacters

Прочие операции со строками

В табл. 14.1 представлены методы типа `String`, предназначенные для полного или частичного копирования строк.

Таблица 14.1. Методы копирования строк

Член	Тип метода	Описание
<code>Clone</code>	Экземплярный	Возвращает ссылку на тот же самый объект (<code>this</code>). Это правильно, так как объекты <code>String</code> неизменяемы. Этот метод реализует интерфейс <code>ICloneable</code> класса <code>StringCopy</code>
<code>Copy</code>	Статический	Возвращает новую строку — дубликат заданной строки. Используется редко и нужен только для приложений, рассматривающих строки как лексемы. Обычно строки с одинаковым набором символов интернируются в одну строку. Этот метод, напротив, создает новый строковый объект и возвращает иной указатель (ссылку), хотя в строках содержатся одинаковые символы
<code>CopyTo</code>	Экземплярный	Копирует группу символов строки в массив символов
<code>Substring</code>	Экземплярный	Возвращает новую строку, представляющую часть исходной строки
<code>ToString</code>	Экземплярный	Возвращает ссылку на тот же объект (<code>this</code>)

Помимо этих методов, у типа `String` есть много статических и экземплярных методов для манипуляций строкой, таких как `Insert`, `Remove`, `PadLeft`, `Replace`, `Split`, `Join`, `ToLower`, `ToUpper`, `Trim`, `Concat`, `Format` и пр. Еще раз повторю, что все эти методы возвращают новые строковые объекты; создать строку можно, но изменить ее нельзя (при условии использования безопасного кода).

Эффективное создание строк

Тип `String` представляет собой неизменяемую строку, а для динамических операций со строками и символами при создании объектов `String` в FCL имеется тип `System.Text.StringBuilder`. Его можно рассматривать как некий общедоступный конструктор для `String`. В общем случае нужно создавать методы, у которых в качестве параметров выступают объекты `String`, а не `StringBuilder`, хотя можно написать метод, возвращающий строку, создаваемую динамически внутри метода.

У объекта `StringBuilder` предусмотрено поле со ссылкой на массив структур `Char`. Используя члены `StringBuilder`, можно эффективно манипулировать этим массивом, сокращая строку и изменяя символы строки. При увеличении строки, представляющей ранее выделенный массив символов, `StringBuilder` автоматически выделит память для нового, большего по размеру массива, скопирует

символы и приступит к работе с новым массивом. А прежний массив попадет в сферу действия сборщика мусора.

Сформировав свою строку с помощью объекта `StringBuilder`, «преобразуйте» массив символов `StringBuilder` в объект `String`, вызвав метод `ToString` типа `StringBuilder`. Этот метод просто возвращает ссылку на поле-строку, управляемую объектом `StringBuilder`. Поскольку массив символов здесь не копируется, метод выполняется очень быстро. Объект `String`, возвращаемый методом `ToString`, не может быть изменен. Поэтому, если вы вызовете метод, который попытается изменить строковое поле, управляемое объектом `StringBuilder`, методы этого объекта, зная, что для него был вызван метод `ToString`, создадут новый массив символов, манипуляции с которым не повлияют на строку, возвращенную предыдущим вызовом `ToString`.

Создание объекта `StringBuilder`

В отличие от класса `String` класс `StringBuilder` в CLR не представляет собой ничего особенного. Кроме того, большинство языков (включая `C#`) не считают `StringBuilder` элементарным типом. Объект `StringBuilder` создается так же, как любой объект неэлементарного типа:

```
StringBuilder sb = new StringBuilder();
```

У типа `StringBuilder` несколько конструкторов. Каждый из них обязан выделять память и инициализировать три внутренних поля, управляемых любым объектом `StringBuilder`.

- ❑ **Максимальная емкость** (`maximum capacity`) — поле типа `Int32`, которое задает максимальное число символов, размещаемых в строке. По умолчанию оно равно `Int32.MaxValue` (около двух миллиардов). Это значение обычно не изменяется, хотя можно задать и меньшее значение, ограничивающее размер создаваемой строки. Для существующего объекта `StringBuilder` это поле изменить нельзя.
- ❑ **Емкость** (`capacity`) — поле типа `Int32`, показывающее размер массива символов `StringBuilder`. По умолчанию оно равно 16. Если известно, сколько символов предполагается разместить в `StringBuilder`, укажите это число при создании объекта `StringBuilder`. При добавлении символов `StringBuilder` определяет, не выходит ли новый размер массива за установленный предел. Если да, то `StringBuilder` автоматически удваивает емкость, и исходя из этого значения, выделяет память под новый массив, а затем копирует символы из исходного массива в новый. Исходный массив в дальнейшем утилизируется сборщиком мусора. Динамическое увеличение массива снижает производительность, поэтому его следует избегать, задавая подходящую емкость в начале работы с объектом.
- ❑ **Массив символов** (`character array`) — массив структур `Char`, содержащий набор символов «строки». Число символов всегда меньше (или равно) емкости

и максимальной емкости. Количество символов в строке можно получить через свойство `Length` типа `StringBuilder`. Значение `Length` всегда меньше или равно емкости `StringBuilder`. При создании `StringBuilder` можно инициализировать массив символов, передавая ему `String` как параметр. Если строка не задана, массив первоначально не содержит символов и свойство `Length` возвращает `0`.

Члены типа `StringBuilder`

Тип `StringBuilder` в отличие от `String` представляет изменяемую строку. Это значит, что многие члены `StringBuilder` изменяют содержимое в массиве символов, не создавая новых объектов, размещаемых в управляемой куче. `StringBuilder` выделяет память для новых объектов только:

- при динамическом увеличении размера строки больше установленной емкости;
- при попытке изменить массив после вызова метода `ToString` типа `StringBuilder`.

В табл. 14.2 представлены методы класса `StringBuilder`.

Таблица 14.2. Члены класса `StringBuilder`

Член	Тип члена	Описание
<code>MaxCapacity</code>	Неизменяемое свойство	Возвращает наибольшее количество символов, которое может быть размещено в строке
<code>Capacity</code>	Изменяемое свойство	Получает/устанавливает размер массива символов. При попытке установить емкость меньшую, чем длина строки, или больше, чем <code>MaxCapacity</code> , генерируется исключение <code>ArgumentOutOfRangeException</code>
<code>EnsureCapacity</code>	Метод	Гарантирует, что размер массива символов будет не меньше, чем значение параметра, передаваемого этому методу. Если значение превышает текущую емкость объекта <code>StringBuilder</code> , размер массива увеличивается. Если текущая емкость больше, чем значение, передаваемое этому свойству, размер массива не изменяется
<code>Length</code>	Изменяемое свойство	Возвращает количество символов в «строке». Эта величина может быть меньше текущей емкости массива символов. Присвоение этому свойству значения <code>0</code> сбрасывает содержимое и очищает строку <code>StringBuilder</code>
<code>ToString</code>	Метод	Версия без параметров возвращает объект <code>String</code> , представляющий поле с массивом символов объекта <code>StringBuilder</code>

Член	Тип члена	Описание
Chars	Изменяемое свойство	Возвращает из массива или устанавливает в массиве символ с заданным индексом. В C# это свойство-индексатор (параметризованное свойство), доступ к которому осуществляется как к элементам массива (с использованием квадратных скобок [])
Clear	Метод	Очищает содержимое объекта StringBuilder, аналогично назначению свойству Length значения 0
Append	Метод	Добавляет единичный объект в массив символов, увеличивая его при необходимости. Объект преобразуется в строку с использованием общего формата и с учетом региональных стандартов, связанных с вызывающим потоком
Insert	Метод	Вставляет единичный объект в массив символов, увеличивая его при необходимости. Объект преобразуется в строку с использованием общего формата и с учетом региональных стандартов, связанных с вызывающим потоком
Append-Format	Метод	Добавляет заданные объекты в массив символов, увеличивая его при необходимости. Объекты преобразуются в строку указанного формата и с учетом заданных региональных стандартов. Это один из наиболее часто используемых методов при работе с объектами StringBuilder
Replace	Метод	Заменяет один символ или строку символов в массиве символов
Remove	Метод	Удаляет диапазон символов из массива символов
Equals	Метод	Возвращает true, только если объекты StringBuilder имеют одну и ту же максимальную емкость, емкость и одинаковые символы в массиве
CopyTo	Метод	Копирует подмножество символов StringBuilder в массив Char

Отмечу одно важное обстоятельство: большинство методов StringBuilder возвращают ссылку на тот же объект StringBuilder. Это позволяет выстроить в цепочку сразу несколько операций:

```
StringBuilder sb = new StringBuilder();
String s = sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").
    Replace(' ', '-').Remove(4, 3).ToString();
Console.WriteLine(s); // "Jeff-Richter"
```

У класса StringBuilder нет некоторых аналогов для методов класса String. Например, у класса String есть методы ToLower, ToUpper, EndsWith, PadLeft, Trim и т. д., отсутствующие у класса StringBuilder. В то же время у класса

`StringBuilder` есть удобный метод `Replace`, выполняющий замену символов и строк лишь в части строки (а не во всей строке). Из-за отсутствия полной аналогии методов иногда приходится прибегать к преобразованиям между `String` и `StringBuilder`. Например, сформировать строку, сделать все буквы прописными, а затем вставить в нее другую строку позволяет следующий код:

```
// Создаем StringBuilder для операций со строками
StringBuilder sb = new StringBuilder();

// Выполняем ряд действий со строками, используя StringBuilder
sb.AppendFormat("{0} {1}" "Jeffrey", "Richter").Replace(" ", "-");

// Преобразуем StringBuilder в String,
// чтобы сделать все символы прописными
String s = sb.ToString().ToUpper();

// Очищаем StringBuilder (выделяется память под новый массив Char)
sb.Length = 0;

// Загружаем строку с прописными String в StringBuilder
// и выполняем остальные операции
sb.Append(s).Insert(8, "Marc-");

// Преобразуем StringBuilder обратно в String
s = sb.ToString();

// Выводим String на экран для пользователя
Console.WriteLine(s); // "JEFFREY-Marc-RICHTER"
```

Я был вынужден написать такой код только потому, что `StringBuilder` не выполняет все операции, которые может выполнить `String`. Надеюсь, в будущем Microsoft улучшит класс `StringBuilder`, дополнив его необходимыми методами для работы со строками.

Получение строкового представления объекта

Часто нужно получить строковое представление объекта, например, для отображения числового типа (такого как `Byte`, `Int32`, `Single` и т. д.) и объекта `DateTime`. Поскольку .NET Framework является объектно-ориентированной платформой, то каждый тип должен сам предоставить код, преобразующий «значение» экземпляра в некий строковый эквивалент. Выбирая способы решения этой задачи, разработчики FCL придумали эталон программирования, предназначенный для повсеместного использования. Рассмотрим этот эталон.

Для получения представления любого объекта в виде строки надо вызвать метод `ToString`. Поскольку этот открытый метод без параметров определен в классе `System.Object`, его можно вызывать для экземпляра любого типа. Семантически `ToString` возвращает строку, которая представляет текущее значение объекта в формате, учитывающем текущие региональные стандарты вызвавшего потока. Строковое представление числа, к примеру, должно правильно отображать разделитель дробной части, разделитель групп разрядов и тому подобные параметры, устанавливаемые региональными стандартами вызывающего потока.

Реализация `ToString` в типе `System.Object` просто возвращает полное имя типа объекта. В этом значении мало пользы, хотя для многих типов такое решение по умолчанию может оказаться единственно разумным. Например, как иначе представить в виде строки такие объекты, как `FileStream` или `Hashtable`?

Типы, которые хотят представить текущее значение объекта в более подходящем виде, должны переопределить метод `ToString`. Все базовые типы, встроенные в FCL (`Byte`, `Int32`, `UInt64`, `Double` и т. д.), имеют переопределенный метод `ToString`, реализация которого возвращает строку с учетом региональных стандартов. В отладчике Visual Studio, когда указатель мыши наводится на соответствующую переменную, появляется всплывающая подсказка. Текст этой подсказки формируется путем вызова метода `ToString` этого объекта. Таким образом, при определении класса вы должны всегда переопределять метод `ToString`, чтобы иметь качественную поддержку при отладке программного кода.

Форматы и региональные стандарты

Метод `ToString` без параметров является источником двух проблем. Во-первых, вызывающая программа не управляет форматированием строки, как, например, в случае, когда приложению нужно представить число в денежном или десятичном формате, в процентном или шестнадцатеричном виде. Во-вторых, вызывающая программа не может выбрать формат, учитывающий конкретные региональные стандарты.

Вторая проблема более остро стоит для серверных приложений и менее актуальна для кода на стороне клиента. Изредка приложению требуется форматировать строку с учетом региональных стандартов, отличных от таковых у вызывающего потока. Для управления форматированием строки нужна версия метода `ToString`, позволяющая задавать специальное форматирование и сведения о региональных стандартах.

Тип может предложить вызывающей программе выбор форматирования и региональных стандартов, если он реализует интерфейс `System.IFormattable`:

```
public interface IFormattable {  
    String ToString(String format, IFormatProvider formatProvider);  
}
```

В FCL у всех базовых типов (Byte, SByte, Int16/UInt16, Int32/UInt32, Int64/UInt64, Single, Double, Decimal и DateTime) есть реализации этого интерфейса. Кроме того, есть такие реализации и у некоторых других типов, например GUID. К тому же каждый перечислимый тип автоматически реализует интерфейс IFormattable, позволяющий получить строковое выражение для числового значения, содержащегося в экземпляре перечислимого типа.

У метода ToString интерфейса IFormattable два параметра. Первый, format, — это строка, сообщающая методу способ форматирования объекта. Второй, formatProvider, — это экземпляр типа, который реализует интерфейс System.IFormatProvider. Этот тип предоставляет методу ToString информацию о региональных стандартах. Как — скоро узнаете.

Тип, реализующий метод ToString интерфейса IFormattable, определяет допустимые варианты форматирования. Если переданная строка форматирования неприемлема, тип должен генерировать исключение System.FormatException.

Многие типы FCL предлагают несколько строк форматирования. Например, тип DateTime поддерживает такие строки: "d" — даты в кратком формате, "D" — даты в полном формате, "g" — даты в общем формате, "M" — формат «месяц/день», "s" — сортируемые даты, "T" — время, "u" — универсальное время в стандарте ISO 8601, "U" — универсальное время в полном формате, "Y" — формат «год/месяц» и т. д. Все перечислимые типы поддерживают строки: "G" — общий формат, "F" — формат флагов, "D" — десятичный формат и "X" — шестнадцатеричный формат. Подробнее о форматировании перечислимых типов см. главу 15.

Кроме того, все встроенные числовые типы поддерживают следующие строки: "C" — формат валют, "D" — десятичный формат, "E" — научный (экспоненциальный) формат, "F" — формат чисел с фиксированной точкой, "G" — общий формат, "N" — формат чисел, "P" — формат процентов, "R" — обратимый (round-trip) формат и "X" — шестнадцатеричный формат. Числовые типы поддерживают также шаблоны форматирования для случаев, когда обычных строк форматирования недостаточно. Шаблоны форматирования содержат специальные символы, позволяющие методу ToString данного типа отобразить нужное количество цифр, место разделителя дробной части, количество знаков в дробной части и т. д. Полную информацию о строках форматирования см. в разделе .NET Framework SDK, посвященном форматированию строк.

Если в качестве строки форматирования выступает null, это равносильно вызову метода ToString с параметром "G". Иначе говоря, объекты форматируют себя сами, применяя по умолчанию «общий формат». Разрабатывая реализацию типа, выберите формат, который, по вашему мнению, будет использоваться чаще всего; это и будет «общий формат». Кстати, вызов метода ToString без параметров означает представление объекта в общем формате.

Закончив со строками форматирования, перейдем к региональным стандартам. По умолчанию форматирование выполняется с учетом региональных стандартов, связанных с вызывающим потоком. Это свойственно методу ToString

без параметров и методу `ToString` интерфейса `IFormattable` со значением `null` в качестве `formatProvider`.

Региональные стандарты влияют на форматирование чисел (включая денежные суммы, целые числа, числа с плавающей точкой и проценты), дат и времени. Метод `ToString` для типа, представляющего GUID, возвращает строку, отображающую только значение GUID. Региональные стандарты вряд ли нужно учитывать при создании такой строки, так как она используется только самой программой.

При форматировании числа метод `ToString` «анализирует» параметр `formatProvider`. Если это `null`, метод `ToString` определяет региональные стандарты, связанные с вызывающим потоком, считывая свойство `System.Threading.Thread.CurrentThread.CurrentCulture`. Оно возвращает экземпляр типа `System.Globalization.CultureInfo`.

Получив объект, `ToString` считывает его свойства `NumberFormat` для форматирования числа или `DateTimeFormat` для форматирования даты. Эти свойства возвращают экземпляры `System.Globalization.NumberFormatInfo` и `System.Globalization.DateTimeFormatInfo` соответственно. Тип `NumberFormatInfo` описывает группу свойств, таких как `CurrencyDecimalSeparator`, `CurrencySymbol`, `NegativeSign`, `NumberGroupSeparator` и `PercentSymbol`. Аналогично, у типа `DateTimeFormatInfo` описаны такие свойства, как `Calendar`, `DateSeparator`, `DayNames`, `LongDatePattern`, `ShortTimePattern` и `TimeSeparator`. Метод `ToString` считывает эти свойства при создании и форматировании строки.

При вызове метода `ToString` интерфейса `IFormattable` вместо `null` можно передать ссылку на объект, тип которого реализует интерфейс `IFormatProvider`:

```
public interface IFormatProvider {  
    Object GetFormat(Type formatType);  
}
```

Основная идея применения интерфейса `IFormatProvider` такова: реализация этого интерфейса означает, что экземпляр типа «знает», как обеспечить учет региональных стандартов при форматировании, а региональные стандарты, связанные с вызывающим потоком, игнорируются.

Тип `System.Globalization.CultureInfo` — один из немногих определенных в FCL типов, в которых реализован интерфейс `IFormatProvider`. Если нужно форматировать строку, скажем, для Вьетнама, следует создать объект `CultureInfo` и передать его `ToString` как параметр `formatProvider`. Вот как формируют строковое представление числа `Decimal` в формате вьетнамской валюты:

```
Decimal price = 123.54M;  
String s = price.ToString("C", new CultureInfo("vi-VN"));  
MessageBox.Show(s);
```

Если собрать и запустить этот код, появится информационное окно (рис. 14.5).



Рис. 14.5. Числовое значение в надлежащем формате, представляющем вьетнамскую валюту

Метод `ToString` типа `Decimal`, исходя из того, что аргумент `formatProvider` отличен от `null`, вызывает метод `GetFormat` объекта:

```
NumberFormatInfo nfi = (NumberFormatInfo)
    formatProvider.GetFormat(typeof(NumberFormatInfo));
```

Так `ToString` запрашивает у объекта (`CultureInfo`) данные о надлежащем форматировании чисел. Числовым типам (вроде `Decimal`) достаточно получить лишь сведения о форматировании чисел. Однако другие типы (вроде `DateTime`) могут вызывать `GetFormat` иначе:

```
DateTimeFormatInfo dtfi = (DateTimeFormatInfo)
    formatProvider.GetFormat(typeof(DateTimeFormatInfo));
```

Действительно, раз параметр `GetFormat` может идентифицировать любой тип, значит, метод достаточно гибок, чтобы получить сведения о форматировании любого типа. Во второй версии .NET Framework с помощью `GetFormat` типы могут запрашивать информацию только о числах и датах (и времени); в будущем этот круг будет расширен.

Кстати, чтобы получить строку для объекта, который не отформатирован в соответствии с определенными региональными стандартами, вызовите статическое свойство `InvariantCulture` класса `System.Globalization.CultureInfo` и передайте возвращенный объект как параметр `formatProvider` методу `ToString`:

```
Decimal price = 123.54M;
String s = price.ToString("C", CultureInfo.InvariantCulture);
MessageBox.Show(s);
```

После компоновки и запуска этого кода появится информационное окно (рис. 14.6). Обратите внимание на первый символ в выходной строке: đ. Он представляет международное обозначение денежного знака (U+00A4).

Обычно нет необходимости выводить строку в формате инвариантных региональных стандартов. В типовом случае нужно просто сохранить строку в файле, отложив ее разбор на будущее.

В FCL интерфейс `IFormatProvider` реализован только для трех типов: для уже упоминавшегося типа `CultureInfo`, а также для типов `NumberFormatInfo`

и `DateTimeFormatInfo`. Когда `GetFormat` вызывается для объекта `NumberFormatInfo`, метод проверяет, является ли запрашиваемый тип `NumberFormatInfo`. Если да, возвращается `this`, нет — `null`. Аналогично, вызов `GetFormat` для объекта `DateTimeFormatInfo` возвращает `this`, если запрашиваемый тип `DateTimeFormatInfo`, и `null` — если нет. Реализация этого интерфейса для этих двух типов упрощает программирование.



Рис. 14.6. Числовое значение в формате, представляющем абстрактную денежную единицу

Чаще всего при получении строкового представления объекта вызывающая программа задает только формат, довольствуясь региональными стандартами, связанными с вызывающим потоком. Поэтому обычно мы вызываем `ToString`, передавая строку форматирования и `null` как параметр `formatProvider`. Для упрощения работы с `ToString` во многие типы включены перегруженные версии метода `ToString`. Например, у типа `Decimal` есть четыре перегруженных метода `ToString`:

```
// Эта версия вызывает ToString(null, null)
// Смысл: общий формат, региональные стандарты потока
public override String ToString();
```

```
// В этой версии выполняется полная реализация ToString
// Здесь реализован метод ToString интерфейса IFormattable
// Смысл: и формат, и региональные стандарты задаются вызывающей программой
public String ToString(String format, IFormatProvider formatProvider);
```

```
// Эта версия просто вызывает ToString(format, null)
// Смысл: формат, заданный вызывающей программой,
// и региональные стандарты потока
public String ToString(String format);
```

```
// Эта версия просто вызывает ToString(null, formatProvider)
// Здесь реализуется метод ToString интерфейса IConvertible
// Смысл: общий формат и региональные стандарты,
// заданные вызывающей программой
public String ToString(IFormatProvider formatProvider);
```

Форматирование нескольких объектов в одну строку

До сих пор речь шла о форматировании своих объектов специальным типом. Однако иногда нужно формировать строки из множества форматированных объектов. В следующем примере в строку включаются дата, имя человека и его возраст:

```
String s = String.Format("On {0:D}. {1} is {2:E} years old.",  
    new DateTime(2010, 4, 22, 14, 35, 5), "Aidan", 7);  
Console.WriteLine(s);
```

Если собрать и запустить этот код в потоке с региональным стандартом en-US, на выходе получится строка:

```
On Thursday, April 22, 2010, Aidan is 7.000000E+000 years old
```

Статический метод `Format` типа `String` получает строку форматирования, в которой подставляемые параметры обозначены своими номерами в фигурных скобках. В этом примере строка форматирования указывает методу `Format` подставить вместо `{0}` первый после строки форматирования параметр (текущие дату и время), вместо `{1}` — следующий параметр (`Aidan`) и вместо `{2}` — третий, последний параметр (`7`).

Внутри метода `Format` для каждого объекта вызывается метод `ToString`, получающий его строковое представление. Все возвращенные строки затем объединяются, а полученный результат возвращается методом. Все было бы замечательно, однако нужно иметь в виду, что ко всем объектам применяется общий формат и региональные стандарты вызывающего потока.

Чтобы расширить стандартное форматирование объекта, нужно добавить внутрь фигурных скобок строку форматирования. В частности, следующий код отличается от предыдущего только наличием строк форматирования для подставляемых параметров `0` и `2`:

```
String s = String.Format("On {0:D}. {1} is {2:E} years old.",  
    new DateTime(2010, 4, 22, 14, 35, 5), "Aidan", 7);  
Console.WriteLine(s);
```

Если собрать и запустить этот код в потоке с региональным стандартом en-US, на выходе вы увидите строку:

```
On Thursday, April 22, 2010, Aidan is 7.000000E+000 years old.
```

Разбирая строку форматирования, метод `Format` «видит», что для подставляемого параметра `0` нужно вызывать описанный в его интерфейсе `IFormattable` метод `ToString`, которому передаются в качестве параметров `0` и `null`. Аналогично, `Format` вызывает метод `ToString` для интерфейса `IFormattable` параметра `2`, передавая ему `E` и `null`. Если у типа нет реализации интерфейса `IFormattable`, то `Format` вызывает его метод `ToString` без параметров, а в результирующую строку добавляется формат по умолчанию.

У класса `String` есть несколько перегруженных версий статического метода `Format`. В одну из них передается объект, реализующий интерфейс `IFormatProvider`, в этом случае при форматировании всех подставляемых параметров можно применять региональные стандарты, задаваемые вызывающей программой. Очевидно, `Format` вызывает метод `ToString` для каждого объекта, передавая ему полученный объект `IFormatProvider`.

Если вместо `String` для формирования строки применяется `StringBuilder`, можно вызывать метод `AppendFormat` класса `StringBuilder`. Этот метод работает так же, как `Format` класса `String`, за исключением того, что результат форматирования добавляется к массиву символов `StringBuilder`. Точно так же в `AppendFormat` передается строка форматирования и имеется версия, которой передается `IFormatProvider`.

У типа `System.Console` тоже есть методы `Write` и `Writeline`, которым передаются строка форматирования и замещаемые параметры. Однако у `Console` нет перегруженных методов `Write` и `Writeline`, позволяющих передавать `IFormatProvider`. Если при форматировании строки нужно применить определенные региональные стандарты, вызовите метод `Format` класса `String`, передав ему нужный объект `IFormatProvider`, а затем подставьте результирующую строку в метод `Write` или `Writeline` класса `Console`. Это не намного усложнит задачу, поскольку, как я уже отмечал, код на стороне клиента редко при форматировании применяет региональные стандарты, отличные от тех, что связаны с вызывающим потоком.

Создание собственного средства форматирования

Уже на этом этапе понятно, что платформа .NET Framework предлагает весьма гибкие средства форматирования. Но это не все. Вы можете написать метод, который будет вызываться в `AppendFormat` типа `StringBuilder` независимо от того, для какого объекта выполняется форматирование. Иначе говоря, для каждого объекта вместо метода `ToString` метод `AppendFormat` вызовет вашу функцию, которая будет форматировать один или несколько объектов так, как вам нужно. Следующая информация применима также к методу `Format` типа `String`.

Попробую пояснить описанный механизм на примере. Допустим, вам нужен форматированный HTML-текст, который пользователь будет просматривать в браузере, причем все значения `Int32` должны выводиться полужирным шрифтом. Для этого всякий раз, когда значение типа `Int32` форматируется в `String`, нужно обрамлять строку тегами полужирного шрифта: `` и ``. Вот как это сделать:

```
using System;  
using System.Text;  
using System.Threading;
```

продолжение ➤

```

public static class Program {
    public static void Main() {
        StringBuilder sb = new StringBuilder();
        sb.AppendFormat(new BoldInt32s(), "{0} {1} {2:M}", "Jeff", 123,
            DateTime.Now);
        Console.WriteLine(sb);
    }
}

internal sealed class BoldInt32s : IFormatProvider, ICustomFormatter {
    public Object GetFormat(Type formatType) {
        if (formatType == typeof(ICustomFormatter)) return this;
        return Thread.CurrentThread.CurrentCulture.GetFormat(formatType);
    }

    public String Format(String format, Object arg, IFormatProvider
        formatProvider) {
        String s;

        IFormattable formattable = arg as IFormattable;

        if (formattable == null) s = arg.ToString();
        else s = formattable.ToString(format, formatProvider);

        if (arg.GetType() == typeof(Int32))
            return "<B>" + s + "</B>";
        return s;
    }
}

```

После компиляции и запуска кода в потоке с региональным стандартом en-US появится строка (дата может отличаться):

```
Jeff <B>123</B> January 23
```

Метод Main формирует пустой объект `StringBuilder`, к которому затем добавляется форматированная строка. При вызове `AppendFormat` в качестве первого параметра подставляется экземпляр класса `BoldInt32s`. В нем, помимо рассмотренного ранее интерфейса `IFormatProvider`, реализован также интерфейс `ICustomFormatter`:

```

public interface ICustomFormatter {
    String Format(String format, Object arg,
        IFormatProvider formatProvider);
}

```

Метод `Format` этого интерфейса вызывается всякий раз, когда методу `AppendFormat` класса `StringBuilder` нужно получить строку для объекта. Внутри этого метода у нас появляется возможность гибкого управления процессом

форматирования строки. Заглянем внутрь метода `AppendFormat`, чтобы узнать поподробнее, как он работает. Следующий псевдокод демонстрирует работу метода `AppendFormat`:

```
public StringBuilder AppendFormat(IFormatProvider formatProvider,
    String format, params Object[] args) {

    // Если параметр IFormatProvider передан, выясним,
    // предлагает ли он объект ICustomFormatter
    ICustomFormatter cf = null;

    if (formatProvider != null)
        cf = (ICustomFormatter)
            formatProvider.GetFormat(typeof(ICustomFormatter));

    // Продолжаем добавлять литеральные символы (не показанные
    // в этом псевдокоде) и замещаемые параметры в массив символов
    // объекта StringBuilder.
    Boolean MoreReplaceableArgumentsToAppend = true;
    while (MoreReplaceableArgumentsToAppend) {
        // argFormat ссылается на замещаемую строку форматирования,
        // полученную из параметра format
        String argFormat = /* ... */;

        // argObj ссылается на соответствующий элемент
        // параметра-массива args
        Object argObj = /* ... */;

        // argStr будет указывать на отформатированную строку,
        // которая добавляется к результирующей строке
        String argStr = null;

        // Если есть специальное средство форматирования,
        // используем его для форматирования аргумента
        if (cf != null)
            argStr = cf.Format(argFormat, argObj, formatProvider);

        // Если специального средства форматирования нет или оно не выполняло
        // форматирование аргумента, попробуем еще что-нибудь
        if (argStr == null) {
            // Выясняем, поддерживает ли тип аргумента
            // дополнительное форматирование
            IFormattable formattable = argObj as IFormattable;
            if (formattable != null) {
                // Да: передаем методу интерфейса для этого типа
                // строку форматирования и класс-поставщик
```

продолжение ➤

```

        argStr = formattable.ToString(argFormat, formatProvider);
    } else {
        // Нет; используем общий формат с учетом
        // региональных стандартов потока
        if (argObj != null) argStr = argObj.ToString();
        else argStr = String.Empty;
    }
}
// Добавляем символы из argStr в массив символов (поле - член класса)
/* ... */

// Проверяем, есть ли еще параметры, нуждающиеся в форматировании
MoreReplaceableArgumentsToAppend = /* ... */;
}
return this;
}

```

Когда Main обращается к методу AppendFormat, тот вызывает метод GetFormat моего поставщика формата, передавая ему тип ICustomFormatter. Метод GetFormat, описанный в моем типе BoldInt32s, «видит», что запрашивается ICustomFormatter, и возвращает ссылку на собственный объект, потому что он реализует этот интерфейс.

Если из GetFormat запрашивается какой-то другой тип, я вызываю метод GetFormat для объекта CultureInfo, связанного с вызывающим потоком.

При необходимости форматировать замещаемый параметр AppendFormat вызывается метод Format класса ICustomFormatter. В моем примере вызывается метод Format, описанный моим типом BoldInt32s. В своем методе Format я проверяю, поддерживает ли формируемый объект расширенное форматирование посредством интерфейса IFormattable. Если нет, то для форматирования объекта я вызываю метод ToString без параметров (унаследованный от Object); если да — вызываю расширенный метод ToString, передавая ему строку форматирования и поставщика формата.

Теперь, имея форматированную строку, я проверяю, имеет ли объект тип Int32, и если да, обрамляю строку HTML-тегами и , после чего возвращаю полученную строку. Если тип объекта отличается от Int32, просто возвращаю форматированную строку без дополнительной обработки.

Получение объекта посредством разбора строки

В предыдущем разделе я рассказал о получении представления определенного объекта в виде строки. Здесь мы пойдем в обратном направлении: рассмотрим, как получить представление конкретной строки в виде объекта. Получать

объект из строки требуется не часто, однако иногда это может оказаться полезным. В компании Microsoft осознали важность формализации механизма, посредством которого строки можно разобрать на объекты.

Любой тип, способный провести синтаксический разбор строки, имеет открытый, статический метод `Parse`. Он получает `String`, а на выходе возвращает экземпляр данного типа; в каком-то смысле `Parse` ведет себя как фабрика. В FCL метод `Parse` существует для всех числовых типов, а также для типов `DateTime`, `TimeSpan` и некоторых других (подобных типам данных SQL).

Посмотрим, как из строки получить целочисленный тип. Все числовые типы (`Byte`, `SByte`, `Int16/UInt16`, `Int32/UInt32`, `Int64/UInt64`, `Single`, `Double` и `Decimal`) имеют минимум один метод `Parse`. Вот как выглядит метод `Parse` для типа `Int32` (для других числовых типов методы `Parse` выглядят аналогично).

```
public static Int32 Parse(String s, NumberStyles style, IFormatProvider
    provider);
```

Взглянув на прототип, вы сразу поймете суть работы этого метода. Параметр `s` типа `String` идентифицирует строковое представление числа, из которого путем синтаксического разбора нужно получить объект `Int32`. Параметр `style` типа `System.Globalization.NumberStyles` — это набор двоичных флагов для идентификации символов, которые метод `Parse` должен найти в строке. А параметр `provider` типа `IFormatProvider` идентифицирует объект, используя который метод `Parse` может получить информацию о региональных стандартах, о чем речь шла ранее.

Далее при обращении к `Parse` вбрасывается исключение `System.FormatException`, так как в начале разбираемой строки находится пробел:

```
Int32 x = Int32.Parse(" 123", NumberStyles.None, null);
```

Чтобы «пропустить» пробел, надо вызвать `Parse` с другим параметром `style`:

```
Int32 x = Int32.Parse(" 123", NumberStyles.AllowLeadingWhite, null);
```

Подробнее о двоичных символах и стандартных комбинациях, определенных в типе `NumberStyles`, см. документацию на .NET Framework SDK.

Вот пример синтаксического разбора строки шестнадцатеричного числа:

```
Int32 x = Int32.Parse("1A", NumberStyles.HexNumber, null);
Console.WriteLine(x); // Отображает "26".
```

Этому методу `Parse` передаются три параметра. Для удобства у многих типов есть перегруженные версии `Parse` с меньшим числом параметров. Например, у типа `Int32` четыре перегруженные версии метода `Parse`:

```
// Передает NumberStyles.Integer в качестве параметра стиля
// и информации о региональных стандартах потока
public static Int32 Parse(String s);
```

продолжение ⇨


```
// Передает информацию о региональных стандартах потока
public static Int32 Parse(String s, NumberStyles style);

// Передает NumberStyles.Integer в качестве параметра стиля
public static Int32 Parse(String s, IFormatProvider provider)

// Это тот метод, о котором я уже рассказывал в этом разделе
public static int Parse(String s, NumberStyles style,
    IFormatProvider provider);
```

У типа `DateTime` также есть метод `Parse`:

```
public static DateTime Parse(String s,
    IFormatProvider provider, DateTimeStyles styles);
```

Этот метод действует подобно методу `Parse` для числовых типов за исключением того, что методу `Parse` типа `DateTime` передается набор двоичных флагов, описанных в перечислимом типе `System.Globalization.DateTimeStyles`, а не в типе `NumberStyles`. Подробнее о двоичных символах и стандартных комбинациях, определенных в типе `DateTimeStyles`, см. документацию на `.NET Framework SDK`.

Для удобства у типа `DateTime` есть три перегруженных метода `Parse`:

```
// Передается информация о региональных стандартах потока,
// а также DateTimeStyles.None в качестве стиля
public static DateTime Parse(String s);

// DateTimeStyles.None передается в качестве стиля
public static DateTime Parse(String s, IFormatProvider provider);

// Этот метод рассмотрен мной в этом разделе
public static DateTime Parse(String s,
    IFormatProvider provider, DateTimeStyles styles);
```

Даты и время плохо поддаются синтаксическому разбору. Многие разработчики столкнулись с тем, что метод `Parse` типа `DateTime` способен получить дату и время из строки, в которой нет ни того, ни другого. Поэтому в тип `DateTime` введен метод `ParseExact`, который анализирует строку согласно некоему шаблону, показывающему, как должна выглядеть строка, содержащая дату или время, и как выполнять ее разбор. О шаблонах форматирования см. раздел, посвященный `DateTimeFormatInfo`, в документации на `.NET Framework SDK`.

ПРИМЕЧАНИЕ

Некоторые разработчики сообщили в Microsoft о следующем факте: при частом вызове `Parse` этот метод вбрасывает исключения (из-за неверных данных, вводимых пользователями), что отрицательно сказывается на производительности приложения. Для таких требующих высокой производительности случаев в Microsoft создали методы `TryParse` для всех числовых

типов данных, для `DateTime`, `TimeSpan` и даже для `IPAddress`. Вот как выглядит один из двух перегруженных методов `TryParse` типа `Int32`:

```
public static Boolean TryParse(String s, NumberStyles style,  
    IFormatProvider provider, out Int32 result);
```

Как видите, метод возвращает `true` или `false`, информируя, удалось ли разобрать строку, преобразовав ее в `Int32`. Если метод возвращает `true`, переменная, переданная по ссылке в результирующем параметре, будет содержать полученное в результате разбора числовое значение. Шаблон `TryXxx` обсуждается в главе 20.

Кодировки: преобразования между символами и байтами

Win32-программистам часто приходится писать код, преобразующий символы и строки из `Unicode` в `Multi-Byte Character Set (MBCS)`. Поскольку я тоже этим занимался, могу авторитетно утверждать, что дело это очень нудное и чреватое ошибками. В CLR все символы представлены 16-разрядными `Unicode`-значениями, а строки состоят только из 16-разрядных `Unicode`-символов. Это намного упрощает работу с символами и строками в период выполнения.

Однако порой текст требуется записать в файл или передать его по сети. Когда текст состоит главным образом из символов английского языка, запись и передача 16-разрядных значений становится неэффективной, поскольку половина значений — нули. Поэтому разумнее сначала *закодировать* (*encode*) 16-разрядные символы в более компактный массив байтов, чтобы потом *декодировать* (*decode*) его в массив 16-разрядных значений.

Кодирование текста помогает также управляемому приложению работать со строками, созданными в системах, не поддерживающих `Unicode`. Так, чтобы создать текстовый файл, предназначенный для японской версии Windows 95, нужно сохранить текст в `Unicode`, используя код `Shift-JIS` (кодировка страницы 932). Аналогично с помощью кода `Shift-JIS` можно прочитать в CLR текстовый файл, созданный в японской версии Windows 95.

Кодирование обычно выполняется, когда надо отправить строку в файл или сетевой поток с помощью типов `System.IO.BinaryWriter` и `System.IO.StreamWriter`. Декодирование обычно выполняется при чтении из файла или сетевого потока с помощью типов `System.IO.BinaryReader` и `System.IO.StreamReader`. Если кодировка явно не указана, все эти типы по умолчанию используют код `UTF-8` (`UTF` означает `Unicode Transformation Format`). Возможно, и вам придется выполнить кодирование или декодирование строки.

К счастью, в FCL есть типы, позволяющие упростить операции кодирования и декодирования. К наиболее часто используемым кодировкам относят `UTF-16` и `UTF-8`.

- ❑ UTF-16 кодирует каждый 16-разрядный символ в 2 байта. При этом символы остаются, как были, и сжатия данных не происходит — скорость процесса отличная. Часто код UTF-16 называют еще Unicode-кодировкой (Unicode encoding). Заметьте также, что, используя UTF-16, можно выполнить преобразование из прямого порядка байтов (big endian) в обратный (little endian), и наоборот.
- ❑ UTF-8 кодирует некоторые символы одним байтом, другие — двумя байтами, третьи — тремя, а некоторые — четырьмя. Символы со значениями ниже 0x0080, которые в основном используются в англоязычных странах, сжимаются в один байт.

Символы между 0x0080 и 0x07FF, хорошо подходящие для европейских и среднеазиатских языков, преобразуются в 2 байта. Символы, начиная с 0x0800 и выше, предназначенные для языков Восточной Азии, преобразуются в 3 байта. И наконец, пары символов-заместителей (surrogate character pairs) записываются в 4 байта. UTF-8 — весьма популярная система кодирования, однако она уступает UTF-16, если нужно кодировать много символов со значениями от 0x0800 и выше.

Хотя для большинства случаев подходят кодировки UTF-16 и UTF-8, FCL поддерживает и менее популярные кодировки.

- ❑ UTF-32 кодирует все символы в 4 байта. Эта кодировка используется для создания простого алгоритма прохода символов, в котором не требуется разбираться с символами, состоящими из переменного числа байтов. В частности, UTF-32 упрощает работу с символами-заместителями, так как каждый символ состоит ровно из 4 байт. Ясно, что UTF-32 неэффективна с точки зрения экономии памяти, поэтому она редко используется для сохранения или передачи строк в файл или по сети, а обычно применяется внутри программ. Стоит также заметить, что UTF-32 можно задействовать для преобразования прямого порядка следования байтов в обратный, и наоборот.
- ❑ UTF-7 обычно используется в старых системах, где под символ отводится 7 разрядов. Этой кодировки следует избегать, поскольку обычно она приводит не к сжатию, а к раздуванию данных. Консорциум Unicode Consortium настоятельно рекомендует отказаться от применения UTF-7.
- ❑ ASCII кодирует 16-разрядные символы в ASCII-символы; то есть любой 16-разрядный символ со значением меньше 0x0080 переводится в одиночный байт. Символы со значением больше 0x007F не поддаются этому преобразованию, и значение символа теряется. Для строк, состоящих из символов в ASCII-диапазоне (от 0x00 до 0x7F), эта кодировка сжимает данные наполовину, причем очень быстро (поскольку старший байт просто отбрасывается). Данный код не годится для символов вне ASCII-диапазона, так как теряются значения символов.

Наконец, FCL позволяет кодировать 16-разрядные символы в произвольную кодовую страницу. Как и в случае с ASCII, это преобразование может привести к потере значений символов, не отображаемых в заданной кодовой странице. Используйте кодировки UTF-16 и UTF-8 во всех случаях, когда не имеете дело со старыми файлами и приложениями, в которых применена какая-либо иная кодировка.

Когда нужно выполнить кодирование или декодирование набора символов, сначала надо получить экземпляр класса, производного от `System.Text.Encoding`. Абстрактный базовый класс `Encoding` имеет несколько статических свойств, каждое из которых возвращает экземпляр класса, производного от `Encoding`.

Вот как можно выполнить кодирование и декодирование символов с использованием кодировки UTF-8:

```
using System;
using System.Text;

public static class Program {
    public static void Main() {
        // Эту строку мы будем кодировать
        String s = "Hi there.";

        // Получаем объект, производный от Encoding, который "умеет" выполнять
        // кодирование и декодирование с использованием UTF-8
        Encoding encodingUTF8 = Encoding.UTF8;

        // Выполняем кодирование строки в массив байтов
        Byte[] encodedBytes = encodingUTF8.GetBytes(s);

        // Показываем значение закодированных байтов
        Console.WriteLine("Encoded bytes: " +
            BitConverter.ToString(encodedBytes));

        // Выполняем декодирование массива байтов обратно в строку
        String decodedString = encodingUTF8.GetString(encodedBytes);

        // Показываем декодированную строку
        Console.WriteLine("Decoded string: " + decodedString);
    }
}
```

Вот результат выполнения этой программы:

```
Encoded bytes: 48-69-20-74-68-65-72-65-2E
Decoded string: Hi there.
```

Помимо UTF8, у класса `Encoding` есть и другие статические свойства: `Unicode`, `BigEndianUnicode`, `UTF32`, `UTF7`, `ASCII` и `Default`. Последнее возвращает объект, который выполняет кодирование и декодирование с учетом кодовой страницы

пользователя, заданной с помощью утилиты Regional and Language Options (Язык и региональные стандарты) панели управления (см. описание Win32-функции GetACP). Однако свойство Default применять не рекомендуется, потому что поведение приложения будет зависеть от настройки машины, то есть при изменении кодовой таблицы, предлагаемой по умолчанию, или выполнении приложения на другой машине приложение поведет себя иначе.

Наряду с перечисленными свойствами, у Encoding есть статический метод GetEncoding, позволяющий указать кодовую страницу (в виде числа или строки). Метод GetEncoding возвращает объект, выполняющий кодирование/декодирование, используя заданную кодовую страницу. Например, можно вызвать GetEncoding с параметром "Shift-JIS" или 932.

Когда делается первый запрос объекта кодирования, свойство класса Encoding (или его метод GetEncoding) создает и возвращает объект для требуемой кодировки. При последующих запросах такого же объекта будет возвращаться уже имеющийся объект; то есть при очередном запросе новый объект не создается. Благодаря этому число объектов в системе не увеличивается и нагрузка на сборщик мусора снижается.

Кроме статических свойств и метода GetEncoding класса Encoding, для создания экземпляра класса кодирования можно задействовать классы System.Text.UnicodeEncoding, System.Text.UTF8Encoding, System.Text.UTF32Encoding, System.Text.UTF7Encoding или System.Text.ASCIIEncoding. Только помните, что в этих случаях в управляемой куче появятся новые объекты, что неминуемо отрицательно скажется на производительности.

У классов UnicodeEncoding, UTF8Encoding, UTF32Encoding и UTF7Encoding есть несколько конструкторов, дающих дополнительные возможности в плане управления процессом кодирования и *маркерами последовательности байтов* (Byte Order Mark, BOM). Первые три класса также имеют конструкторы, которые позволяют заставить класс вбрасывать исключение при декодировании некорректной последовательности байтов; эти конструкторы нужно использовать для обеспечения безопасности приложения и защиты от приема некорректных входных данных.

Возможно, при работе с BinaryWriter или StreamWriter вам понадобится явное создание экземпляров этих классов. У класса ASCIIEncoding лишь один конструктор, и поэтому возможности управления кодированием здесь невелики. Получать объект ASCIIEncoding (точнее, ссылку на него) всегда следует путем запроса свойства ASCII класса Encoding. Никогда не создавайте самостоятельно экземпляр класса ASCIIEncoding — при этом создаются дополнительные объекты в куче, что отрицательно сказывается на производительности.

Вызвав для объекта, производного от Encoding, метод GetBytes, можно преобразовать массив символов в массив байтов. (У этого метода есть несколько перегруженных версий.) Для обратного преобразования вызовите метод GetChars или более удобный GetString. (Эти методы также имеют несколько перегруженных версий.) Работа методов GetBytes и GetString продемонстрирована в при-

веденном ранее примере. Кстати, у всех типов, производных от `Encoding`, есть метод `GetByteCount`, который, не выполняя реального кодирования, подсчитывает количество байтов, необходимых для кодирования данного набора символов. Он может пригодиться, когда нужно выделить память под массив байтов. Имеется также аналогичный метод `GetCharCount`, который возвращает число подлежащих декодированию символов, не выполняя реального декодирования. Эти методы полезны, когда требуется сэкономить память и многократно использовать массив.

Методы `GetByteCount` и `GetCharCount` работают не так быстро, поскольку для получения точного результата они должны анализировать массив символов/байтов. Если скорость важнее точности, вызывайте `GetMaxByteCount` или `GetMaxCharCount` — оба метода принимают целое число, в котором задается число символов или байтов соответственно, и возвращают максимально возможный размер массива.

Каждый объект, производный от `Encoding`, имеет набор открытых неизменяемых свойств, дающих более подробную информацию о кодировании. Подробнее см. описание этих свойств в документации на `.NET Framework SDK`.

Чтобы проиллюстрировать свойства и их назначение, я написал программу, в которой эти свойства вызываются для разных вариантов кодирования:

```
using System;  
using System.Text;
```

```
public static class Program {  
    public static void Main() {  
        foreach (EncodingInfo ei in Encoding.GetEncodings()) {  
            Encoding e = ei.GetEncoding();  
            Console.WriteLine("{1}{0}" +  
                "\tCodePage={2}, WindowsCodePage={3}{0}" +  
                "\tWebName={4}, HeaderName={5}, BodyName={6}{0}" +  
                "\tIsBrowserDisplay={7}, IsBrowserSave={8}{0}" +  
                "\tIsMailNewsDisplay={9}, IsMailNewsSave={10}{0}",  
                Environment.NewLine,  
                e.EncodingName, e.CodePage, e.WindowsCodePage,  
                e.WebName, e.HeaderName, e.BodyName,  
                e.IsBrowserDisplay, e.IsBrowserSave,  
                e.IsMailNewsDisplay, e.IsMailNewsSave);  
        }  
    }  
}
```

Вот результат работы этой программы (для экономии бумаги я сократил текст):

```
IBM EBCDIC (US-Canada)  
CodePage=37, WindowsCodePage=1252  
WebName=IBM037, HeaderName=IBM037, BodyName=IBM037
```

продолжение ➤

```
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False
```

OEM United States

```
CodePage=437, WindowsCodePage=1252
WebName=IBM437, HeaderName=IBM437, BodyName=IBM437
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False
```

IBM EBCDIC (International)

```
CodePage=500, WindowsCodePage=1252
WebName=IBM500, HeaderName=IBM500, BodyName=IBM500
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Arabic (ASMO 708)

```
CodePage=708, WindowsCodePage=1256
WebName=ASMO-708, HeaderName=ASMO-708, BodyName=ASMO-708
IsBrowserDisplay=True, IsBrowserSave=True
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Unicode

```
CodePage=1200, WindowsCodePage=1200
WebName=utf-16, HeaderName=utf-16, BodyName=utf-16
IsBrowserDisplay=False, IsBrowserSave=True
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Unicode (Big-Endian)

```
CodePage=1201, WindowsCodePage=1200
WebName=unicodeFFFE, HeaderName=unicodeFFFE, BodyName=unicodeFFFE
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Western European (DOS)

```
CodePage=850, WindowsCodePage=1252
WebName=ibm850, HeaderName=ibm850, BodyName=ibm850
IsBrowserDisplay=False, IsBrowserSave=False
IsMailNewsDisplay=False, IsMailNewsSave=False
```

Unicode (UTF-8)

```
CodePage=65001, WindowsCodePage=1200
WebName=utf-8, HeaderName=utf-8, BodyName=utf-8
IsBrowserDisplay=True, IsBrowserSave=True
IsMailNewsDisplay=True, IsMailNewsSave=True
```

Обзор наиболее популярных методов, предоставляемых классами, производными от Encoding, завершает табл. 14.3.

Таблица 14.3. Методы классов, производных от Encoding

Метод	Описание
GetPreamble	Возвращает массив байтов, показывающих, что нужно записать в поток перед записью кодированных байтов. Часто такие байты называют ВОМ-байтами (byte order mark) или преамбулой (preamble). Когда вы приступаете к чтению из потока, ВОМ-байты помогают автоматически определить кодировку потока, чтобы правильно выбрать надлежащий декодировщик. В некоторых классах, производных от Encoding, этот метод возвращает массив из 0 байт, что означает отсутствие преамбулы. Объект UTF8Encoding может быть создан явно, так, чтобы этот метод возвращал массив из 3 байт: 0xEF, 0xBB, 0xBF. Объект UnicodeEncoding может быть создан явно, так, чтобы этот метод возвращал массив из двух байт: 0xFE, 0xFF для прямого порядка следования байтов (big endian) или 0xFF, 0xFE — для обратного (little endian)
Convert	Преобразует массив байтов из одной кодировки в другую. Внутри этот статический метод вызывает метод GetChars для объекта в исходной кодировке и передает результат методу GetBytes для объекта в целевой кодировке. Полученный массив байтов возвращается вызывающей программе
Equals	Возвращает true, если два производных от Encoding объекта представляют одну кодовую страницу и одинаковую преамбулу
GetHashCode	Возвращает кодовую страницу объекта кодирования

Кодирование и декодирование потоков символов и байтов

Представьте, что вы читаете закодированную в UTF-16 строку с помощью объекта `System.Net.Sockets.NetworkStream`. Весьма вероятно, что байты из потока поступают группами разного размера, например сначала придут 5 байт, а затем 7. В UTF-16 каждый символ состоит из двух байт. Поэтому в результате вызова метода `GetString` класса `Encoding` с передачей первого массива из 5 байт будет возвращена строка, содержащая только два символа. При следующем вызове `GetString` из потока поступят следующие 7 байт, и `GetString` вернет строку, содержащую три символа, причем все неверные!

Причина искажения данных состоит в том, что ни один из производных от `Encoding` классов не отслеживает состояние потока между двумя вызовами своих методов. Если вы выполняете кодирование или декодирование символов и байтов, поступающих порциями, то вам придется приложить дополнительные усилия для отслеживания состояния между вызовами, чтобы избежать потери данных.

Чтобы выполнить декодирование порции данных, следует получить ссылку на производный от `Encoding` объект (как описано в предыдущем разделе) и вызвать его метод `GetDecoder`. Этот метод возвращает ссылку на вновь созданный

объект, чей тип является производным от класса `System.Text.Decoder`. Класс `Decoder`, подобно классу `Encoding`, является абстрактным базовым классом. Из документации на `.NET Framework SDK` вы не узнаете, что некоторые классы представляют собой конкретные реализации класса `Decoder`, хотя FCL описывает группу производных от `Decoder` классов. Все эти классы являются внутренними для FCL, однако метод `GetDecoder` создает экземпляры этих классов и возвращает их коду вашего приложения.

У всех производных от `Decoder` классов существуют два метода: `GetChars` и `GetCharCount`. Естественно, они служат для декодирования и работают аналогично рассмотренным ранее методам `GetChars` и `GetCharCount` класса `Encoding`. Когда вы вызываете один них, он декодирует массив байтов, насколько это возможно. Если в массиве не хватает байтов для формирования символа, то оставшиеся байты сохраняются внутри объекта декодирования. При следующем вызове одного из этих методов объект декодирования берет оставшиеся байты и складывает их с вновь полученным массивом байтов — благодаря этому декодирование данных, поступающих порциями, выполняется корректно. Объекты `Decoder` весьма удобны для чтения байтов из потока.

Тип, производный от `Encoding`, может служить для кодирования/декодирования без отслеживания состояния. Однако тип, производный от `Decoder`, можно использовать только для декодирования. Чтобы выполнить кодирование строки порциями, вместо метода `GetDecoder` класса `Encoding` применяется метод `GetEncoder`. Он возвращает вновь созданный объект, производный от класса `System.Text.Encoder`, который также является абстрактным базовым классом. И опять, в документации на `.NET Framework SDK` нет описания классов, представляющих собой конкретную реализацию класса `Encoder`, хотя в FCL определена группа производных от `Encoder` классов. Подобно классам, производным от `Decoder`, они являются внутренними для FCL, однако метод `GetEncoder` создает экземпляры этих классов и возвращает их коду приложения.

Все классы, производные от `Encoder`, имеют два метода: `GetBytes` и `GetByteCount`. При каждом вызове объект, производный от `Encoder`, отслеживает оставшуюся необработанной информацию, так что можно кодировать данные порциями.

Кодирование и декодирование строк в кодировке Base-64

В настоящее время популярность кодировок UTF-16 и UTF-8 растет. Помимо прочего, это связано еще и с возможностью кодировать последовательности байтов в строки в кодировке base-64. В FCL есть методы для кодирования и декодирования в кодировке base-64, но не думайте, что это происходит посредством типа, производного от `Encoding`. По определенным причинам кодирование и декодирование в кодировке base-64 выполняется с помощью статических методов, предоставляемых типом `System.Convert`.

Чтобы декодировать строку в кодировке base-64 в массив байтов, вызовите статический метод `FromBase64String` или `FromBase64CharArray` класса `Convert`. Аналогично для декодирования массива байтов в строку base-64 служит статический метод `ToBase64String` или `ToBase64CharArray` класса `Convert`. Следующий код демонстрирует использование этих методов:

```
using System;
```

```
public static class Program {  
    public static void Main() {  
        // Получаем набор из 10 байт, сгенерированных случайным образом  
        Byte[] bytes = new Byte[10];  
        new Random().NextBytes(bytes);  
  
        // Отображаем байты  
        Console.WriteLine(BitConverter.ToString(bytes));  
  
        // Кодируем байты в строку в кодировке base-64 и выводим эту строку  
        String s = Convert.ToBase64String(bytes);  
        Console.WriteLine(s);  
  
        // Декодируем строку в кодировке base-64 обратно в байты и выводим ее  
        bytes = Convert.FromBase64String(s);  
        Console.WriteLine(BitConverter.ToString(bytes));  
    }  
}
```

После компиляции этого кода и запуска выполняемого модуля получим следующие строки (ваш результат может отличаться от моего, поскольку байты получены случайным образом):

```
3B-B9-27-40-59-35-86-54-5F-F1  
07knQFk1h1Rf8Q==  
3B-B9-27-40-59-35-86-54-5F-F1
```

Защищенные строки

Часто объекты `String` применяют для хранения конфиденциальных данных, таких как пароли или информация кредитной карты. К сожалению, объекты `String` хранят массив символов в памяти, и если разрешить выполнение небезопасного или неуправляемого кода, он может просмотреть адресное пространство кода, найти строку с конфиденциальной информацией и использовать ее в своих неблагоприятных целях. Даже если объект `String` существует недолго и становится добычей сборщика мусора, CLR может не сразу задействовать ранее занятую этим объектом память (особенно если речь идет об объектах `String` предыдущих версий), оставляя символы объекта в памяти, где они могут быть

скомпрометированы. Кроме того, поскольку строки являются неизменяемыми, при манипуляции ими старые версии «висят» в памяти, в результате разные версии строки остаются в различных областях памяти.

В некоторых государственных учреждениях действуют строгие требования безопасности, гарантирующие определенный уровень защиты. Для решения таких задач специалисты Microsoft добавили в FCL безопасный строковый класс `System.Security.SecureString`. При создании объекта `SecureString` его код выделяет блок неуправляемой памяти, которая содержит массив символов. Сборщик мусора ничего не знает об этой неуправляемой памяти.

Символы строки шифруются для защиты конфиденциальной информации от любого потенциально опасного или неуправляемого кода. Для дописывания в конец строки, вставки, удаления или замены отдельных символов в защищенной строке служат соответственно методы `AppendChar`, `InsertAt`, `RemoveAt` и `SetAt`. При вызове любого из этих методов код метода расшифровывает символы, выполняет операцию и затем обратно шифрует строку. Это означает, что символы находятся в незашифрованном состоянии в течение очень короткого периода времени. Это также означает, что символы строки модифицируются в том же месте, где хранятся, но скорость операций все равно конечна, так что прибегать к ним желательно пореже.

Класс `SecureString` реализует интерфейс `IDisposable`, служащий для надежного уничтожения конфиденциальной информации, хранимой в строке. Когда приложению больше не нужно хранить конфиденциальную строковую информацию, достаточно просто вызвать метод `Dispose` типа `SecureString`. Код `Dispose` обнуляет содержимое буфера памяти, чтобы предотвратить доступ постороннего кода, и только после этого буфер освобождается. Заметьте: класс `SecureString` наследует от класса `CriticalFinalizerObject` (см. главу 21), который гарантирует вызов метода `Finalize` попавшего в распоряжение сборщика мусора объекта `SecureString`, обнуление строки и последующее освобождение буфера. В отличие от объекта `String`, при уничтожении объекта `SecureString` символы зашифрованной строки в памяти не остаются.

Узнав, как создавать и изменять объект `SecureString`, пора поговорить о его использовании. К сожалению, в последней версии FCL поддержка класса `SecureString` ограничена. Точнее, методов, принимающих параметр `SecureString`, очень немного. В версии 2 инфраструктуры .NET Framework передать `SecureString` в качестве пароля можно:

- ❑ при работе с криптографическим провайдером (`Cryptographic Service Provider, CSP`), см. класс `System.Security.Cryptography.CspParameters`;
- ❑ при создании, импорте или экспорте сертификата в формате X.509, см. классы `System.Security.Cryptography.X509Certificates.X509Certificate` и `System.Security.Cryptography.X509Certificates.X509Certificate2`;
- ❑ при запуске нового процесса под определенной учетной записью пользователя, см. классы `System.Diagnostics.Process` и `System.Diagnostics.ProcessStartInfo`;

- ❑ при организации нового сеанса записи журнала событий, см. класс `System.Diagnostics.Eventing.Reader.EventLogSession`;
- ❑ при использовании элемента управления `System.Windows.Controls.PasswordBox`, см. класс свойства `SecurePassword`.

Наконец, вы вправе создавать собственные методы, принимающие в качестве аргумента объект `SecureString`. В методе надо задействовать объект `SecureString` для создания буфера неуправляемой памяти, хранящего расшифрованные символы, до использования этого буфера в методе. Чтобы сократить до минимума временное «окно» доступа к конфиденциальным данным, ваш код должен обращаться к расшифрованной строке минимально возможное время. После использования строки надо обнулить буфер и освободить его как можно скорее. Также никогда не размещайте содержимое `SecureString` в типе `String` — в этом случае незашифрованная строка находится в куче и не обнуляется, пока память не будет задействована повторно после сборки мусора. Класс `SecureString` не переопределяет метод `ToString` специально — это нужно для предотвращения раскрытия конфиденциальных данных (что может произойти при преобразовании их в `String`).

Вот пример, демонстрирующий, как нужно инициализировать и использовать `SecureString` (при компиляции нужно указать параметр `/unsafe` компилятора C#):

```
using System;
using System.Security;
using System.Runtime.InteropServices;

public static class Program {
    public static void Main() {
        using (SecureString ss = new SecureString()) {
            Console.WriteLine("Please enter password: ");
            while (true) {
                ConsoleKeyInfo cki = Console.ReadKey(true);
                if (cki.Key == ConsoleKey.Enter) break;

                // Присоединить символы пароля в конец SecureString
                ss.AppendChar(cki.KeyChar);
                Console.WriteLine("*");
            }
            Console.WriteLine();

            // Пароль введен, отобразим его для демонстрационных целей
            DisplaySecureString(ss);
        }
        // После 'using' SecureString обрабатывается методом Disposed,
        // поэтому никаких конфиденциальных данных в памяти нет
    }
}
```

```
// Этот метод небезопасен, потому что обращается к неуправляемой памяти
private unsafe static void DisplaySecureString(SecureString ss) {
    Char* pc = null;
    try {
        // Расшифровка SecureString в буфер неуправляемой памяти
        pc = (Char*) Marshal.SecureStringToCoTaskMemUnicode(ss);

        // Доступ к буферу неуправляемой памяти,
        // который хранит расшифрованную версию SecureString
        for (Int32 index = 0; pc[index] != 0; index++)
            Console.Write(pc[index]);
    }
    finally {
        // Обеспечиваем обнуление и освобождение буфера неуправляемой памяти,
        // который хранит расшифрованные символы SecureString
        if (pc != null)
            Marshal.ZeroFreeCoTaskMemUnicode((IntPtr) pc);
    }
}
```

Класс `System.Runtime.InteropServices.Marshal` предоставляет 5 методов, которые служат для расшифровки символов `SecureString` в буфер неуправляемой памяти. Все методы, за исключением аргумента `SecureString`, статические и возвращают `IntPtr`. У каждого метода есть связанный метод, который нужно обязательно вызывать для обнуления и освобождения внутреннего буфера. В табл. 14.4 приведены методы класса `System.Runtime.InteropServices.Marshal`, используемые для расшифровки `SecureString` в буфер неуправляемой памяти, а также связанные методы для обнуления и освобождения буфера.

Таблица 14.4. Методы класса `Marshal` для работы с защищенными строками

Метод расшифровки <code>SecureString</code> в буфер	Метод обнуления и освобождения буфера
<code>SecureStringToBSTR</code>	<code>ZeroFreeBSTR</code>
<code>SecureStringToCoTaskMemAnsi</code>	<code>ZeroFreeCoTaskMemAnsi</code>
<code>SecureStringToCoTaskMemUnicode</code>	<code>ZeroFreeCoTaskMemUnicode</code>
<code>SecureStringToGlobalAllocAnsi</code>	<code>ZeroFreeGlobalAllocAnsi</code>
<code>SecureStringToGlobalAllocUnicode</code>	<code>ZeroFreeGlobalAllocUnicode</code>

Глава 15. Перечислимые типы и битовые флаги

Перечислимые типы и битовые флаги поддерживаются в Windows долгие годы, поэтому я уверен, что многие из вас уже знакомы с их применением. Но настоящему объектно-ориентированным перечислимые типы и битовые флаги становятся в общезыковой исполняющей среде (CLR) и библиотеке классов .NET Framework (FCL). Здесь у них появляются особые качества, которые, полагаю, многим разработчикам пока неизвестны. Меня приятно удивило, насколько благодаря этим новшествам, о которых, собственно, и идет разговор в этой главе, можно облегчить разработку приложений.

Перечислимые типы

Перечислимым (enumerated type) называют тип, в котором описан набор пар, состоящих из символьных имен и значений. Далее приведен тип `Color`, определяющий совокупность идентификаторов, каждый из которых обозначает определенный цвет:

```
internal enum Color {  
    White. // Присваивается значение 0  
    Red.   // Присваивается значение 1  
    Green. // Присваивается значение 2  
    Blue.  // Присваивается значение 3  
    Orange // Присваивается значение 4  
}
```

Программист, конечно же, может вместо `White` написать `0`, вместо `Green` — `1` и т. д. Однако перечислимый тип все-таки лучше жестко заданных в исходном коде числовых значений, и вот почему.

- ❑ Программу, где используются перечислимые типы, проще написать, ее легче анализировать, меньше проблем с ее сопровождением. Символьное имя перечислимого типа проходит через весь код, и занимаясь то одной, то другой частью программы, программист не обязан помнить значение каждого «зашифрованного» в коде значения (что `White` равен `0`, а `0` означает `White`). Если же числовое значение символа почему-либо изменилось, то нужно только перекомпилировать исходный код, не изменяя в нем ни буквы. Кроме того, работая с инструментами документирования и другими утилитами, такими как отладчик, программист видит осмысленные символьные имена, а не цифры.

- ❑ Перечислимые типы подвергаются строгой проверке типов. Например, компилятор сообщит об ошибке, если в качестве значения я попытаюсь передать методу тип `Color.Orange` (оранжевый цвет), когда метод ожидает перечислимый тип `Fruit` (фрукт).

В CLR перечислимые типы — это не просто идентификаторы, с которыми имеет дело компилятор. Перечислимые типы играют важную роль в системе типов, на них возлагается решение очень серьезных задач, просто немислимых для перечислимых типов в других средах (например, в неуправляемом языке C++).

Каждый перечислимый тип напрямую наследует от типа `System.Enum`, производного от `System.ValueType`, а тот, в свою очередь, — от `System.Object`. Из этого следует, что перечислимые типы относятся к значимым типам (см. главу 5) и могут выступать как в неупакованной, так и в упакованной формах. Однако в отличие от других значимых типов, у перечислимого типа не может быть методов, свойств и событий. Впрочем, как вы увидите в конце данной главы, симитировать наличие метода у перечислимого типа можно при помощи механизма *методов расширения* (extension methods).

При компиляции перечислимого типа компилятор C# превращает каждый идентификатор в константное поле типа. Например, предыдущее перечисление `Color` компилятор видит примерно так:

```
internal struct Color : System.Enum {
    // Далее перечислены открытые константы,
    // определяющие символьные имена и значения
    public const Color White = (Color) 0;
    public const Color Red = (Color) 1;
    public const Color Green = (Color) 2;
    public const Color Blue = (Color) 3;
    public const Color Orange = (Color) 4;

    // Далее находится открытое поле экземпляра со значением переменной Color
    // Код с прямой ссылкой на этот экземпляр невозможен
    public Int32 value__;
}
```

Однако компилятор C# не будет обрабатывать такой код, потому что он не разрешает определять типы, производные от специального типа `System.Enum`. Данный пример просто демонстрирует внутреннюю суть происходящего. В общем-то, перечислимый тип — это обычная структура, внутри которой описан набор константных полей и одно экземплярное поле. Константные поля попадают в метаданные сборки, откуда их можно извлечь с помощью механизма отражения. Это означает, что в период выполнения можно получить все идентификаторы и их значения, связанные перечислимым типом, а также преобразовать строковый идентификатор в эквивалентное ему числовое значение. Эти операции предоставлены базовым типом `System.Enum`, который предлагает статические и экземплярные методы, выполняющие специальные операции над

экземплярами перечислимых типов, избавляя вас от необходимости использовать отражение. Мы поговорим о них подробно чуть позже.

ВНИМАНИЕ

Описанные перечислимым типом символы являются константами, и компилятор преобразует ссылку на такой символ в числовое значение. В результате определяющая перечислимый тип сборка может оказаться ненужной в период выполнения. Но она требуется, если в коде присутствует ссылка не на определенные перечислимым типом символы, а на сам тип. То есть возникает проблема версий, связанная с тем, что символы перечислимого типа являются константами, а не значениями, предназначенными только для чтения. Впрочем, эта тема подробно освещена в главе 7.

К примеру, для типа `System.Enum` существует статический метод `GetUnderlyingType`, а для типа `System.Type` — экземплярный метод `GetEnumUnderlyingType`:

```
public static Type GetUnderlyingType(Type enumType): // Определен
                                                    // в типе System.Enum
public Type GetEnumUnderlyingType(): // Определен в типе System.Type
```

Оба этих метода возвращают базовый тип, используемый для хранения значения перечислимого типа. В основе любого перечисления лежит один из основных типов, например `byte`, `sbyte`, `short`, `ushort`, `int` (именно он используется в C# по умолчанию), `uint`, `long` и `ulong`. Все эти элементарные типы C# имеют аналоги в FCL. Однако компилятор C# пропустит только элементарный тип; задание базового класса FCL (например, `Int32`) приведет к сообщению об ошибке (ошибка CS1008: ожидается тип `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`):

```
error CS1008: Type byte, sbyte, short, ushort, int, uint, long, or ulong
expected
```

Вот как должно выглядеть на C# объявление перечисления, в основе которого лежит тип `byte` (`System.Byte`):

```
internal enum Color : byte {
    White,
    Red,
    Green,
    Blue,
    Orange
}
```

Если перечисление `Color` определено подобным образом, метод `GetUnderlyingType` вернет следующий результат:

```
// Эта строка выводит "System.Byte"
Console.WriteLine(Enum.GetUnderlyingType(typeof(Color)));
```


Компилятор C# считает перечислимые типы элементарными. Поэтому для операций с их экземплярами применяются уже знакомые нам операторы (`==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `^`, `&`, `|`, `~`, `++` и `--`). Все они действуют на поле `value__` экземпляра перечисления. А компилятор C# допускает приведение экземпляров одного перечислимого типа к другому. Также поддерживается явное и неявное приведение к числовому типу.

Чтобы узнать значение одной или нескольких строк экземпляра перечисления, используйте метод `System.Enum.ToString`, унаследованный от `System.Enum`:

```
Color c = Color.Blue;
Console.WriteLine(c);           // "Blue" (Общий формат)
Console.WriteLine(c.ToString()); // "Blue" (Общий формат)
Console.WriteLine(c.ToString("G")); // "Blue" (Общий формат)
Console.WriteLine(c.ToString("D")); // "3" (Десятичный формат)
Console.WriteLine(c.ToString("X")); // "03" (Шестнадцатеричный формат)
```

ПРИМЕЧАНИЕ

При работе с шестнадцатеричным форматом метод `ToString` всегда возвращает прописные буквы. Количество возвращенных цифр зависит от типа, лежащего в основе перечисления. Для типов `byte/sbyte` — это две цифры, для типов `short/ushort` — четыре, для типов `int/uint` — восемь, а для типов `long/ulong` — снова две. При необходимости добавляются ведущие нули.

Помимо метода `ToString` тип `System.Enum` предлагает статический метод `Format`, служащий для форматирования значения перечислимого типа:

```
public static String Format(Type enumType, Object value, String format);
```

В общем случае метод `ToString` требует меньшего объема кода и проще в вызове. Зато методу `Format` можно передать числовое значение в качестве параметра `value`, даже если у вас отсутствует экземпляр перечисления. Например, этот код выведет строку `"Blue"`:

```
// В результате выводится строка "Blue"
Console.WriteLine(Enum.Format(typeof(Color), 3, "G"));
```

ПРИМЕЧАНИЕ

Можно объявить перечисление, различные идентификаторы которого имеют одинаковое числовое значение. В процессе преобразования числового значения в символ посредством общего форматирования методы типа вернут один из символов, правда, неизвестно какой. Если соответствия не обнаруживается, возвращается строка с числовым значением.

Статический метод `GetValues` типа `System.Enum` и метод `GetEnumValues` экземпляра `System.Type` создают массив, элементами которого становятся символьные имена перечисления. И каждый элемент будет содержать соответствующее числовое значение:

```
public static Array GetValues(Type enumType); // Определен в System.Enum
public Array GetEnumValues();                // Определен в System.Type
```

Этот метод вместе с методом `ToString` позволяет вывести все идентификаторы и числовые значения перечисления:

```
Color[] colors = (Color[]) Enum.GetValues(typeof(Color));
Console.WriteLine("Number of symbols defined: " + colors.Length);
Console.WriteLine("Value\tSymbol\n-----\t-----");
foreach (Color c in colors) {
    // Выводим каждый идентификатор в десятичном и общем форматах
    Console.WriteLine("{0:5:D}\t{0:G}", c);
}
```

Вот к какому результату приводит выполнение этого кода:

```
Number of symbols defined: 5
```

Value	Symbol
-----	-----
0	White
1	Red
2	Green
3	Blue
4	Orange

Здесь мы рассматриваем наиболее интересные операции, применимые к перечислимым типам. Полагаю, что показывать символьные имена элементов пользовательского интерфейса (раскрывающихся списков, полей со списком и т. п.) чаще всего вы будете с помощью метода `ToString`, используя общий формат, если, правда, выводимые строки не требуют локализации (эту операцию перечислимые типы не поддерживают). Помимо метода `GetValues`, типы `System.Enum` и `System.Type` предоставляют еще два метода возвращения идентификаторов:

```
// Возвращает строковое представление числового значения
public static String GetName(Type enumType, Object value); // Определен
                                                           // в System.Enum
public String GetEnumName(Object value); // Определен в System.Type
```

```
// Возвращает массив строк: по одной на каждое
// символьное имя из перечисления
public static String[] GetNames(Type enumType); // Определен в System.Enum
public String[] GetEnumNames(); // Определен в System.Type
```

Мы рассмотрели несколько методов, позволяющих найти символьное имя, или идентификатор перечислимого типа. Однако нужен еще и метод определения

значения, соответствующего идентификатору, например, вводимому пользователем в текстовое поле. Преобразование идентификатора в экземпляр перечислимого типа легко реализуется статическими методами `Parse` и `TryParse` типа `Enum`:

```
public static Object Parse(Type enumType, String value);
public static Object Parse(Type enumType, String value, Boolean ignoreCase);
public static Boolean TryParse<TEnum>(String value,
    out TEnum result) where TEnum: struct;
public static Boolean TryParse<TEnum>(String value,
    Boolean ignoreCase, out TEnum result)
where TEnum : struct;
```

Вот пример применения данных методов:

```
// Так как Orange определен как 4, 'с' присваивается значение 4
Color c = (Color) Enum.Parse(typeof(Color), "orange", true);
// Так как Brown не определен, вбрасывается исключение ArgumentException
c = (Color) Enum.Parse(typeof(Color), "Brown", false);
// Создается экземпляр перечисления Color со значением 1
Enum.TryParse<Color>("1", false, out c);
// Создается экземпляр перечисления Color со значение 23
Enum.TryParse<Color>("23", false, out c);
```

Наконец, рассмотрим статический метод `IsDefined` типа `Enum` и метод `IsEnumDefined` типа `Type`:

```
public static Boolean IsDefined(Type enumType, Object value); // Определен
                                                             // в System.Enum
public Boolean IsEnumDefined(Object value); // Определен в System.Type
```

С их помощью определяется допустимость числового значения для данного перечисления:

```
// Выводит значение "True", так как в перечислении Color
// идентификатор Red определен как 1
Console.WriteLine(Enum.IsDefined(typeof(Color), 1));
// Выводит значение "True", так как в перечислении Color
// идентификатор White определен как 0
Console.WriteLine(Enum.IsDefined(typeof(Color), "White"));
// Выводит значение "False", так как выполняется проверка с учетом регистра
Console.WriteLine(Enum.IsDefined(typeof(Color), "white"));
// Выводит значение "False", так как в перечислении Color
// отсутствует идентификатор со значением 10
Console.WriteLine(Enum.IsDefined(typeof(Color), 10));
```

Метод `IsDefined` часто используется для проверки параметров. Например:

```
public void SetColor(Color c) {
    if (!Enum.IsDefined(typeof(Color), c)) {
        throw(new ArgumentOutOfRangeException("c", c, "Invalid Color value.));
    }
}
```

```
// Задать цвет, как White, Red, Green, Blue или Orange
...
}
```

Без подобной проверки не обойтись, потому что пользователь вполне может вызвать метод `SetColor` вот таким способом:

```
SetColor((Color) 547);
```

Так как соответствие числу 547 в перечислении отсутствует, метод `SetColor` бросает исключение `ArgumentOutOfRangeException`, в результате мы увидим, какой параметр некорректен и почему.

ВНИМАНИЕ

При всем удобстве метода `IsDefined` применять его следует с осторожностью, во-первых, он всегда выполняет поиск с учетом регистра, во-вторых, работает крайне медленно, так как в нем используются отражения. Самостоятельно написав код проверки возможных значений, вы повысите производительность своего приложения. Кроме того, метод работает только для перечислимых типов, определенных в той сборке, из которой он вызывается. Например, пусть перечисление `Color` определено в одной сборке, а метод `SetColor` — в другой. При вызове методом `SetColor` метода `IsDefined` все будет работать, если цвет имеет значение `White`, `Red`, `Green`, `Blue` или `Orange`. Однако если в будущем мы добавим в перечисление `Color` цвет `Purple`, метод `SetColor` начнет использовать неизвестное ему значение, и результат его работы станет непредсказуемым.

Напоследок упомянем набор статических методов `ToObject` типа `System.Enum`, преобразующих экземпляры типа `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64` или `UInt64` в экземпляры перечислимого типа.

Перечислимые типы всегда применяют в сочетании с другим типом. Обычно их используют в качестве параметров методов или возвращаемых типов, свойств или полей. Часто спрашивают, где определять перечислимый тип: внутри или на уровне того типа, которому он требуется. В FCL вы увидите, что обычно перечислимый тип определяется на уровне класса, которому он требуется. Причина проста: сокращение объема набираемого разработчиком кода. Поэтому, пока не возник конфликт имен, определяйте свой перечислимый тип на одном уровне с основным классом.

Битовые флаги

Программисты часто работают с наборами битовых флагов. Метод `GetAttributes` типа `System.IO.File` возвращает экземпляр типа `FileAttributes`. Тип `FileAttributes` является экземпляром перечислимого типа, основанного на типе `Int32`, где каж-

дый разряд соответствует какому-то атрибуту файла. В FCL тип `FileAttributes` описан следующим образом:

```
[Flags, Serializable]
public enum FileAttributes {
    ReadOnly = 0x0001,
    Hidden = 0x0002,
    System = 0x0004,
    Directory = 0x0010,
    Archive = 0x0020,
    Device = 0x0040,
    Normal = 0x0080,
    Temporary = 0x0100,
    SparseFile = 0x0200,
    ReparsePoint = 0x0400,
    Compressed = 0x0800,
    Offline = 0x1000,
    NotContentIndexed = 0x2000,
    Encrypted = 0x4000
}
```

Выяснить, является ли файл скрытым, позволяет следующий код:

```
String file = Assembly.GetEntryAssembly().Location;
FileAttributes attributes = File.GetAttributes(file);
Console.WriteLine("Is {0} hidden? {1}", file, (
    attributes & FileAttributes.Hidden) != 0);
```

ПРИМЕЧАНИЕ

В классе `Enum` метод `HasFlag` определяется следующим образом:

```
public Boolean HasFlag(Enum flag);
```

С его помощью можно переписать вызов метода `Console.WriteLine`:

```
Console.WriteLine("Is {0} hidden? {1}", file,
    attributes.HasFlag(FileAttributes.Hidden));
```

Однако я не рекомендую использовать метод `HasFlag`. Дело в том, что он принимает параметры типа `Enum`, а значит, передаваемые ему значения должны быть упакованы, что требует места в памяти.

Данный код демонстрирует, как изменить файлу атрибуты «только для чтения» и «скрытый»:

```
File.SetAttributes(file, FileAttributes.ReadOnly | FileAttributes.Hidden);
```

Из описания типа `FileAttributes` видно, что, как правило, при создании набора комбинируемых друг с другом битовых флагов используют перечислимые типы. Однако несмотря на внешнюю схожесть, перечислимые типы семантиче-

ски отличаются от битовых флагов. Если в первом случае мы имеем отдельные числовые значения, то во втором приходится иметь дело с набором флагов, одни из которых установлены, а другие нет.

Определяя перечислимый тип, предназначенный для идентификации битовых флагов, каждому идентификатору следует явно присвоить числовое значение. Обычно в соответствующем идентификатору значении установлен лишь один бит. Также часто приходится видеть идентификатор `None`, значение которого определено как 0. Еще можно определить идентификаторы, представляющие часто используемые комбинации (см. приведенный далее символ `ReadWrite`). Настоятельно рекомендуется применять к перечислимому типу специализированный атрибут типа `System.FlagsAttribute`:

```
[Flags] // Компилятор C# допускает значение "Flags" или "FlagsAttribute"
internal enum Actions {
    None = 0
    Read = 0x0001,
    Write = 0x0002,
    ReadWrite = Actions.Read | Actions.Write,
    Delete = 0x0004,
    Query = 0x0008,
    Sync = 0x0010
}
```

Для работы с перечислимым типом `Actions` можно использовать все описанные в предыдущем разделе методы. Хотя иногда возникает необходимость изменить поведение ряда функций. К примеру, рассмотрим код:

```
Actions actions = Actions.Read | Actions.Delete; // 0x0005
Console.WriteLine(actions.ToString());           // "Read, Delete"
```

Метод `ToString` пытается преобразовать числовое значение в его символьный эквивалент. Но у числового значения `0x0005` нет символьного эквивалента. Однако обнаружив у типа `Actions` атрибут `[Flags]`, метод `ToString` рассматривает числовое значение уже как набор битовых флагов. Так как биты `0x0001` и `0x0002` установлены, метод `ToString` формирует строку `"Read, Delete"`. Если в описании типа `Actions` убрать атрибут `[Flags]`, метод вернет строку `"5"`.

В предыдущем разделе мы рассмотрели метод `ToString` и привели три способа форматирования выходной строки: `"G"` (общий), `"D"` (десятичный) и `"X"` (шестнадцатеричный). Форматируя экземпляр перечислимого типа с использованием общего формата, метод сначала определяет наличие атрибута `[Flags]`. Если атрибут не указан, отыскивается и возвращается идентификатор, соответствующий данному числовому значению. Обнаружив же данный атрибут, метод действует по следующему алгоритму:

1. Получает набор числовых значений, определенных в перечислении, и сортирует их в нисходящем порядке.

2. Для каждого значения выполняется операция конъюнкции (AND) с экземпляром перечисления. В случае равенства результата числовому значению связанная с ним строка добавляется в итоговую строку, соответствующие же биты считаются учтенными и сбрасываются. Операция повторяется до завершения проверки всех числовых значений или до сброса все битов экземпляров перечисления.
3. Если после проверки всех числовых значений экземпляр перечисления все еще не равен нулю, это означает наличие несброшенных битов, которым не сопоставлены идентификаторы. В этом случае метод возвращает исходное число экземпляра перечисления в виде строки.
4. Если исходное значение экземпляра перечисления не равно нулю, метод возвращает набор символов, разделенных запятой.
5. Если исходным значением экземпляра перечисления был ноль, а в перечислимом типе есть идентификатор с таким значением, метод возвращает этот идентификатор.
6. Если алгоритм доходит до данного шага, возвращается 0.

Тип Actions можно определить и без атрибута [Flags], получив при этом правильную результирующую строку. Для этого достаточно указать формат "F":

```
// [Flags] // Теперь это просто комментарий
internal enum Actions {
    None = 0
    Read = 0x0001,
    Write = 0x0002,
    ReadWrite = Actions.Read | Actions.Write,
    Delete = 0x0004,
    Query = 0x0008,
    Sync = 0x0010
}

Actions actions = Actions.Read | Actions.Delete; // 0x0005
Console.WriteLine(actions.ToString("F"));        // "Read, Delete"
```

Если числовое значение содержит бит, которому не соответствует какой-либо идентификатор, в возвращаемой строке окажется только десятичное число, равное исходному значению, и ни одного идентификатора.

Заметьте: идентификаторы, которые вы определяете в перечислимом типе, не обязаны быть степенью двойки. Например, в типе Actions можно описать идентификатор с именем All, имеющий значение 0x001F. Результатом форматирования экземпляра типа Actions со значением 0x001F станет строка "All". Других идентификаторов в строке не будет.

Пока мы говорили лишь о преобразовании числовых значений в строку флагов. Однако вы можете также получить числовое значение строки, содержащей

разделенные запятой идентификаторы, воспользовавшись статическим методом `Parse` типа `Enum` или методом `TryParse`. Рассмотрим это на примере:

```
// Так как Query определен как 8, 'a' получает начальное значение 8
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query", true);
Console.WriteLine(a.ToString()); // "Query"

// Так как у нас определены и Query, и Read, 'a' получает
// начальное значение 9
Enum.TryParse<Actions>("Query, Read", false, out a);
Console.WriteLine(a.ToString()); // "Read, Query"

// Создаем экземпляр перечисления Actions enum со значением 28
a = (Actions) Enum.Parse(typeof(Actions), "28", false);
Console.WriteLine(a.ToString()); // "Delete, Query, Sync"
```

Алгоритм работы методов `Parse` и `TryParse` таков:

1. Удаляются все пробелы в начале и конце строки.
2. Если первым символом в строке является цифра, знак «плюс» (+) или знак «минус» (–), строка считается числом и возвращается экземпляр перечисления, числовое значение которого совпадает с числом, полученным в результате преобразования строки.
3. Переданная строка разбивается на разделенные запятыми символы, и у каждого символа удаляются все пробелы в начале и конце.
4. Выполняется поиск каждой символьной строки среди идентификаторов перечисления. Если символ найти не удастся, метод `Parse` вбрасывает исключение `System.ArgumentException`, а метод `TryParse` возвращает значение `false`. При обнаружении символа его числовое значение путем дизъюнкции (OR) присоединяется к результирующему значению, и метод переходит к анализу следующего символа.
5. После обнаружения и анализа всех символов результат возвращается программе.

Никогда не следует применять метод `IsDefined` с перечислимыми типами битовых флагов. Это не будет работать по двум причинам:

- ❑ Переданную ему строку метод не разбивает на отдельные символы, а ищет целиком, вместе с запятыми. Однако в перечислениях невозможен идентификатор, содержащий запятые, а значит, результат поиска всегда будет нулевым.
- ❑ После передачи ему числового значения метод ищет всего один символ перечислимого типа, значение которого совпадает с переданным числом. Для битовых флагов вероятность получения положительного результата при таком сравнении ничтожно мала, и обычно метод возвращает значение `false`.

Добавление методов к перечислимым типам

В начале главы уже упоминалось, что определить метод как часть перечислимого типа невозможно. Это ограничение удручало меня в течение многих лет, так как то и дело возникали ситуации, когда требовалось снабдить перечислимые типы методами. К счастью, теперь его можно обойти при помощи относительно нового для C# механизма *методов расширения* (extension method), который подробно рассматривается в главе 8.

Для добавления методов к перечислимому типу `FileAttributes` нужно определить статический класс с методами расширения. Делается это следующим образом:

```
internal static class FileAttributesExtensionMethods {
    public static Boolean IsSet(
        this FileAttributes flags, FileAttributes flagToTest) {
        if (flagToTest == 0)
            throw new ArgumentOutOfRangeException(
                "flagToTest", "Value must not be 0");
        return (flags & flagToTest) == flagToTest;
    }

    public static Boolean IsClear(
        this FileAttributes flags, FileAttributes flagToTest) {
        if (flagToTest == 0)
            throw new ArgumentOutOfRangeException(
                "flagToTest", "Value must not be 0");
        return !IsSet(flags, flagToTest);
    }

    public static Boolean AnyFlagsSet(
        this FileAttributes flags, FileAttributes testFlags) {
        return ((flags & testFlags) != 0);
    }

    public static FileAttributes Set(
        this FileAttributes flags, FileAttributes setFlags) {
        return flags | setFlags;
    }

    public static FileAttributes Clear(
        this FileAttributes flags, FileAttributes clearFlags) {
        return flags & ~clearFlags;
    }
}
```

```
public static void ForEach(this FileAttributes flags,
    Action<FileAttributes> processFlag) {
    if (processFlag == null) throw new ArgumentNullException("processFlag");
    for (UInt32 bit = 1; bit != 0; bit <= 1) {
        UInt32 temp = ((UInt32)flags) & bit;
        if (temp != 0) processFlag((FileAttributes)temp);
    }
}
```

Далее показан код, демонстрирующий вызов одного из таких методов. Как легко заметить, он выглядит так, как выглядел бы вызов методов перечислимого типа:

```
FileAttributes fa = FileAttributes.System;
fa = fa.Set(FileAttributes.ReadOnly);
fa = fa.Clear(FileAttributes.System);
fa.ForEach(f => Console.WriteLine(f));
```

Глава 16. Массивы

Массив представляет собой механизм, позволяющий рассматривать отдельные элементы как единую коллекцию. Общеязыковая исполняющая среда Microsoft .NET (CLR) поддерживает *одномерные* (single-dimension), *многомерные* (multidimension) и *вложенные* (jagged) массивы. Базовым для всех массивов является тип System.Array, производный от System.Object. А это значит, что массивы всегда относятся к ссылочному типу и размещаются в управляемой куче, а переменная в приложении содержит не сам массив, а ссылку на него. Рассмотрим пример:

```
Int32[] myIntegers;           // Объявление ссылки на массив
myIntegers = new Int32[100]; // Создание массива типа Int32 из 100 элементов
```

В первой строке объявляется переменная myIntegers, которая будет ссылаться на одномерный массив элементов типа Int32. Вначале ей присваивается значение null, так как память под массив пока не выделена. Во второй строке выделяется память под 100 значений типа Int32; и всем им присваивается начальное значение 0. Поскольку массивы относятся к ссылочным типам, блок памяти для хранения 100 неупакованных экземпляров типа Int32 выделяется в управляемой куче. Вообще говоря, помимо элементов массива в этом блоке размещается указатель на тип объекта и значение SyncBlockIndex, а также некоторые дополнительные члены. Адрес этого блока памяти заносится в переменную myIntegers.

Можно также создать массивы ссылочного типа:

```
Control[] myControls;           // Объявление ссылки на массив
myControls = new Control[50]; // Создание массива из 50 ссылок
                               // на переменную Control
```

Переменная myControls из первой строки может указывать на одномерный массив ссылок на элементы Control. Вначале ей присваивается значение null, ведь память под массив пока не выделена. Во второй строке выделяется память под 50 ссылок на Control, и все они инициализируются значением null. Поскольку Control относится к ссылочным типам, массив формируется путем создания ссылок, а не каких-либо реальных объектов. Возвращенный адрес блока памяти заносится в переменную myControls.

На рис. 16.1 показано, как выглядят массивы значимого и ссылочного типов в управляемой куче.

На этом рисунке показан массив Controls после выполнения следующих инструкций:

```
myControls[1] = new Button();
myControls[2] = new TextBox();
myControls[3] = myControls[2]; // Два элемента ссылаются на один объект
```

```
myControls[46] = new DataGrid();
myControls[48] = new ComboBox();
myControls[49] = new Button();
```

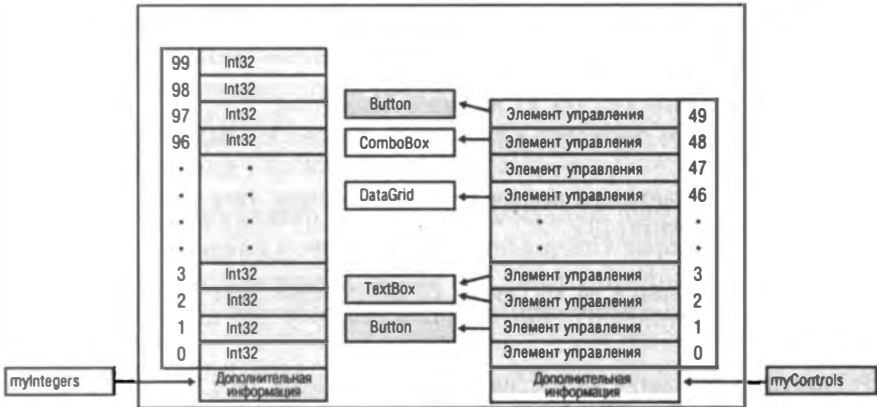


Рис. 16.1. Массивы значимого и ссылочного типов в управляемой куче

Согласно общезыковой спецификации (CLS), нумерация элементов в массиве должна начинаться с нуля. Только в этом случае методы, написанные на С#, смогут передать ссылку на созданный массив коду, написанному на другом языке, скажем, на Microsoft Visual Basic .NET. Кроме того, поскольку массивы с начальным нулевым индексом получили очень большое распространение, специалисты Microsoft постарались оптимизировать их работу. Тем не менее иные варианты индексации массивов в CLR допускаются, хотя это не приветствуется. В случаях когда производительность и межъязыковая совместимость программ не имеют большого значения, можно использовать массивы, начальный индекс которых отличен от 0. Мы подробно рассмотрим их чуть позже.

На рисунке видно, что в массиве присутствует некая *дополнительная информация*. Это сведения о размерности массива, его нижней границе (чаще всего это 0) и количестве элементов в каждом измерении. Здесь же указывается тип элементов массива. Далее мы рассмотрим методы запроса этих данных.

Пока что нам известен только процесс создания одномерных массивов. По возможности нужно ограничиваться одномерными массивами с нулевым начальным индексом, которые называют иногда *SZ-массивами*, или *векторами*. Векторы обеспечивают наилучшую производительность, поскольку для операций с ними используются команды промежуточного языка (Intermediate Language, IL), например, `newarr`, `ldelem`, `ldlema`, `ldlen` и `stelem`. Впрочем, если у вас есть такое желание, можно применять и многомерные массивы. Вот как они создаются:

```
// Создание двумерного массива типа Doubles
Double[,] myDoubles = new Double[10, 20];
```

```
// Создание трехмерного массива ссылок на строки
String[,,] myStrings = new String[5, 3, 10];
```

CLR поддерживает также *вложенные* (jagged) массивы. Производительность одномерных вложенных массивов с нулевым начальным индексом такая же, как у обычных векторов. Однако обращение к элементу вложенного массива означает обращение к двум или больше массивам одновременно. Вот пример массива многоугольников, где каждый многоугольник состоит из массива экземпляров типа `Point`:

```
// Создание одномерного массива из массивов типа Point
Point[][] myPolygons = new Point[3][];

// myPolygons[0] ссылается на массив из 10 экземпляров типа Point
myPolygons[0] = new Point[10];

// myPolygons[1] ссылается на массив из 20 экземпляров типа Point
myPolygons[1] = new Point[20];

// myPolygons[2] ссылается на массив из 30 экземпляров типа Point
myPolygons[2] = new Point[30];

// вывод точек первого многоугольника
for (Int32 x = 0; x < myPolygons[0].Length; x++)
    Console.WriteLine(myPolygons[0][x]);
```

ПРИМЕЧАНИЕ

CLR проверяет корректность индексов. То есть если у вас имеется массив, состоящий из 100 элементов с индексами от 0 до 99, попытка обратиться к его элементу по индексу -5 или 100 станет причиной исключения `System.IndexOutOfRangeException`. Доступ к памяти за пределами массива нарушает безопасность типов и создает брешь в защите, недопустимую для верифицированного CLR-кода. Проверка индекса обычно не влияет на производительность, так как компилятор выполняет ее всего один раз перед началом цикла, а не на каждой итерации. Впрочем, если вы считаете, что проверка индексов влияет на скорость выполнения программы, используйте для доступа к массиву небезопасный код. Эта процедура рассмотрена в разделе «Производительность доступа к массиву» данной главы.

Инициализация элементов массива

В предыдущем разделе рассмотрена процедура создания элементов массива и присвоения им начальных значений. Синтаксис C# позволяет совместить эти операции:

```
String[] names = new String[] { "Aidan", "Grant" };
```

Набор разделенных запятой символов в фигурных скобках называется *инициализатором массива* (array initializer). Сложность каждого символа может быть произвольной, а в случае многомерного массива инициализатор может оказаться вложенным. В показанном примере фигурирует всего два простых выражении типа String. Объявляя в методе локальную переменную, ссылающуюся на инициализированный массив, можно упростить код, воспользовавшись переменной неявного типа var:

```
// Использование локальной переменной неявного типа:  
var names = new String[] { "Aidan", "Grant" };
```

В результате компилятор причислит локальную переменную names к типу String[], так как именно к этому типу принадлежит выражение, расположенное справа от оператора присваивания (=). Обратите внимание, что в следующем фрагменте кода отсутствует спецификация типа между операторами new и []:

```
// Задание типа массива с помощью локальной переменной неявного типа:  
var names = new[] { "Aidan", "Grant", null };
```

Компилятор определяет тип выражений, используемых для инициализации элементов массива, и по результатам выбирает базовый класс, который лучше всего описывает все элементы. В показанном примере компилятор обнаруживает два элемента типа String и значение null. Но так как последнее может быть назначено любому ссылочному типу, выбор делается в пользу создания и инициализации массива ссылок типа String.

Еще пример:

```
// Ошибочное задание типа массива с помощью локальной  
// переменной неявного типа  
var names = new[] { "Aidan", "Grant", 123 };
```

На такой код компилятор реагирует сообщением (ошибка CS0826: подходящего типа для неявно заданного массива не обнаружено):

```
error CS0826: No best type found for implicitly-typed array
```

Дело в том, что общим базовым типом для двух строк и значения типа Int32 является тип Object. Для компилятора это означает необходимость создать массив ссылок типа Object, а затем упаковать значение типа Int32 и заставить последний элемент массива ссылаться на результат упаковки, имеющий значение 123. Разработчики сочли задачу упаковки элементов массива слишком сложной для режима выполнения, предлагаемого по умолчанию, поэтому в такой ситуации просто появляется сообщение об ошибке.

В качестве синтаксического бонуса можно указать возможность вот такой инициализации массива:

```
String[] names = { "Aidan", "Grant" };
```

Обратите внимание, что справа от оператора присваивания располагаются только начальные значения элементов массива. Ни оператора `new`, ни типа, ни квадратных скобок там нет. К сожалению, в этом случае компилятор не разрешает использовать локальные переменные неявного типа:

```
// Ошибочное использование локальной переменной
var names = { "Aidan", "Grant" };
```

Попытка компиляции такой строчки приведет к появлению двух сообщений:

```
error CS0820: Cannot initialize an implicitly-typed local variable with an
array initializer
error CS0622: Can only use array initializer expressions to assign array types.
Try using a new expression instead
```

Первое говорит о том, что локальной переменной неявного типа невозможно присвоить начальное значение при помощи инициализатора массива, а второе информирует, что данный инициализатор применяется только для назначения типов массивам и рекомендует вам воспользоваться оператором `new`. В принципе, компилятор вполне способен выполнить все эти действия самостоятельно, но разработчики решили, что это — слишком сложная задача. Ведь пришлось бы определять тип массива, создавать его при помощи оператора `new`, присваивать элементам начальные значения, а кроме того, определять тип локальной переменной.

Напоследок хотелось бы рассмотреть процедуру неявного задания типа массива в случае анонимных типов и локальных переменных неявного типа. (Об анонимных типах см. главу 10.)

Рассмотрим следующий код:

```
// Применение переменных и массивов неявно заданного типа.
// а также анонимного типа:
var kids = new[] {new { Name="Aidan" }, new { Name="Grant" }};
```

```
// Пример применения (с другой локальной переменной неявно заданного типа):
foreach (var kid in kids)
    Console.WriteLine(kid.Name);
```

В этом примере для присваивания начальных значений элементам массива используются два выражения, каждое из которых представляет собой анонимный тип (ведь после оператора `new` ни в одном из случаев не фигурирует имя типа). Благодаря идентичной структуре этих выражений (поле `Name` типа `String`) компилятор относит оба объекта к одному типу. Теперь мы можем воспользоваться возможностью неявного задания типа массива (когда между оператором `new` и квадратными скобками отсутствует имя типа). В результате компилятор самостоятельно определит тип, сконструирует массив и инициализирует его элементы как ссылки на два экземпляра одного и того же анонимного типа. В итоге ссылка на этот объект присваивается локальной переменной `kids`, тип которой определит компилятор.

Затем только что созданный и инициализированный массив используется в цикле `foreach`, в котором фигурирует и переменная `kid` неявного типа. Вот результат выполнения такого кода:

Aidan
Grant

Приведение типов в массивах

В CLR для массивов с элементами ссылочного типа допустимо неявное приведение. В рамках решения этой задачи оба типа массивов должны быть одной размерности, кроме того, должно иметь место неявное или явное преобразование из типа элементов исходного массива в целевой тип. Массивы с элементами значимых типов для подобных преобразований не подходят. Впрочем, данное ограничение можно обойти при помощи метода `Array.Copy`, который создает новый массив и заполняет его элементами.

Вот пример приведения типа в массиве:

```
// Создание двумерного массива FileStream
FileStream[,] fs2dim = new FileStream[5, 10];

// Неявное приведение к массиву типа Object
Object[,] o2dim = fs2dim;

// Невозможно приведение двумерного массива к одномерному
// Ошибка компиляции CS0030: Невозможно преобразовать тип 'object[*,*]' в
// 'System.IO.Stream[]'
Stream[] s1dim = (Stream[]) o2dim;

// Явное приведение к двумерному массиву Stream
Stream[,] s2dim = (Stream[,]) o2dim;

// Явное приведение к двумерному массиву String
// Компилируется, но во время выполнения
// вбрасывается исключение InvalidCastException
String[,] st2dim = (String[,]) o2dim;

// Создание одномерного массива Int32 (значимый тип)
Int32[] i1dim = new Int32[5];

// Невозможно приведение массива значимого типа
// Ошибка компиляции CS0030: Невозможно преобразовать
// тип 'int[]' в 'object[]'
Object[] oldim = (Object[]) i1dim;
```



```
// Создание нового массива и приведение элементов к нужному типу
// при помощи метода Array.Copy
// Создаем массив ссылок на упакованные элементы типа Int32
Object[] obldim = new Object[ildim.Length];
Array.Copy(ildim, obldim, ildim.Length);
```

Метод `Array.Copy` не просто копирует элементы одного массива в другой. Он действует как функция `memmove` языка C, но при этом правильно обрабатывает перекрывающиеся области памяти. Он также способен при необходимости преобразовывать элементы массива в процессе их копирования. Метод `Copy` выполняет следующие действия:

- ❑ Упаковку элементов значимого типа в элементы ссылочного типа, например копирование `Int32[]` в `Object[]`.
- ❑ Распаковку элементов ссылочного типа в элементы значимого типа, например копирование `Object[]` в `Int32[]`.
- ❑ Расширение (*widening*) элементарных значимых типов, например копирование `Int32[]` в `Double[]`.
- ❑ Понижающее приведение в случаях, когда совместимость массивов невозможно определить по их типам. Сюда относятся, к примеру, приведение массива типа `Object[]` в массив типа `IFormattable[]`. Если все объекты в массиве `Object[]` реализуют интерфейс `IFormattable[]`, приведение пройдет успешно.

Вот еще один пример применения метода `Copy`:

```
// Определение значимого типа, реализующего интерфейс
internal struct MyValueType : IComparable {
    public Int32 CompareTo(Object obj) {
        ...
    }
}

public static class Program {
    public static void Main() {
        // Создание массива из 100 элементов значимого типа
        MyValueType[] src = new MyValueType[100];
        // Создание массива ссылок IComparable
        IComparable[] dest = new IComparable[src.Length];
        // Присваивание элементам массива IComparable ссылок на упакованные
        // версии элементов исходного массива
        Array.Copy(src, dest, src.Length);
    }
}
```

Нетрудно догадаться, что FCL достаточно часто использует достоинства метода `Array.Copy`.

Бывают ситуации, когда полезно изменить тип массива, то есть выполнить его *ковариацию* (*array covariance*). Однако следует помнить, что эта

операция сказывается на производительности. Допустим, вы написали такой код:

```
String[] sa = new String[100];
Object[] oa = sa; // oa ссылается на массив элементов типа String
oa[5] = "Jeff";    // CLR проверяет принадлежность oa к типу String;
                  // Проверка проходит успешно
oa[3] = 5;         // CLR проверяет принадлежность oa к типу Int32;
                  // Вбрасывается исключение ArrayTypeMismatchException
```

В этом коде переменная `oa`, тип которой определен как `Object[]`, ссылается на массив типа `String[]`. Затем вы пытаетесь присвоить одному из элементов этого массива значение `5`, относящееся к типу `Int32`, производному от типа `Object`. Естественно, CLR проверяет корректность такого присваивания, то есть в процессе выполнения контролирует наличие в массиве элементов типа `Int32`. В данном случае такие элементы отсутствуют, что и становится причиной исключения `ArrayTypeMismatchException`.

ПРИМЕЧАНИЕ

Для простого копирования части элементов из одного массива в другой имеет смысл использовать метод `BlockCopy` класса `System.Buffer`, который работает быстрее метода `Array.Copy`. К сожалению, этот метод поддерживает только элементарные типы и не имеет таких же широких возможностей приведения, как `Array.Copy`. Параметры типа `Int32` выражаются путем смещения байтов внутри массива, а не при помощи индексов. То есть метод `BlockCopy` подходит для поразрядного копирования совместимых данных из массива одного типа в другой. К примеру, таким способом можно скопировать массив типа `Byte[]`, содержащий символы `Unicode`, в массив типа `Char[]`. Этот метод частично компенсирует отсутствие возможности считать массив просто блоком памяти произвольного типа.

Для надежного копирования набора элементов из одного массива в другой используйте метод `ConstrainedCopy` класса `System.Array`. Он гарантирует, что в случае неудачного копирования появится исключение, но данные в целевом массиве не затрагиваются. Это позволяет использовать метод `ConstrainedCopy` в области ограниченного выполнения (`Constrained Execution Region`, `CER`). Гарантии, которые он дает, обусловлены требованием, чтобы тип элементов исходного массива совпадал с типом элементов целевого или был производным от него. Кроме того, метод не поддерживает упаковку, распаковку или нисходящее приведение.

Базовый класс System.Array

Рассмотрим объявление переменной массива:

```
FileStream[] fsArray;
```

Объявление переменной массива подобным образом приводит к автоматическому созданию типа `FileStream[]` для домена приложений.

Тип `FileStream[]` является производным от `System.Array` и соответственно наследует оттуда все методы и свойства. Для их вызова служит переменная `fsArray`. Это упрощает работу с массивами, ведь в классе `System.Array` есть множество полезных методов и свойств, в том числе `Clone`, `CopyTo`, `GetLength`, `GetLongLength`, `GetLowerBound`, `GetUpperBound`, `Length` и `Rank`.

Класс `System.Array` содержит также статические методы для работы с массивами, в том числе `AsReadOnly`, `BinarySearch`, `Clear`, `ConstrainedCopy`, `ConvertAll`, `Copy`, `Exists`, `Find`, `FindAll`, `FindIndex`, `FindLast`, `FindLastIndex`, `ForEach`, `IndexOf`, `LastIndexOf`, `Resize`, `Reverse`, `Sort` и `TrueForAll`. В качестве параметра они принимают ссылку на массив. Кроме того, для этих методов существуют перегруженные версии. Более того, для многих из них имеются обобщенные перегруженные версии, обеспечивающие контроль типов во время компиляции и высокую производительность. Я настоятельно рекомендую самостоятельно почитать о них в документации на SDK.

Реализация интерфейсов `IEnumerable`, `ICollection` и `IList`

Многие методы работают с коллекциями, поскольку они объявлены с такими параметрами, как интерфейсы `IEnumerable`, `ICollection` и `IList`. Им можно передавать и массивы, так как эти три необобщенных интерфейса реализованы в классе `System.Array`. Данная реализация возможна благодаря тому, что эти интерфейсы считают любой элемент принадлежащим классу `System.Object`. Однако хотелось бы также, чтобы класс `System.Array` реализовывал обобщенные эквиваленты этих интерфейсов, обеспечивая лучший контроль типов во время компиляции и повышенную производительность.

Команда разработчиков CLR решила, что не стоит осуществлять реализацию интерфейсов `IEnumerable<T>`, `ICollection<T>` и `IList<T>` классом `System.Array`, так как в этом случае возникают проблемы с многомерными массивами, а также с массивами, в которых нумерация не начинается с нуля. Ведь определение этих интерфейсов в указанном классе означает необходимость поддержки массивов всех типов. Вместо этого разработчики пошли на хитрость: при создании одномерного массива с начинающейся с нуля индексацией CLR автоматически реализует интерфейсы `IEnumerable<T>`, `ICollection<T>` и `IList<T>` (здесь `T` — тип элементов массива), а также три интерфейса для всех базовых типов массива при условии, что эти типы являются ссылочными. Ситуацию иллюстрирует следующая иерархия.

`Object`

```
Array (non-generic IEnumerable, ICollection, IList)
    Object[] (IEnumerable, ICollection, IList of Object)
```

```
String[] (IEnumerable, ICollection, IList of String)
Stream[] (IEnumerable, ICollection, IList of Stream)
FileStream[] (IEnumerable, ICollection, IList of FileStream)
.
.      (другие массивы ссылочных типов)
.
```

Пример:

```
FileStream[] fsArray;
```

В этом случае при создании типа `FileStream[]` CLR автоматически анализирует в нем интерфейсы `IEnumerable<FileStream>`, `ICollection<FileStream>` и `IList<FileStream>`. Более того, тип `FileStream[]` будет реализовывать интерфейсы базовых классов `IEnumerable<Stream>`, `IEnumerable<Object>`, `ICollection<Stream>`, `ICollection<Object>`, `IList<Stream>` и `IList<Object>`. Так как все эти интерфейсы реализуются средой CLR автоматически, переменная `fsArray` может применяться во всех случаях использования этих интерфейсов. Например, ее можно передавать в методы с такими прототипами:

```
void M1(IList<FileStream> fsList) { ... }
void M2(ICollection<Stream> sCollection) { ... }
void M3(IEnumerable<Object> oEnumerable) { ... }
```

Обратите внимание, что если массив содержит элементы значимого типа, класс, которому он принадлежит, не будет реализовывать интерфейсы базовых классов элемента. Например:

```
DateTime[] dtArray; // Массив элементов значимого типа
```

В данном случае тип `DateTime[]` будет реализовывать только интерфейсы `IEnumerable<DateTime>`, `ICollection<DateTime>` и `IList<DateTime>`; версии этих интерфейсов, общие для классов `System.ValueType` или `System.Object`, реализованы не будут. А это значит, что переменную `dtArray` нельзя передать показанному ранее методу `M3` в качестве аргумента. Ведь массивы значимых и ссылочных типов располагаются в памяти по-разному (об этом рассказывалось в начале данной главы).

Передача и возврат массивов

Передавая массив в метод в качестве аргумента, вы на самом деле передаете ссылку на него. А значит, метод может модифицировать элементы массива. Этого можно избежать, передав в качестве аргумента копию массива. Имейте в виду, что метод `Array.Copy` выполняет неполное копирование, и если элементы массива относятся к ссылочному типу, в новом массиве окажутся ссылки на существующие объекты.

Аналогично, отдельные методы возвращают ссылку на массив. Если метод создает и инициализирует массив, возвращение такой ссылки не вызывает

проблем; если же вы хотите, чтобы метод возвращал ссылку на внутренний массив, ассоциированный с полем, то сначала решите, вправе ли вызывающая программа иметь доступ к этому массиву. Как правило, делать этого не стоит. Поэтому лучше пусть метод создаст массив, вызовет метод `Array.Copy`, а затем вернет ссылку на новый массив. Еще раз напомним, что данный метод выполняет неполное копирование исходного массива.

Результатом вызова метода, возвращающего ссылку на массив, не содержащий элементов, является либо значение `null`, либо ссылка на массив с нулевым числом элементов. В такой ситуации Microsoft настоятельно рекомендует второй вариант, поскольку подобная реализация упрощает код. К примеру, данный код выполняется правильно даже при отсутствии элементов, подлежащих обработке:

```
// Пример простого для понимания кода
Appointment[] appointments = GetAppointmentsForToday();
for (Int32 a = 0; a < appointments.Length; a++) {
    ...
}
```

Следующий фрагмент кода также корректно выполняется при отсутствии элементов, но он уже сложнее:

```
// Пример более сложного кода
Appointment[] appointments = GetAppointmentsForToday();
if (appointments != null) {
    for (Int32 a = 0, a < appointments.Length; a++) {
        // Выполняем действия с элементом appointments[a]
    }
}
```

Вызывающим программам требуется меньше времени на обслуживание методов, которые вместо `null` возвращают массивы с нулевым числом элементов. То же относится к полям. Если у вашего типа есть поле, являющееся ссылкой на массив, то в него следует помещать ссылку на массив, даже если в массиве отсутствуют элементы.

Массивы с ненулевой нижней границей

Как уже упоминалось, массивы с ненулевой нижней границей вполне допустимы. Создавать их можно при помощи статического метода `CreateInstance` типа `Array`. Существует несколько перегруженных версий этого метода, позволяющих задать тип элементов, размерность, нижнюю границу массива, а также количество элементов в каждом измерении. Метод выделяет память, записывает заданные параметры в служебную область выделенного блока и возвращает

ссылку на массив. При наличии двух и более измерений ссылку, возвращенную методом `CreateInstance`, можно привести к типу переменной `ElementType[]` (здесь `ElementType` — имя типа), чтобы упростить доступ к элементам массива. Для доступа к элементам одномерных массивов пользуйтесь методами `GetValue` и `SetValue` класса `Array`.

Рассмотрим процесс динамического создания двухмерного массива значений типа `System.Decimal`. Первое измерение составят годы с 2005 по 2009 включительно, а второе — кварталы с 1 по 4 включительно. Все элементы обрабатываются в цикле. Прописав в коде границы массива в явном виде, мы получили бы выигрыш в производительности, но вместо этого воспользуемся методами `GetLowerBound` и `GetUpperBound` класса `System.Array`:

using System;

```
public static class DynamicArrays {
    public static void Main() {
        // Требуется двухмерный массив [2005..2009][1..4]
        Int32[] lowerBounds = { 2005, 1 };
        Int32[] lengths = { 5, 4 };
        Decimal[,] quarterlyRevenue = (Decimal[,])
            Array.CreateInstance(typeof(Decimal), lengths, lowerBounds);

        Console.WriteLine("{0,4} {1,9} {2,9} {3,9} {4,9}",
            "Year", "Q1", "Q2", "Q3", "Q4");
        Int32 firstYear = quarterlyRevenue.GetLowerBound(0);
        Int32 lastYear = quarterlyRevenue.GetUpperBound(0);
        Int32 firstQuarter = quarterlyRevenue.GetLowerBound(1);
        Int32 lastQuarter = quarterlyRevenue.GetUpperBound(1);

        for (Int32 year = firstYear; year <= lastYear; year++) {
            Console.Write(year + " ");
            for (Int32 quarter = firstQuarter;
                quarter <= lastQuarter; quarter++) {
                Console.Write("{0,9:C} ", quarterlyRevenue[year, quarter]);
            }
            Console.WriteLine();
        }
    }
}
```

После компиляции и выполнения этого кода получаем:

Year	Q1	Q2	Q3	Q4
2005	\$0.00	\$0.00	\$0.00	\$0.00
2006	\$0.00	\$0.00	\$0.00	\$0.00
2007	\$0.00	\$0.00	\$0.00	\$0.00
2008	\$0.00	\$0.00	\$0.00	\$0.00
2009	\$0.00	\$0.00	\$0.00	\$0.00

Производительность доступа к массиву

В CLR поддерживаются массивы двух типов:

- ❑ Одномерные массивы с нулевым начальным индексом. Иногда их называют *SZ-массивами* (от английского single-dimensional, zero-based), или *векторами*.
- ❑ Одномерные и многомерные массивы с неизвестным начальным индексом.

Рассмотрим их на примере следующего кода (результат его работы приводится в комментариях):

```
using System;
public sealed class Program {
    public static void Main() {
        Array a;

        // Создание одномерного массива с нулевым
        // начальным индексом и без элементов
        a = new String[0];
        Console.WriteLine(a.GetType()); // "System.String[]"

        // Создание одномерного массива с нулевым
        // начальным индексом и без элементов
        a = Array.CreateInstance(typeof(String),
            new Int32[] { 0 }, new Int32[] { 0 });
        Console.WriteLine(a.GetType()); // "System.String[]"

        // Создание одномерного массива с начальным индексом 1 и без элементов
        a = Array.CreateInstance(typeof(String),
            new Int32[] { 0 }, new Int32[] { 1 });
        Console.WriteLine(a.GetType()); // "System.String[*]" <-- ВНИМАНИЕ!

        Console.WriteLine();

        // Создание двухмерного массива с нулевым
        // начальным индексом и без элементов
        a = new String[0, 0];
        Console.WriteLine(a.GetType()); // "System.String[,]"

        // Создание двухмерного массива с нулевым
        // начальным индексом и без элементов
        a = Array.CreateInstance(typeof(String),
            new Int32[] { 0, 0 }, new Int32[] { 0, 0 });
        Console.WriteLine(a.GetType()); // "System.String[,]"
```

```
// Создание двумерного массива с начальным индексом 1 и без элементов
a = Array.CreateInstance(typeof(String),
    new Int32[] { 0, 0 }, new Int32[] { 1, 1 });
Console.WriteLine(a.GetType()); // "System.String[]"
}
```

Рядом с каждой инструкцией `Console.WriteLine` в виде комментария показан результат действия. Для одномерных массивов с нулевой нижней границей это `System.String[]`, если же индексация начинается с единицы, выводится уже `System.String[*]`. Знак `*` указывает на осведомленность CLR о ненулевой нижней границе. Так как в C# объявить переменную типа `String[*]` невозможно, синтаксис этого языка запрещает обращение к одномерным массивам с ненулевой нижней границей. Впрочем, обойти это ограничение можно с помощью методов `GetValue` и `SetValue` класса `Array`, но дополнительные затраты на вызов метода снижают эффективность работы программы.

Для многомерных массивов, независимо от нижней границы, отображается один и тот же тип: `System.String[.]`. Во время выполнения CLR рассматривает их как массивы с ненулевой нижней границей. Логично было бы предположить, что имя типа будет представлено как `System.String[*,*]`, но в CLR для многомерных массивов не используется знак `*`. Ведь иначе он выводился бы во всех случаях, создавая путаницу.

Доступ к элементам одномерного массива с нулевой нижней границей осуществляется немного быстрее, чем доступ к элементам многомерных массивов или массивов с ненулевой нижней границей. Есть несколько причин такому поведению. Во-первых, специальные команды для работы с одномерными массивами с нулевой нижней границей (`newarr`, `ldelem`, `ldlema`, `ldlen` и `stelem`) позволяют JIT-компилятору генерировать оптимизированный код. При этом предполагается, что первый индекс равен нулю, то есть при доступе к элементам отсутствует необходимость вычислять смещение. Кроме того, в общем случае компилятор умеет выносить код проверки границ за пределы цикла. К примеру, рассмотрим следующий код:

```
using System;

public static class Program {
    public static void Main() {
        Int32[] a = new Int32[5];
        for(Int32 index = 0; index < a.Length; index++) {
            // Какие-то действия с элементом a[index]
        }
    }
}
```

Обратите внимание на вызов свойства `Length` в проверочном выражении цикла `for`. Фактически при этом вызывается метод. Но JIT-компилятор «знает»,

что свойство `Length` принадлежит классу `Array`, поэтому создает код, в котором оно вызывается всего один раз, сохраняя полученный результат в промежуточной переменной. Именно ее значение проверяется на каждой итерации цикла. В результате такой код работает очень быстро. Некоторые разработчики недооценивают возможности JIT-компилятора и пишут «умный код», пытаясь помочь его работе. Однако такие попытки практически всегда приводят к снижению производительности, а также делают готовую программу непонятной и неудобной для редактирования. Поэтому пусть свойство `Length` вызывается автоматически.

Кроме того, JIT-компилятор «знает», что цикл обращается к элементам массива с нулевой нижней границей, указывая `Length - 1`. Поэтому он в процессе выполнения генерирует код, проверяющий, все ли элементы находятся в границах массива. А именно, проверяется условие:

```
(0 >= a.GetLowerBound(0)) && ((Length - 1) <= a.GetUpperBound(0))
```

Проверка осуществляется до начала цикла. В случае положительного результата компилятор не создает в теле цикла кода, проверяющего, не вышел ли индекс элемента за границы диапазона. Именно за счет этого обеспечивается высокая производительность доступа к массиву.

К сожалению, обращение к элементам многомерного массива или массива с ненулевой нижней границей происходит намного медленней. Ведь в этих случаях код проверки индекса не выносится за пределы цикла и проверка осуществляется на каждой итерации. Кроме того, компилятор добавляет код, вычитающий из текущего индекса нижнюю границу массива. Это также замедляет работу программы даже в случае многомерных массивов с нулевой нижней границей.

Если вы серьезно озабочены проблемой производительности, имеет смысл использовать вложенные массивы. Кроме того, в C# и CLR возможен доступ к элементам массива при помощи небезопасного (неверифицируемого) кода. В этом случае процедура проверки индексов массива просто отключается. Данная техника применима только к массивам типа `SByte`, `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Char`, `Single`, `Double`, `Decimal`, `Boolean`, а также к массивам перечислимого типа или структуры значимого типа с полями одного из вышеуказанных типов.

Эту мощную возможность следует использовать крайне осторожно, так как она дает прямой доступ к памяти. При этом выход за границы массива не сопровождается вбрасыванием исключения, вместо этого происходит повреждение памяти, нарушение безопасности типов и, скорее всего, в системы безопасности программы появляется брешь. Поэтому сборке, содержащей небезопасный код, следует обеспечить полное доверие или же предоставить разрешение `Security Permission`, включив свойство `Skip Verification`.

Следующий код демонстрирует три варианта доступа к двумерному массиву, включая безопасный доступ, доступ через вложенный массив и небезопасный доступ:

```
using System;
using System.Diagnostics;

public static class Program {
    private const Int32 c_numElements = 10000;

    public static void Main() {
        const Int32 testCount = 10;
        Stopwatch sw;

        // Объявление двумерного массива
        Int32[,] a2Dim = new Int32[c_numElements, c_numElements];

        // Объявление двумерного массива как вложенного
        Int32[][] aJagged = new Int32[c_numElements][];
        for (Int32 x = 0; x < c_numElements; x++)
            aJagged[x] = new Int32[c_numElements];

        // 1: Обращение к элементам стандартным, безопасным способом
        sw = Stopwatch.StartNew();
        for (Int32 test = 0; test < testCount; test++)
            Safe2DimArrayAccess(a2Dim);
        Console.WriteLine("{0}: Safe2DimArrayAccess", sw.Elapsed);

        // 2: Обращение к элементам методом вложенных массивов
        sw = Stopwatch.StartNew();
        for (Int32 test = 0; test < testCount; test++)
            SafeJaggedArrayAccess(aJagged);
        Console.WriteLine("{0}: SafeJaggedArrayAccess", sw.Elapsed);

        // 3: Обращение к элементам небезопасным методом
        sw = Stopwatch.StartNew();
        for (Int32 test = 0; test < testCount; test++)
            Unsafe2DimArrayAccess(a2Dim);
        Console.WriteLine("{0}: Unsafe2DimArrayAccess", sw.Elapsed);
        Console.ReadLine();
    }

    private static Int32 Safe2DimArrayAccess(Int32[,] a) {
        Int32 sum = 0;
        for (Int32 x = 0; x < c_numElements; x++) {
            for (Int32 y = 0; y < c_numElements; y++) {
```

продолжение ➤

```

        sum += a[x, y];
    }
}
return sum;
}

private static Int32 SafeJaggedArrayAccess(Int32[][] a) {
    Int32 sum = 0;
    for (Int32 x = 0; x < c_numElements; x++) {
        for (Int32 y = 0; y < c_numElements; y++) {
            sum += a[x][y];
        }
    }
    return sum;
}

private static unsafe Int32 Unsafe2DimArrayAccess(Int32[,] a) {
    Int32 sum = 0;
    fixed (Int32* pi = a) {
        for (Int32 x = 0; x < c_numElements; x++) {
            Int32 baseOfDim = x * c_numElements;
            for (Int32 y = 0; y < c_numElements; y++) {
                sum += pi[baseOfDim + y];
            }
        }
    }
    return sum;
}
}

```

Метод `Unsafe2DimArrayAccess` имеет модификатор `unsafe`, который необходим для инструкции `fixed` языка C#. При вызове компилятора следует установить переключатель `/unsafe` или флажок **Allow Unsafe Code** на вкладке **Build** окна свойств проекта в программе Microsoft Visual Studio.

Вот какой результат я получил после выполнения этой программы:

```

00:00:02.0017692: Safe2DimArrayAccess
00:00:01.5197844: SafeJaggedArrayAccess
00:00:01.7343436: Unsafe2DimArrayAccess

```

Как видите, медленней всего работает безопасный способ. Доступ к вложенным массивам занимает чуть меньше времени, но следует помнить, что вложенные массивы создаются медленнее многомерных, так как при этом для каждого измерения в куче формируется новый объект, что требует регулярного внимания со стороны сборщика мусора. Соответственно, вам придется делать выбор. Если требуется создавать много массивов с нечастым обращением к их элементам, эффективней выбрать многомерные массивы. В случае же когда массив создается всего один раз, зато доступ к его элементам осуществляется

часто, лучшую производительность обеспечит вложенный массив. Разумеется, в основном приходится иметь дело со вторым вариантом.

Вариант доступа к массиву с применением небезопасного кода является самым быстрым. Время доступа почти на порядок меньше, чем в случае с вложенными массивами. При этом вы работаете с реальным двухмерным массивом, экономя ресурсы на создании вложенного массива. На первый взгляд кажется, что самым предпочтительным является «небезопасный» подход, но у него есть три серьезных недостатка:

- ❑ код обращения к элементам массива менее читабелен и более сложен в написании из-за присутствия инструкции `fixed` и вычисления адресов памяти;
- ❑ ошибка в расчетах может привести к перезаписи памяти, не принадлежащей массиву, в результате возможны разрушение памяти, нарушение безопасности типов и потенциальные бреши в системе безопасности;
- ❑ из-за высокой вероятности проблем CLR запрещает работу небезопасного кода в недостаточно безопасных средах (таких как Microsoft Silverlight).

Небезопасный доступ к массивам и массивы фиксированного размера

Небезопасный доступ к массиву является крайне мощным средством, так как именно такой доступ дает возможность работать:

- ❑ с элементами управляемого массива, расположенными в куче (как показано в предыдущем разделе);
- ❑ с элементами массива, расположенными в неуправляемой куче (пример `SecureString` из главы 14 демонстрирует небезопасный метод доступа к массиву, возвращаемому методом `SecureStringToCoTaskMemUnicode` класса `System.Runtime.InteropServices.Marshal`);
- ❑ с элементами массива, расположенными в стеке потока.

Если вопрос производительности для вас крайне важен, управляемый массив можно вместо кучи поместить в стек потока. Для этого вам потребуется инструкция `stackalloc` языка C# (ее принцип действия напоминает функцию `alloca` языка C). Она позволяет создавать одномерные массивы элементов значимого типа с нулевой нижней границей. При этом значимый тип не должен содержать никаких полей ссылочного типа. По сути, вы выделяете блок памяти, которым можно управлять при помощи небезопасных указателей, поэтому адрес этого буфера нельзя передавать большинству FCL-методов. Выделенная в стеке память (массив) автоматически освобождается после завершения метода. Именно за счет этого и достигается выигрыш в производительности. При этом для компилятора C# должен быть задан параметр `/unsafe`.

Рассмотрим пример использования инструкции `stackalloc` в методе `StackallocDemo`:

```
using System;
```

```
public static class Program {
    public static void Main() {
        StackallocDemo();
        InlineArrayDemo();
    }

    private static void StackallocDemo() {
        unsafe {
            const Int32 width = 20;
            Char* pc = stackalloc Char[width]; // В стеке выделяется
                                                // память под массив

            String s = "Jeffrey Richter";      // 15 символов

            for (Int32 index = 0; index < width; index++) {
                pc[width - index - 1] =
                    (index < s.Length) ? s[index] : '.';
            }

            // Следующая инструкция выводит на экран ".....rethciR yerffedJ"
            Console.WriteLine(new String(pc, 0, width));
        }
    }

    private static void InlineArrayDemo() {
        unsafe {
            CharArray ca; // В стеке выделяется память под массив
            Int32 widthInBytes = sizeof(CharArray);
            Int32 width = widthInBytes / 2;

            String s = "Jeffrey Richter";      // 15 символов

            for (Int32 index = 0; index < width; index++) {
                ca.Characters[width - index - 1] =
                    (index < s.Length) ? s[index] : '.';
            }

            // Следующая инструкция выводит на экран ".....rethciR yerffedJ"
            Console.WriteLine(new String(ca.Characters, 0, width));
        }
    }
}
```

```
internal unsafe struct CharArray {  
    // Этот массив встраивается в структуру  
    public fixed Char Characters[20];  
}
```

Так как массивы относятся к ссылочным типам, поле массива, определенное в структуре, является указателем или ссылкой на этот массив; при этом сам он располагается вне памяти структуры. Впрочем, существует возможность встроить массив непосредственно в структуру. Вы это видели в показанном коде на примере структуры CharArray. При этом должны соблюдаться следующие условия:

- ❑ тип должен быть структурой (значимым типом), встраивать массивы в класс (ссылочный тип) нельзя;
- ❑ поле или структура, в которой оно определено, должно быть помечено модификатором `unsafe`;
- ❑ поле массива должен быть помечено модификатором `fixed`;
- ❑ массив должен быть одномерным и с нулевой нижней границей;
- ❑ элементы массива могут принадлежать только к типам: `Boolean`, `Char`, `SByte`, `Byte`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single` и `Double`.

Встроенные массивы обычно применяются в сценариях, работающих с небезопасным кодом, в котором неуправляемая структура данных также содержит встроенный массив. Впрочем, никто не запрещает использовать их и в других случаях, например, как показано ранее в методе `InlineArrayDemo`, который по-своему решает ту же задачу, что и метод `StackallocDemo`.

Глава 17. Делегаты

В этой главе рассказывается о чрезвычайно полезном механизме, который используется уже много лет и называется функциями обратного вызова. В Microsoft .NET Framework этот механизм поддерживается при помощи *делегатов* (delegates). В отличие от других платформ, например неуправляемого языка C++, здесь делегаты имеют более развернутую функциональность. Например, они обеспечивают безопасность типов при выполнении обратного вызова (способствуя решению одной из важнейших задач CLR). Кроме того, именно они позволяют поддерживать последовательный вызов нескольких методов и вызывать как статические, так и экземплярные методы.

Знакомство с делегатами

Функция `qsort` исполняющей среды C сортирует элементы массивов при помощи функции обратного вызова. В Microsoft Windows механизм обратного вызова обеспечивает выполнение оконных процедур, процедур перехвата, асинхронного вызова процедур и др. В .NET Framework методы обратного вызова также имеют многочисленные применения. К примеру, зарегистрировав такой метод, вы начинаете получать различные уведомления: о необработанных исключениях, изменении состояния окон, выборе пунктов меню, изменениях файловой системы и завершении асинхронных операций.

В неуправляемом языке C/C++ адрес функции — не более чем адрес в памяти, не несущий дополнительной информации. С его помощью вы не узнаете ни количество ожидаемых функцией параметров, ни их тип, ни тип возвращаемого функцией значения, ни правила ее вызова. Другими словами, в данном случае функции обратного вызова не обеспечивают безопасность типов (хотя их и отличает высокая скорость выполнения).

В .NET Framework функции обратного вызова играют не менее важную роль, чем при неуправляемом программировании для Windows. Однако данная платформа снабжена делегатами — механизмом, обеспечивающим безопасность типов. Рассмотрим процесс их объявления, создания и использования:

```
using System;  
using System.Windows.Forms;  
using System.IO;
```

```
// Объявление делегата; экземпляр ссылается на метод  
// с параметром типа Int32, возвращающий значение void  
internal delegate void Feedback(Int32 value);
```

```
public sealed class Program {
    public static void Main() {
        StaticDelegateDemo();
        InstanceDelegateDemo();
        ChainDelegateDemo1(new Program());
        ChainDelegateDemo2(new Program());
    }

    private static void StaticDelegateDemo() {
        Console.WriteLine("----- Static Delegate Demo -----");
        Counter(1, 3, null);
        Counter(1, 3, new Feedback(Program.FeedbackToConsole));
        Counter(1, 3, new Feedback(FeedbackToMsgBox)); // Префикс "Program."
                                                         // не обязателен
        Console.WriteLine();
    }

    private static void InstanceDelegateDemo() {
        Console.WriteLine("----- Instance Delegate Demo -----");
        Program p = new Program();
        Counter(1, 3, new Feedback(p.FeedbackToFile));
        Console.WriteLine();
    }

    private static void ChainDelegateDemo1(Program p) {
        Console.WriteLine("----- Chain Delegate Demo 1 -----");
        Feedback fb1 = new Feedback(FeedbackToConsole);
        Feedback fb2 = new Feedback(FeedbackToMsgBox);
        Feedback fb3 = new Feedback(p.FeedbackToFile);
        Feedback fbChain = null;
        fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
        fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
        fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
        Counter(1, 2, fbChain);
        Console.WriteLine();
        fbChain = (Feedback)
        Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
        Counter(1, 2, fbChain);
    }

    private static void ChainDelegateDemo2(Program p) {
        Console.WriteLine("----- Chain Delegate Demo 2 -----");
        Feedback fb1 = new Feedback(FeedbackToConsole);
        Feedback fb2 = new Feedback(FeedbackToMsgBox);
        Feedback fb3 = new Feedback(p.FeedbackToFile);
        Feedback fbChain = null;
```



```

    fbChain += fb1;
    fbChain += fb2;
    fbChain += fb3;
    Counter(1, 2, fbChain);
    Console.WriteLine();
    fbChain -= new Feedback(FeedbackToMsgBox);
    Counter(1, 2, fbChain);
}

private static void Counter(Int32 from, Int32 to, Feedback fb) {
    for (Int32 val = from; val <= to; val++) {
        // Если указаны методы обратного вызова, вызываем их
        if (fb != null)
            fb(val);
    }
}

private static void FeedbackToConsole(Int32 value) {
    Console.WriteLine("Item=" + value);
}

private static void FeedbackToMsgBox(Int32 value) {
    MessageBox.Show("Item=" + value);
}

private void FeedbackToFile(Int32 value) {
    StreamWriter sw = new StreamWriter("Status", true);
    sw.WriteLine("Item=" + value);
    sw.Close();
}
}

```

Рассмотрим этот код более подробно. Прежде всего следует обратить внимание на объявление внутреннего делегата `Feedback`. Он задает сигнатуру метода обратного вызова. Данный делегат определяет метод, принимающий один параметр типа `Int32` и возвращающий значение `void`. Он напоминает ключевое слово `typedef` из C/C++, которое предоставляет адрес функции.

Класс `Program` определяет закрытый статический метод `Counter`. Он перебирает целые числа в диапазоне, заданном аргументами `from` и `to`. Также он принимает параметр `fb`, который является ссылкой на делегат `Feedback`. Метод `Counter` просматривает числа в цикле и для каждого из них при условии, что переменная `fb` не равна `null`, выполняет метод обратного вызова (определенный переменной `fb`). При этом методу обратного вызова передается значение обрабатываемого элемента и его номер. Данный метод может быть реализован любым способом, позволяющим обрабатывать элементы так, как это требуется.

Обратный вызов статических методов

Теперь, когда мы разобрали принцип работы метода `Counter`, рассмотрим процедуру использования делегатов для вызова статических методов. Для примера возьмем метод `StaticDelegateDemo` из представленного в предыдущем разделе кода.

Метод `StaticDelegateDemo` вызывает метод `Counter`, передавая в третий параметр `fb` значение `null`. В результате при обработке элементов не задействуется метод обратного вызова.

При втором вызове метода `Counter` методом `StaticDelegateDemo` третьему параметру передается только что созданный делегат `Feedback`. Этот делегат служит оболочкой для другого метода, позволяя выполнить обратный вызов последнего косвенно, через оболочку. В рассматриваемом примере имя статического метода `Program.FeedbackToConsole` передается конструктору `Feedback`, указывая, что именно для него требуется создать оболочку. Возвращенная оператором `new` ссылка передается третьему параметру метода `Counter`, который в процессе выполнения будет вызывать статический метод `FeedbackToConsole`. Последний же просто выводит на консоль строку с названием обрабатываемого элемента.

ПРИМЕЧАНИЕ

Метод `FeedbackToConsole` определен в типе `Program` как закрытый, но при этом может быть вызван методом `Counter`. Так как оба метода определены в пределах одного типа, проблем с безопасностью не возникает. Но даже если бы метод `Counter` был определен в другом типе, это не сказалось бы на работе кода. Другими словами, если код одного типа вызывает посредством делегата закрытый член другого типа, проблем с безопасностью или уровнем доступа не возникает, если делегат создан кодом, имеющим нужный уровень доступа.

Третий вызов метода `Counter` внутри метода `StaticDelegateDemo` отличается от второго тем, что делегат `Feedback` является оболочкой для статического метода `Program.FeedbackToMsgBox`. Именно метод `FeedbackToMsgBox` создает строку, указывающую на обрабатываемый элемент, которая затем выводится в окне в виде сообщения.

В этом примере ничто не нарушает безопасность типов. К примеру, при создании делегата `Feedback` компилятор гарантирует, что сигнатуры методов `FeedbackToConsole` и `FeedbackToMsgBox` типа `Program` совместимы с сигнатурой делегата. Это означает, что оба метода будут принимать один и тот же аргумент (типа `Int32`) и возвращать значение одного и того же типа (`void`). Однако попробуем определить метод `FeedbackToConsole` вот так:

```
private static Boolean FeedbackToConsole(String value) {  
    ...  
}
```

В этом случае компилятор выдаст сообщение об ошибке» (сигнатура метода `FeedbackToConsole` не соответствует типу делегата):

```
error CS0123: No overload for 'FeedbackToConsole' matches delegate 'Feedback'
```

Как C#, так и CLR поддерживают прямую и обратную ковариацию ссылочных типов при привязке метода к делегату. *Ковариация* (covariance) означает, что метод может вернуть тип, производный от типа, возвращаемого делегатом. *Обратная ковариация* (contra-variance) означает, что метод может принимать параметр, который является базовым для типа параметра делегата. Например:

```
delegate Object MyCallback(FileStream s);
```

Определив делегат таким образом, можно получить экземпляр этого делегата, связанный с методом, прототип которого выглядит примерно так:

```
String SomeMethod(Stream s);
```

Здесь тип значения, возвращаемого методом `SomeMethod` (тип `String`), является производным от типа, возвращаемого делегатом (`Object`); такая ковариация разрешена. Тип параметра метода `SomeMethod` (тип `Stream`) является базовым классом для типа параметра делегата (`FileStream`); обратная ковариация тоже разрешена.

Обратите внимание, что прямая и обратная ковариация поддерживаются только для ссылочных типов, но не для значимых типов или значения `void`. К примеру, связать следующий метод с делегатом `MyCallback` невозможно:

```
Int32 SomeOtherMethod(Stream s);
```

Несмотря на то что тип значения, возвращаемого методом `SomeOtherMethod` (то есть `Int32`), является производным от типа значения, возвращаемого методом `MyCallback` (то есть `Object`), ковариация невозможна, потому что `Int32` — это значимый тип. Значимые типы и значение `void` не могут использоваться для прямой и обратной ковариации, потому что их структура памяти меняется, в то время как для ссылочных типов структурой памяти в любом случае остается указатель. К счастью, при попытке выполнить запрещенные действия компилятор возвращает сообщение об ошибке.

Обратный вызов экземплярных методов

Мы рассмотрели процедуру вызова при помощи делегатов статических методов, но они позволяют вызывать также экземплярные методы заданного объекта. Рассмотрим механизм этого вызова на примере метода `InstanceDelegateDemo` из показанного ранее кода.

Обратите внимание, что объект `p` типа `Program` создается внутри метода `InstanceDelegateDemo`. При этом у него отсутствуют экземплярные поля и свойства, поскольку он сконструирован с демонстрационными целями. Когда при вызове метода `Counter` создается делегат `Feedback`, его конструктору передается объект `p.FeedbackToFile`. В результате делегат превращается в оболочку для ссылки на метод `FeedbackToFile`, который является не статическим, а экземплярным методом. Когда метод `Counter` обращается к методу обратного вызова, который задан аргументом `fb`, вызывается экземплярный метод `FeedbackToFile`, а адрес только что созданного объекта `p` передается этому методу в качестве явного аргумента.

Метод `FeedbackToFile` отличается от методов `FeedbackToConsole` и `FeedbackToMsgBox` тем, что открывает файл и дописывает в его конец строку (созданный им файл `Status` находится в папке **AppBase** приложения).

Как видите, делегаты могут служить оболочкой как для статических, так и для экземплярных методов. В последнем случае делегат должен знать, какой экземпляр объекта будет обрабатывать вызываемый им метод. Создав оболочку для экземплярного метода, вы даете коду внутри объекта доступ к различным членам экземпляра объекта. Это означает наличие у объекта состояния, которое может использоваться во время выполнения метода обратного вызова.

Раскрытие тайны делегатов

На первый взгляд работать с делегатами легко. Они определяются при помощи ключевого слова `delegate`, оператор `new` создает экземпляры делегатов, а для обратного вызова служит уже знакомый синтаксис. В последнем случае вместо имени метода указывается ссылающаяся на делегат переменная.

На самом деле все обстоит несколько сложнее, чем демонстрируют приведенные примеры. Пользователи просто не осознают всей сложности процесса благодаря работе компиляторов и CLR. Однако в этом разделе рассматриваются все тонкости реализации делегатов, так как это поможет нам понять принцип их работы и научиться применять их эффективно и рационально. Рассматриваются также дополнительные возможности делегатов.

Внимательно посмотрите на следующую строку:

```
internal delegate void Feedback(Int32 value);
```

Она заставляет компилятор создать полное определение класса, которое выглядит примерно так:

```
internal class Feedback : System.MulticastDelegate {  
    // Конструктор  
    public Feedback(Object object, IntPtr method);  
  
    // Метод, прототип которого задан в исходном тексте  
    public virtual void Invoke(Int32 value);
```

```
// Методы, обеспечивающие асинхронный обратный вызов
public virtual IAsyncResult BeginInvoke(Int32 value,
    AsyncCallback callback, Object object);
public virtual void EndInvoke(IAsyncResult result);
}
```

Определенному компилятором классу принадлежат четыре метода: конструктор, а также методы `Invoke`, `BeginInvoke` и `EndInvoke`. В этой главы мы в основном будем рассматривать конструктор и метод `Invoke`. Информацию о методах `BeginInvoke` и `EndInvoke` вы найдете в главе 27.

Исследовав итоговую сборку при помощи утилиты `ILDasm.exe`, можно убедиться, что компилятор действительно автоматически сгенерировал этот класс (рис 17.1).

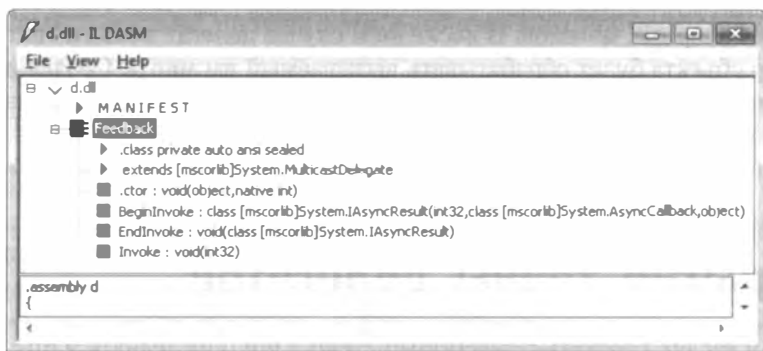


Рис. 17.1. Сгенерированные компилятором метаданные делегата

В этом примере компилятор определил класс `Feedback`, производный от типа `System.MulticastDelegate` из библиотеки классов `Framework Class Library` (все типы делегатов являются потомками `MulticastDelegate`).

ВНИМАНИЕ

Класс `System.MulticastDelegate` является производным от класса `System.Delegate`, который, в свою очередь, наследует от класса `System.Object`. Два класса делегатов появились исторически, в то время как в FCL предполагался только один. Вам следует помнить об обоих классах, так как даже если выбрать в качестве базового класс `MulticastDelegate`, все равно иногда приходится работать с делегатами, использующими методы класса `Delegate`. Скажем, именно этому классу принадлежат статические методы `Combine` и `Remove` (о том, зачем они нужны, мы поговорим чуть позже). Сигнатуры этих методов указывают, что они принимают параметры класса `Delegate`. Так как тип вашего делегата является производным от класса `MulticastDelegate`, для которого базовым является класс `Delegate`, методом можно передавать экземпляры типа делегата.

Это закрытый класс, так как делегат был объявлен с модификатором `internal`. Если объявить его с модификатором `public`, сгенерированный компилятором класс `Feedback` будет открытым. Следует помнить, что делегаты можно определять как внутри класса (вложенные в другой класс), так и в глобальной области видимости. По сути, так как делегаты являются классами, их можно определить в любом месте, где может быть определен класс.

Любые типы делегатов — это потомки класса `MulticastDelegate`, от которого они наследуют все поля, свойства и методы. Три самых важных закрытых поля описаны в табл. 17.1.

Таблица 17.1. Важнейшие закрытые поля класса `MulticastDelegate`

Поле	Тип	Описание
<code>_target</code>	<code>System.Object</code>	Если делегат является оболочкой статического метода, это поле содержит значение <code>null</code> . Если делегат является оболочкой экземплярного метода, поле ссылается на объект, с которым будет работать метод обратного вызова. Другими словами, поле указывает на значение, которое следует передать параметру <code>this</code> экземплярного метода
<code>_methodPtr</code>	<code>System.IntPtr</code>	Внутреннее целочисленное значение, используемое CLR для идентификации метода обратного вызова
<code>_invocationList</code>	<code>System.Object</code>	Это поле обычно имеет значение <code>null</code> . Оно может ссылаться на массив делегатов при построении из них цепочки (об этом мы поговорим чуть позже)

Обратите внимание, что конструктор всех делегатов принимает два параметра: ссылку на объект и целое число, ссылающееся на метод обратного вызова. Но в тексте исходного кода туда передаются такие значения, как `Program.FeedbackToConsole` или `p.FeedbackToFile`. Думаю, интуиция уже подсказала вам, что этот код компилировать нельзя!

Тем не менее компилятор знает о том, что создается делегат, и, проанализировав код, определяет объект и метод, на которые мы ссылаемся. Ссылка на объект передается в параметре `object` конструктора. Специальное значение `IntPtr` (получаемое из маркеров метаданных `MethodDef` или `MemberRef`), идентифицирующее метод, передается в параметре `method`. В случае статических методов параметр `object` передает значение `null`. Внутри конструктора значения этих двух аргументов сохраняются в закрытых полях `_target` и `_methodPtr` соответственно. Кроме того, конструктор присваивает значение `null` полю `_invocationList`. О назначении этого поля мы подробно поговорим в разделе, посвященном цепочкам делегатов.

Таким образом, любой делегат — это всего лишь оболочка метода и обрабатываемого этим методом объекта. Поэтому в следующих строчках кода

переменные fbStatic и fbInstance ссылаются на два разных объекта Feedback, инициализированных, как показано на рис. 17.2:

```
Feedback fbStatic = new Feedback(Program.FeedbackToConsole);
Feedback fbInstance = new Feedback(new Program().FeedbackToFile);
```

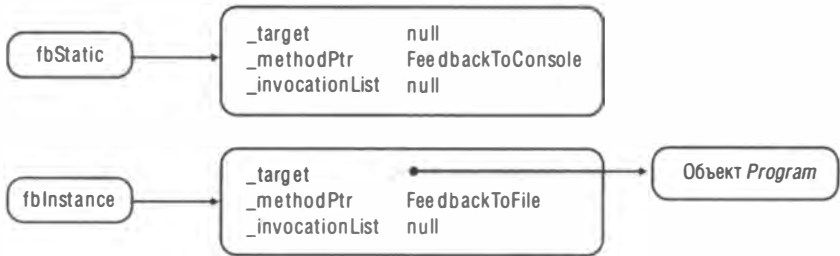


Рис. 17.2. Верхняя переменная ссылается на делегат статического метода, нижняя — на делегат экземплярного метода

В классе Delegate определены два открытых, предназначенных только для чтения экземплярных свойства: Target и Method. Их значения можно запросить при наличии ссылки на делегат. Свойство Target возвращает ссылку на объект, с которым работает метод обратного вызова. По сути, это свойство возвращает значение закрытого поля _target. Если делегат является оболочкой для статического метода, свойство Target возвращает значение null. Свойство Method возвращает ссылку на объект System.Reflection.MethodInfo, идентифицирующий метод обратного вызова. Внутренний механизм свойства Method преобразует значение закрытого поля _methodPtr в объект MethodInfo и возвращает этот объект.

Эту информацию можно использовать по-разному. К примеру, вы можете проверить, не ссылается ли объект на экземплярный метод заданного типа:

```
Boolean DelegateRefersToInstanceMethodOfTypes(
    MulticastDelegate d, Type type) {
    return((d.Target != null) && d.Target.GetType() == type);
}
```

Можно также написать код, проверяющий, не носит ли метод обратного вызова определенного имени (к примеру, FeedbackToMsgBox):

```
Boolean DelegateRefersToMethodOfName(
    MulticastDelegate d, String methodName) {
    return(d.Method.Name == methodName);
}
```

Существует множество вариантов использования этих свойств.

Теперь, когда вы познакомились с процессом создания делегатов и их внутренней структурой, поговорим о методах обратного вызова. Рассмотрим еще раз код метода Counter:

```
private static void Counter(Int32 from, Int32 to, Feedback fb) {
    for (Int32 val = from; val <= to; val++) {
        // Если указаны методы обратного вызова, вызываем их
        if (fb != null)
            fb(val);
    }
}
```

Обратите внимание на строку под комментарием. Инструкция `if` сначала проверяет, не содержит ли переменная `fb` значения `null`. Если проверка пройдена, обращаемся к методу обратного вызова. Такая проверка необходима потому, что `fb` — это всего лишь переменная, ссылающаяся на делегат `Feedback`; она может иметь, в том числе, значение `null`. Может показаться, что происходит вызов функции `fb`, которой передается один параметр (`val`). Но у нас нет функции с таким именем. И компилятор генерирует код вызова метода `Invoke` делегата, так как он знает, что переменная `fb` ссылается на делегат. Другими словами:

```
fb(val);
```

Обнаружив эту строку, компилятор генерирует такой же код, как и для строки:

```
fb.Invoke(val);
```

Воспользовавшись утилитой `ILDasm.exe` для исследования кода метода `Counter`, можно убедиться, что компилятор генерирует код, вызывающий метод `Invoke`. Далее показан IL-код метода `Counter`. Команда в строке `IL_0009` является вызовом метода `Invoke` объекта `Feedback`.

```
.method private hidebysig static void Counter(int32 from,
                                              int32 'to',
                                              class Feedback fb) cil managed
{
    // Code size 23 (0x17)
    .maxstack 2
    .locals init (int32 V_0)
    IL_0000: ldarg.0
    IL_0001: stloc.0
    IL_0002: br.s IL_0012
    IL_0004: ldarg.2
    IL_0005: brfalse.s IL_000e
    IL_0007: ldarg.2
    IL_0008: ldloc.0
    IL_0009: callvirt instance void Feedback::Invoke(int32)
```

продолжение ➤


```

IL_000e: ldloc.0
IL_000f: ldc.i4.1
IL_0010: add
IL_0011: stloc.0
IL_0012: ldloc.0
IL_0013: ldarg.1
IL_0014: ble.s IL_0004
IL_0016: ret
} // end of method Program::Counter

```

В принципе, можно отредактировать метод Counter, заставив его в явном виде вызывать метод Invoke:

```

private static void Counter(Int32 from, Int32 to, Feedback fb) {
    for (Int32 val = from; val <= to; val++) {
        // Если указаны методы обратного вызова, вызываем их
        if (fb != null)
            fb.Invoke(val);
    }
}

```

Надеюсь, вы помните, что компилятор определяет метод Invoke при определении класса Feedback. Вызывая этот метод, он использует закрытые поля `_target` и `_methodPtr` для вызова желаемого метода на заданном объекте. Обратите внимание, что сигнатура метода Invoke совпадает с сигнатурой делегата, ведь и делегат Feedback, и метод Invoke принимают один параметр типа Int32 и возвращают значение void.

Обратный вызов нескольких методов (цепочки делегатов)

Делегаты полезны сами по себе, но еще более полезными их делает механизм цепочек. *Цепочкой* (chaining) называется коллекция делегатов, дающая возможность вызывать все методы, представленные этими делегатами. Чтобы понять, как работает цепочка, вернитесь к коду в начале этой главы и найдите там метод ChainDelegateDemo1. В этом методе после инструкции Console.WriteLine создаются три делегата, на которые ссылаются переменные fb1, fb2 и fb3 соответственно (рис. 17.3).

Ссылочная переменная на делегат Feedback, которая называется fbChain, должна ссылаться на цепочку, или набор делегатов, служащих оболочками для методов обратного вызова. Присвоение переменной fbChain начального значения null указывает на отсутствие методов обратного вызова. Открытый статический метод Combine класса Delegate добавляет в цепочку делегаты:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
```

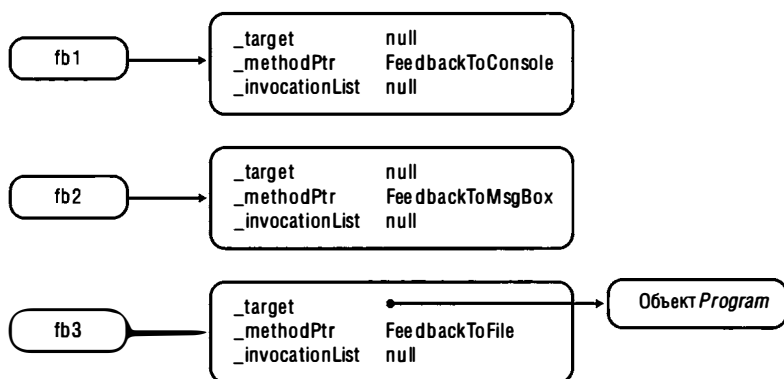


Рис. 17.3. Начальное состояние делегатов, на которые ссылаются переменные fb1, fb2 и fb3

При выполнении этой строки метод `Combine` видит, что мы пытаемся объединить значение `null` с переменной `fb1`. В итоге он возвращает значение в переменную `fb1`, а затем заставляет переменную `fbChain` сослаться на делегат, на который уже ссылается переменная `fb1`. Эта схема демонстрируется на рис. 17.4.

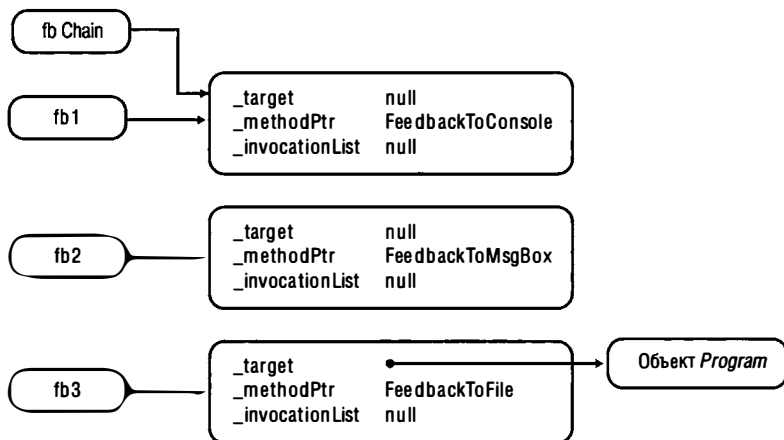


Рис. 17.4. Состояние делегатов после добавления в цепочку нового члена

Чтобы добавить в цепочку еще один делегат, снова воспользуемся методом `Combine`:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
```

Метод `Combine` видит, что переменная `fbChain` уже ссылается на делегат, поэтому он создает нового делегата, который присваивает своим закрытым

полям `_target` и `_methodPtr` некоторые значения. В данном случае они не важны, но важно, что поле `_invocationList` инициализируется ссылкой на массив делегатов. Первому элементу массива (с индексом 0) присваивается ссылка на делегат, служащий оболочкой метода `FeedbackToConsole` (именно на этот делегат ссылается переменная `fbChain`). Второму элементу массива (с индексом 1) присваивается ссылка на делегат, служащий оболочкой метода `FeedbackToMsgBox` (на этот делегат ссылается переменная `fb2`). Напоследок переменной `fbChain` присваивается ссылка на вновь созданный делегат (рис. 17.5).

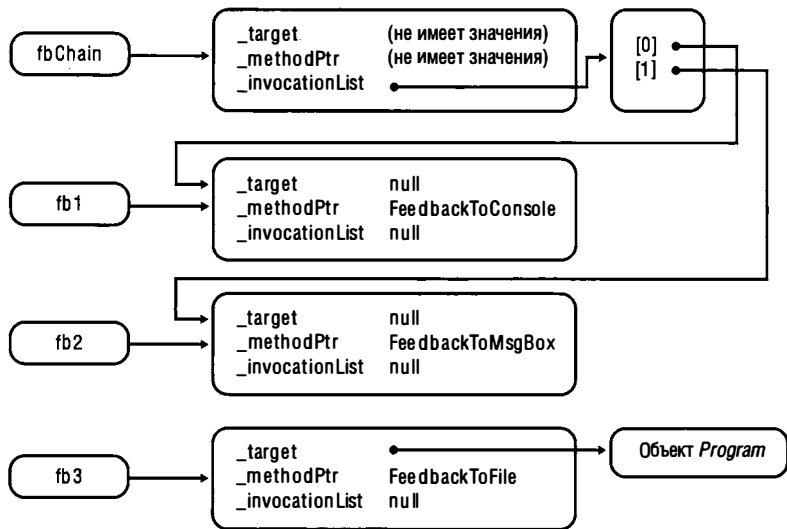


Рис. 17.5. Делегаты после вставки в цепочку второго члена

Для добавления в цепочку третьего делегата снова вызывается метод `Combine`:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
```

И снова, видя, что переменная `fbChain` уже ссылается на делегат, метод создаст очередного делегата, как показано на рис. 17.6. Как и в предыдущих случаях, новый делегат присваивает начальные значения своим закрытым полям `_target` и `_methodPtr`, в то время как поле `_invocationList` инициализируется ссылкой на массив делегатов. Первому и второму элементам массива (с индексами 0 и 1) присваиваются ссылки на те же делегаты, на которые ссылался предыдущий делегат. Третий элемент массива (с индексом 2) становится ссылкой на делегат, служащий оболочкой метода `FeedbackToFile` (именно на этот делегат ссылается переменная `fb3`). Наконец, переменной `fbChain` присваивается ссылка на вновь созданный делегат. При этом ранее созданный делегат и массив, на который ссылается его поле `_invocationList`, теперь подлежат обработке механизмом сборки мусора.

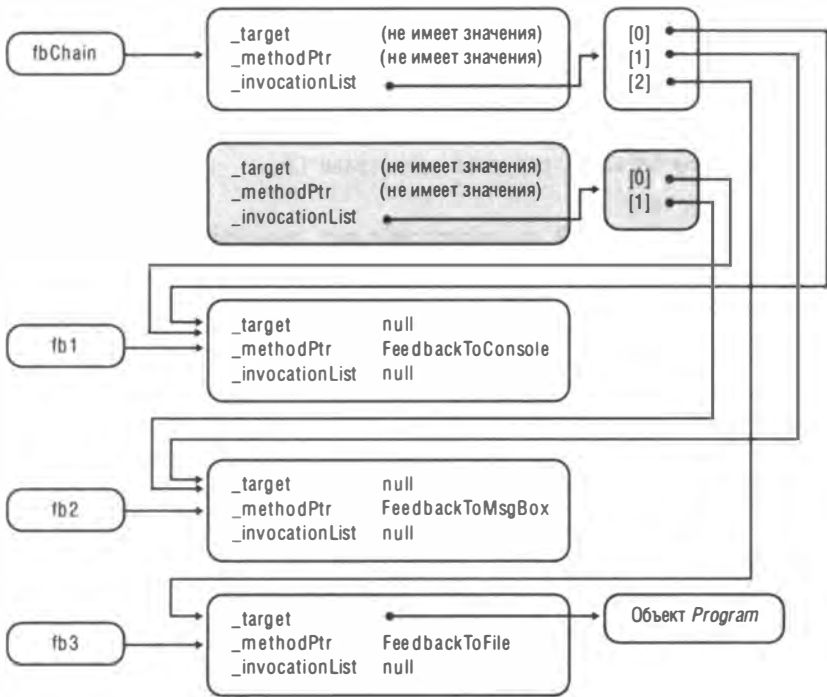


Рис. 17.6. Окончательный вид цепочки делегатов

После выполнения кода, создающего цепочку, переменная fbChain передается методу Counter:

```
Counter(1, 2, fbChain);
```

Метод Counter содержит код неявного вызова метода Invoke для делегата Feedback. Впрочем, об этом мы уже говорили. Когда метод Invoke вызывается для делегата, ссылающегося на переменную fbChain, этот делегат обнаруживает, что значение поля _invocationList отлично от null. Это приводит к выполнению цикла, перебирающего все элементы массива, вызывая для них метод, оболочкой которого служит указанный делегат. В нашем примере методы вызываются в следующей последовательности: FeedbackToConsole, FeedbackToMsgBox и, наконец, FeedbackToFile.

Реализация метода Invoke класса Feedback выглядит примерно так (в псевдокоде):

```
public void Invoke(Int32 value) {
    Delegate[] delegateSet = _invocationList as Delegate[];
    if (delegateSet != null) {
        // Этот массив указывает на делегаты, которые следует вызвать
        foreach (Feedback d in delegateSet)
            d(value); // Вызов каждого делегата
    }
}
```

продолжение ➤

```

    } else {
        // Этот делегат определяет используемый метод обратного вызова
        // Применяем метод обратного вызова к указанному объекту
        _methodPtr.Invoke(_target, value);
        // Строка выше – приблизительная копия реального кода
        // Происходящее не иллюстрируется средствами C#
    }
}

```

Для удаления делегатов из цепочки применяется статический метод `Remove` объекта `Delegate`. Эта процедура демонстрируется в конце кода метода `ChainDelegateDemo1`:

```

fbChain = (Feedback) Delegate.Remove(
    fbChain, new Feedback(FeedbackToMsgBox));

```

Метод `Remove` сканирует массив делегатов (с конца и до члена с нулевым индексом), управляемый делегатом, на который ссылается первый параметр (в нашем примере это `fbChain`). Он ищет делегат, поля `_target` и `_methodPtr` которого совпадают с соответствующими полями второго аргумента (в нашем примере это новый делегат `Feedback`). При обнаружении совпадения, если в массиве осталось более одного элемента, создается новый делегат. Создается массив `_invocationList`, который инициализируется ссылкой на все элементы исходного массива за исключением удаляемого. После этого возвращается ссылка на новый делегат. При удалении последнего элемента цепочки метод `Remove` возвращает значение `null`. Следует помнить, что метод `Remove` за один раз удаляет всего одного делегата, а не все элементы с указанными значениями полей `_target` и `_methodPtr`.

Ранее мы также рассматривали делегат `Feedback` возвращающий значение типа `void`. Однако этот делегат можно было определить и так:

```

public delegate Int32 Feedback(Int32 value);

```

В этом случае псевдокод метода `Invoke` выглядел бы следующим образом:

```

public Int32 Invoke(Int32 value) {
    Int32 result;
    Delegate[] delegateSet = _invocationList as Delegate[];
    if (delegateSet != null) {
        // Массив указывает на делегаты, которые нужно вызвать
        foreach (Feedback d in delegateSet)
            result = d(value); // Вызов делегата
    } else {
        // Этот делегат определяет используемый метод обратного вызова
        // Применяем метод обратного вызова к указанному объекту
        result = _methodPtr.Invoke(_target, value);
        // Строка выше – приблизительная копия реального кода
        // Происходящее не иллюстрируется средствами C#
    }
    return result;
}

```

По мере вызова отдельных делегатов возвращаемое значение сохраняется в переменной `result`. После завершения цикла в этой переменной оказывается только результат вызова последнего делегата (предыдущие возвращаемые значения отбрасываются); именно это значение возвращается коду, вызвавшему метод `Invoke`.

Поддержка цепочек делегатов в C#

Компилятор C# облегчает жизнь разработчикам, автоматически обеспечивая перегрузку операторов `+=` и `-=` для экземпляров делегатов. Эти операторы вызывают методы `Delegate.Combine` и `Delegate.Remove` соответственно. Они упрощают построение цепочек делегатов. В результате компиляции методов `ChainDelegateDemo1` и `ChainDelegateDemo2` (см. пример в начале главы) получается идентичный IL-code. Единственная разница в том, что благодаря операторам `+=` и `-=` исходный код метода `ChainDelegateDemo2` получается проще.

Для доказательства идентичности скомпилируйте IL-код обоих методов и изучите его при помощи утилиты `ILDasm.exe`. Вы убедитесь, что компилятор C# действительно заменяет все операторы `+=` и `-=` вызовами статических методов `Combine` и `Remove` типа `Delegate` соответственно.

Дополнительные средства управления цепочками делегатов

Итак, вы научились создавать цепочки делегатов и вызывать все их компоненты. Последняя возможность реализуется благодаря наличию в методе `Invoke` кода, просматривающего все элементы массива делегатов. Этого простого алгоритма хватает для большинства сценариев, но у него есть ряд ограничений. К примеру, сохраняется только последнее из значений, возвращаемых методами обратного вызова. Получить все остальные значения нельзя. И это — не единственное ограничение. Скажем, в ситуации, когда один из делегатов в цепочке становится причиной исключения или оказывается заблокированным, работа цепочки останавливается. Так что данный алгоритм крайне ненадежен.

В качестве альтернативы можно воспользоваться экземплярным методом `GetInvocationList` класса `MulticastDelegate`. Этот метод позволяет в явном виде вызвать любой из делегатов в цепочке:

```
public abstract class MulticastDelegate : Delegate {  
    // Создает массив, каждый элемент которого ссылается  
    // на делегат в цепочке  
    public sealed override Delegate[] GetInvocationList();  
}
```

Метод `GetInvocationList` работает с объектами, наследующими от класса `MulticastDelegate`. Он возвращает массив ссылок, каждая из которых указывает на какой-то делегат в цепочке. По сути, этот метод создает массив и инициа-

лизирует его элементы ссылками на соответствующие делегаты; в конце возвращается ссылка на этот массив. Если поле `_invocationList` содержит значение `null`, возвращаемый массив будет содержать всего один элемент, ссылающийся на единственный делегат в цепочке — экземпляр самого делегата.

Написать алгоритм, в явном виде вызывающий каждый элемент массива, несложно:

```
using System;
using System.Text;

// Определение компонента Light
internal sealed class Light {
    // Метод возвращает состояние объекта Light
    public String SwitchPosition() {
        return "The light is off";
    }
}

// Определение компонента Fan
internal sealed class Fan {
    // Метод возвращает состояние объекта Fan
    public String Speed() {
        throw new InvalidOperationException("The fan broke due to overheating");
    }
}

// Определение компонента Speaker
internal sealed class Speaker {
    // Метод возвращает состояние объекта Speaker
    public String Volume() {
        return "The volume is loud";
    }
}

public sealed class Program {

    // Определение делегатов, позволяющих запрашивать состояние компонентов
    private delegate String GetStatus();
    public static void Main() {

        // Объявление пустой цепочки делегатов
        GetStatus getStatus = null;

        // Создание трех компонентов и добавление в цепочку
        // методов проверки их состояния
        getStatus += new GetStatus(new Light().SwitchPosition);
        getStatus += new GetStatus(new Fan().Speed);
        getStatus += new GetStatus(new Speaker().Volume);
```

```
// Сводный отчет о состоянии трех компонентов
Console.WriteLine(GetComponentStatusReport(getStatus));
}

// Метод запрашивает компоненты и возвращает их состояние
private static String GetComponentStatusReport(GetStatus status) {
    // Если цепочка пуста, действий не нужно
    if (status == null) return null;
    // Построение отчета о состоянии
    StringBuilder report = new StringBuilder();
    // Создание массива из делегатов цепочки
    Delegate[] arrayOfDelegates = status.GetInvocationList();
    // Циклическая обработка делегатов массива
    foreach (GetStatus getStatus in arrayOfDelegates) {
        try {

            // Получение статуса компонента и добавление его в отчет
            report.AppendFormat("{0}{1}{1}", getStatus(), Environment.NewLine);
        }
        catch (InvalidOperationException e) {

            // В отчете генерируется запись об ошибке для этого компонента
            Object component = getStatus.Target;
            report.AppendFormat(
                "Failed to get status from {1}{2}{0} Error: {3}{0}{0}",
                Environment.NewLine,
                ((component == null) ? "" : component.GetType() + "."),
                getStatus.Method.Name,
                e.Message);
        }
    }

    // Возвращение сводного отчета вызывающему коду
    return report.ToString();
}
}
```

Вот результат работы этого кода:

```
The light is off
Failed to get status from Fan.Speed
Error: The fan broke due to overheating
The volume is loud
```

Обобщенные делегаты

Много лет назад, когда среда .NET Framework только начинала разрабатываться, в Microsoft ввели понятие делегатов. По мере добавления в FCL классов появля-

лись и новые типы делегатов. Со временем их накопилось изрядное количество. Только в библиотеке `mscorlib.dll` их около 50. Вот некоторые из них:

```
public delegate void TryCode(Object userData);
public delegate void WaitCallback(Object state);
public delegate void TimerCallback(Object state);
public delegate void ContextCallback(Object state);
public delegate void SendOrPostCallback(Object state);
public delegate void ParameterizedThreadStart(Object obj);
```

Вы заметили, что объединяет все перечисленные делегаты? На самом деле, они одинаковы: переменная любого из этих делегатов должна ссылаться на метод, принимающий параметры типа `Object` и возвращающий значение `void`. Соответственно, нам не нужен весь этот набор делегатов. Вполне можно обойтись всего одним.

Так как современная версия `.NET Framework` поддерживает обобщения, нам на самом деле нужно всего лишь несколько обобщенных делегатов (определенных в пространстве имен `System`), представляющих методы, которые могут принимать до 16 аргументов:

```
public delegate void Action(); // Этот делегат не обобщенный
public delegate void Action<T>(T obj);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate void Action<T1, ..., T16>(T1 arg1, ..., T16 arg16);
```

Итак, в `.NET Framework` имеются 17 делегатов `Action`, от не имеющих аргументов вообще, до имеющих 16 аргументов. Чтобы вызвать метод с большим количеством аргументов, вам потребуется определить собственный делегат, но это уже экзотическая ситуация.

Кроме делегатов `Action` в `.NET Framework` имеется 17 делегатов `Func`, которые обеспечивают возврат значений методами обратного вызова:

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T, TResult>(T arg);
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate TResult Func<T1, ..., T16, TResult>(T1 arg1, ..., T16 arg16);
```

Вместо определения собственных типов делегатов рекомендуется по мере возможности использовать обобщенные делегаты. Ведь это уменьшает количество типов в системе и упрощает код. Однако, если нужно передать аргумент по ссылке, используя ключевые слова `ref` или `out`, может потребоваться определение собственного делегата:

```
delegate void Bar(ref Int32 z);
```

Аналогично нужно действовать в ситуациях, когда требуется, чтобы делегат принял переменное число параметров при помощи ключевого слова `params`, если вы хотите задать значение по умолчанию для аргументов делегата или если требуется ограничить обобщенный тип аргумента делегата, как в следующем примере:

```
delegate void EventHandler<TEventArgs>(Object sender, TEventArgs e)
where TEventArgs : EventArgs;
```

ПРИМЕЧАНИЕ

Типы делегатов `Action` и `Func`, принимающие от 0 до 8 аргументов, определены в библиотеке `mscorlib.dll`, так как методы, оперирующие подобным количеством аргументов, встречаются сплошь и рядом. Делегаты же, принимающие от 9 до 16 аргументов, определены в библиотеке `System.Core.dll`, так как работать с ними приходится довольно редко. На самом деле, определения последних в основном предназначены для внутреннего использования в динамических языках программирования, разработчики же к ним практически не обращаются.

При работе с делегатами, использующими обобщенные аргументы и возвращающими значения, не следует забывать про прямую и обратную ковариацию, так как это расширяет область применения делегатов. Дополнительную информацию по этой теме вы найдете в главе 12.

Упрощенный синтаксис для работы с делегатами

Большинство программистов не любит делегаты из-за сложного синтаксиса. К примеру, рассмотрим строку:

```
button1.Click += new EventHandler(button1_Click);
```

Здесь `button1_Click` — это примерно следующий метод:

```
void button1_Click(Object sender, EventArgs e) {
    // Действия после щелчка на кнопке...
}
```

Первая строка кода как бы регистрирует адрес метода `button1_Click` в кнопке, привязывая вызов метода к щелчку на ней. Большинство программистов считает неразумным создавать делегат `EventHandler` всего лишь для того, чтобы указать на адрес метода `button1_Click`. Однако данный делегат нужен среде CLR, так как он служит оболочкой, гарантирующей вызов метода с соблюдением безопасности типов. Оболочка также поддерживает вызов экземплярных

методов и создание цепочек. Тем не менее программисты не хотят вникать во все эти детали и предпочли бы записать код следующим образом:

```
button1.Click += button1_Click;
```

К счастью, компилятор C# допускает упрощения синтаксиса при работе с делегатами. Однако перед тем как перейти к рассмотрению соответствующих возможностей, следует заметить, что это — не более чем упрощенные пути создания IL-кода, необходимого CLR для нормальной работы с делегатами. Кроме того, следует учитывать, что поддерживается этот упрощенный синтаксис только в C#; прочие компиляторы, скорее всего, его не поддерживают.

Упрощение 1: не создаем объект делегата

Как вы уже видели, C# позволяет указывать имя метода обратного вызова без создания служащего для него оболочки делегата. Вот еще один пример:

```
internal sealed class AClass {
    public static void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem(SomeAsyncTask, 5);
    }
    private static void SomeAsyncTask(Object o) {
        Console.WriteLine(o);
    }
}
```

Статический метод `QueueUserWorkItem` класса `ThreadPool` ожидает ссылку на делегат `WaitCallback`, который, в свою очередь, ссылается на метод `SomeAsyncTask`. Так как компилятор в состоянии догадаться, что именно имеется в виду, можно опустить строки, относящиеся к созданию делегата `WaitCallback`, сделав общий код более легким для чтения и понимания. В процессе компиляции IL-код, генерирующий новый делегат `WaitCallback`, создается автоматически. Именно это и дает нам возможность упростить запись.

Упрощение 2: не определяем метод обратного вызова

В приведенном фрагменте кода метод обратного вызова `SomeAsyncTask` передается методу `QueueUserWorkItem` класса `ThreadPool`. C# позволяет подставить реализацию метода обратного вызова непосредственно в код, а не в сам метод. Скажем, наш код можно записать так:

```
internal sealed class AClass {
    public static void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem( obj => Console.WriteLine(obj), 5);
    }
}
```

Обратите внимание, что первый «аргумент» метода `QueueUserWorkItem` (он выделен полужирным шрифтом) представляет собой фрагмент кода! Формально в C# он называется *лямбда-выражением* (lambda expression) и распознается по наличию оператора `=>`. Лямбда-выражения используются в тех местах, где компилятор ожидает присутствия делегата. Обнаружив лямбда-выражение, компилятор автоматически определяет в классе новый закрытый метод (в нашем примере — `AClass`). Этот метод называется *анонимной функцией* (anonymous function), так как вы обычно не знаете его имени, которое автоматически создается компилятором. Впрочем, никто не мешает исследовать полученный код при помощи утилиты `ILDasm.exe`. Именно она помогла узнать после компиляции написанного фрагмента кода, что методу было присвоено имя `<CallbackWithoutNewingADelegateObject>b__0`, а также, что метод принимает всего один аргумент типа `Object`, возвращая значение типа `void`.

Компилятор выбирает для метода имя, начинающееся с символа `<`, потому что в C# идентификаторы не могут содержать этот символ. Такой подход гарантирует, что программист не сможет случайно выбрать для какого-нибудь из своих методов имя, совпадающее с автоматически созданным компилятором. При этом, если в C# идентификаторы не могут содержать символа `<`, в CLR это разрешено. Несмотря на возможность доступа к методу через отражения путем передачи его имени в виде строки, следует помнить, что компилятор может по-разному генерировать это имя при каждом следующем проходе.

Утилита `ILDasm.exe` позволяет также заметить, что компилятор C# применяет к методу атрибут `System.Runtime.CompilerServices.CompilerGeneratedAttribute`. Это дает инструментам и утилитам возможность понять, что метод создан автоматически, а не написан программистом. В этот сгенерированный компилятором метод и помещается код справа от оператора `=>`.

ПРИМЕЧАНИЕ

При написании лямбда-выражений невозможно применить к сгенерированному компилятором методу пользовательские атрибуты или модификаторы (например, `unsafe`). Впрочем, обычно это не является проблемой, так как созданные компилятором анонимные методы всегда закрыты. Каждый такой метод является статическим или нестатическим в зависимости от того, будет ли он иметь доступ к каким-либо экземплярным членам. Соответственно, применять к этим методам модификаторы `public`, `protected`, `internal`, `virtual`, `sealed`, `override` или `abstract` просто не требуется.

Написанный код компилятор C# переписывает примерно таким образом (комментарии вставлены мною):

```
internal sealed class AClass {  
    // Это закрытое поле создано для кэширования делегата  
    // Преимущество: CallbackWithoutNewingADelegateObject не будет
```

продолжение ➤

```

// создавать новый объект при каждом вызове
// Недостатки: Кэшированные объекты недоступны для сборщика мусора
[CompilerGenerated]
private static WaitCallback <>9__CachedAnonymousMethodDelegate1;

public static void CallbackWithoutNewingADelegateObject() {
    if (<>9__CachedAnonymousMethodDelegate1 == null) {
        // При первом вызове делегат создается и кэшируется
        <>9__CachedAnonymousMethodDelegate1 =
            new WaitCallback(<CallbackWithoutNewingADelegateObject>b__0);
    }
    ThreadPool.QueueUserWorkItem(<>9__CachedAnonymousMethodDelegate1, 5);
}

[CompilerGenerated]
private static void <CallbackWithoutNewingADelegateObject>b__0(
    Object obj) {
    Console.WriteLine(obj);
}
}

```

Лямбда-выражение должно совпадать с сигнатурой делегата `WaitCallback`: возвращать `void` и принимать параметр типа `Object`. Впрочем, я указал имя параметра, просто поместив переменную `obj` слева от оператора `=>`. Расположенный справа от этого оператора метод `Console.WriteLine` действительно возвращает `void`. Если бы расположенное справа выражение не возвращало `void`, сгенерированный компилятором код просто проигнорировал бы возвращенное значение, ведь в противном случае не удалось бы соблюсти требования делегата `WaitCallback`.

Следует помнить, что анонимная функция помечается как `private`; в итоге доступ к методу остается только у кода, определенного внутри этого же типа (хотя отражение позволит узнать о наличии метода). Обратите внимание, что анонимный метод определен как статический. Это связано с отсутствием у кода доступа к каким-либо членам экземпляра (ведь метод `CallbackWithoutNewingADelegateObject` сам по себе статический). Впрочем, код может ссылаться на любые определенные в классе статические поля или методы. Например:

```

internal sealed class AClass {
    private static String sm_name; // Статическое поле
    public static void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem(
            // Код обратного вызова может ссылаться на статические члены
            obj => Console.WriteLine(sm_name + ": " + obj),
            5);
    }
}

```

Не будь метод `CallbackWithoutNewingADelegateObject` статическим, код анонимного метода мог бы содержать ссылки на члены экземпляра. Но даже

при отсутствии таких ссылок компилятор все равно генерирует статический анонимный метод, так как он эффективнее экземплярного метода, потому что не требует дополнительный параметр `this`. Если же в коде анонимного метода наличествуют ссылки на члены экземпляра, компилятор создает нестатический анонимный метод:

```
internal sealed class AClass {
    private String m_name; // Поле экземпляра
    // Метод экземпляра
    public void CallbackWithoutNewingADelegateObject() {
        ThreadPool.QueueUserWorkItem(
            // Код обратного вызова может ссылаться на члены экземпляра
            obj => Console.WriteLine(m_name+ ": " + obj),
            5);
    }
}
```

Имена аргументов, которые следует передать лямбда-выражению, указываются слева от оператора `=>`. При этом следует придерживаться правил, которые мы рассмотрим на примерах:

```
// Если делегат не содержит аргументов, используйте круглые скобки
Func<String> f = () => "Jeff";
```

```
// Для делегатов с одним и более аргументами
// можно в явном виде указать типы
Func<Int32, String> f2 = (Int32 n) => n.ToString();
Func<Int32, Int32, String> f3 =
    (Int32 n1, Int32 n2) => (n1 + n2).ToString();
```

```
// Компилятор может задавать типы для делегатов с одним и более аргументами
Func<Int32, String> f4 = (n) => n.ToString();
Func<Int32, Int32, String> f5 = (n1, n2) => (n1 + n2).ToString();
```

```
// Если аргумент у делегата всего один, круглые скобки можно опустить
Func<Int32, String> f6 = n => n.ToString();
```

```
// Для аргументов ref/out нужно в явном виде указывать ref/out и тип
Bar b = (out Int32 n) => n = 5;
```

Предположим, что в последнем случае делегат `Bar` определен следующим образом:

```
delegate void Bar(out Int32 z);
```

Тело анонимной функции пишется справа от оператора `=>`. Оно обычно состоит из простых или сложных выражений, возвращающих некое значение. В рассмотренном примере это было лямбда-выражение, возвращающее строки всем переменным делегата `Func`. Чаще всего тело анонимной функции

состоит из одной инструкции. К примеру, вызванному методу `ThreadPool.QueueUserWorkItem` было передано лямбда-выражение, что привело к вызову метода `Console.WriteLine` (возвращающего значение типа `void`).

Чтобы вставить в тело функции несколько инструкций, заключите их в фигурные скобки. Если делегат ожидает возвращенного значения, не забудьте инструкцию `return`, как показано в следующем примере:

```
Func<Int32, Int32, String> f7 = (n1, n2) => {
    Int32 sum = n1 + n2; return sum.ToString(); };
```

ВНИМАНИЕ

Хотя это и не кажется очевидным, основная выгода от использования лямбда-выражений состоит в том, что они снижают уровень неопределенности вашего кода. Обычно приходится писать отдельный метод, присваивать ему имя и передавать это имя методу, в котором требуется делегат. Именно имя позволяет ссылаться на фрагмент кода. И если ссылка на один и тот же фрагмент требуется в различных местах программы, создание метода — это самое правильное решение. Если же ссылка на фрагмент кода предполагается всего одна, на помощь приходят лямбда-выражения. Именно они позволяют встраивать фрагменты кода в нужное место, избавляя от необходимости их именования и повышая тем самым продуктивность работы программиста.

ПРИМЕЧАНИЕ

В C# 2.0 впервые появился механизм анонимных методов. Подобно лямбда-выражениям (появившимся в C# 3.0), анонимные методы описывают синтаксис создания анонимных функций. Рекомендуется использовать лямбда-выражения вместо анонимных методов, так как их синтаксис лаконичнее и делает код более читабельным. Разумеется, компилятор до сих пор поддерживает анонимные функции, так что необходимости вносить исправления в код, написанный на C# 2.0, нет. Тем не менее в этой книге рассматривается только синтаксис лямбда-выражений.

Упрощение 3: не создаем вручную оболочку локальных переменных класса для передачи их в метод обратного вызова

Вы уже видели, что код обратного вызова может ссылаться на другие члены класса. Но иногда требуется ссылка этого кода на локальный параметр или переменную внутри определяемого метода. Вот интересный пример:

```
internal sealed class AClass {
    public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {
        // Какие-то локальные переменные
```

```
Int32[] squares = new Int32[numToDo];
AutoResetEvent done = new AutoResetEvent(false);

// Решаются задачи в других потоках
for (Int32 n = 0; n < squares.Length; n++) {
    ThreadPool.QueueUserWorkItem(
        obj => {
            Int32 num = (Int32) obj;

            // Обычно решение этой задачи требует больше времени
            squares[num] = num * num;

            // Если это последняя задача, продолжаем выполнять главный поток
            if (Interlocked.Decrement(ref numToDo) == 0)
                done.Set();
        },
        n);
}

// Ожидаем завершения остальных потоков
done.WaitOne();

// Вывод результатов
for (Int32 n = 0; n < squares.Length; n++)
    Console.WriteLine("Index {0}. Square={1}", n, squares[n]);
}
```

Этот пример демонстрирует, насколько легко в C# реализуются задачи, считавшиеся достаточно сложными. В представленном здесь методе определен единственный параметр `numToDo` и две локальные переменные `squares` и `done`. На эти переменные ссылается тело лямбда-выражения.

А теперь представим, что код из лямбда-выражения помещен в отдельный метод (как того требует CLR). Каким образом передать туда значения переменных? Для этого потребуются вспомогательный класс, определяющий поле для каждого значения, которое требуется передать в код обратного вызова. Кроме того, этот код следует определить во вспомогательном классе как экземплярный метод. Тогда метод `UsingLocalVariablesInTheCallbackCode` создаст экземпляр вспомогательного класса, присвоит полям значения локальных переменных и, наконец, создаст делегат, связанный с вспомогательным классом и экземплярным методом.

ПРИМЕЧАНИЕ

Когда лямбда-выражение заставляет компилятор генерировать класс с превращенными в поля параметрами/локальными переменными, увеличивается время жизни объекта, на который ссылаются эти переменные.

Обычно параметры/локальные переменные уничтожаются после завершения метода, в котором они используются. В данном же случае они остаются, пока не будет уничтожен объект, содержащий поле. В большинстве приложений это не имеет особого значения, тем не менее этот факт следует знать.

Это нудная и чреватая ошибками работа, и разумеется, компилятор выполнит ее за вас. Приведенный код он перепишет примерно так (комментарии мои):

```
internal sealed class AClass {
    public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {

        // Какие-то локальные переменные
        WaitCallback callback1 = null;

        // Создание экземпляра вспомогательного класса
        <>c__DisplayClass2 class1 = new <>c__DisplayClass2();

        // Инициализация полей вспомогательного класса
        class1.numToDo = numToDo;
        class1.squares = new Int32[class1.numToDo];
        class1.done = new AutoResetEvent(false);

        // Решаются задачи в других потоках
        for (Int32 n = 0; n < class1.squares.Length; n++) {
            if (callback1 == null) {
                // Новый делегат привязывается к объекту вспомогательного класса
                // и его анонимному экземплярному методу
                callback1 = new WaitCallback(
                    class1.<UsingLocalVariablesInTheCallbackCode>b__0);
            }
            ThreadPool.QueueUserWorkItem(callback1, n);
        }

        // Ожидание завершения остальных потоков
        class1.done.WaitOne();

        // Вывод результатов
        for (Int32 n = 0; n < class1.squares.Length; n++)
            Console.WriteLine("Index {0}, Square={1}", n, class1.squares[n]);
    }

    // Вспомогательному классу присваивается необычное имя, чтобы
    // избежать конфликтов и предотвратить доступ из класса AClass
    [CompilerGenerated]
    private sealed class <>c__DisplayClass2 : Object {
```

```
// В коде обратного вызова каждой локальной переменной
// используется одно открытое поле
public Int32[] squares;
public Int32 numToDo;
public AutoResetEvent done;

// Открытый конструктор без параметров
public <>c__DisplayClass2 { }

// Открытый экземплярный метод с кодом обратного вызова
public void <UsingLocalVariablesInTheCallbackCode>b__0(Object obj) {
    Int32 num = (Int32) obj;
    squares[num] = num * num;
    if (Interlocked.Decrement(ref numToDo) == 0)
        done.Set();
}
}
}
```

ВНИМАНИЕ

Без сомнения велик соблазн начать злоупотреблять лямбда-выражениями. Лично мне потребовалось время, чтобы привыкнуть к ним. Ведь код, который вы пишете внутри метода, на самом деле этому методу не принадлежит, что затрудняет отладку и пошаговое выполнение. Хотя я был откровенно поражен тем, насколько хорошо отладчик Visual Studio работал с моим кодом.

Я установил для себя правило: если в методе обратного вызова предполагается более трех строк кода, не использовать лямбда-выражения. Вместо этого следует вручную создать метод и присвоить ему имя. Впрочем, при разумном подходе лямбда-выражения способны серьезно повысить продуктивность работы программиста и упростить поддержку кода. Вот пример кода, в котором лямбда-выражения смотрятся очень естественно, и без них написание, чтение и редактирование кода было бы намного сложнее:

```
// Создание и инициализация массива String
String[] names = { "Jeff", "Kristin", "Aidan", "Grant" };

// Извлечение имен со строчной буквой 'a'
Char charToFind = 'a';
names = Array.FindAll(names, name => name.IndexOf(charToFind) >= 0);

// Преобразование всех символов строки в верхний регистр
names = Array.ConvertAll(names, name => name.ToUpper());

// Вывод результатов
Array.ForEach(names, Console.WriteLine);
```

Делегаты и отражение

Все показанные в этой главе примеры использования делегатов требовали, чтобы разработчик заранее знал прототип метода обратного вызова. Скажем, если переменная `fb` ссылается на делегат `Feedback`, код обращения к делегату мог бы выглядеть примерно так:

```
fb(item): // параметр item определен как Int32
```

Как видите, разработчик должен знать количество и тип параметров метода обратного вызова. К счастью, у вас почти всегда есть эта информация, так что написать подобный этому код — не проблема.

Впрочем, иногда возникают ситуации, когда на момент компиляции эти сведения отсутствуют. В главе 11 при обсуждении типа `EventSet` приводился соответствующий пример, в котором словарь поддерживался набором разных типов делегатов. Для вызова события во время выполнения производился поиск и вызов делегата из словаря. Однако при этом было невозможно узнать во время компиляции, какой делегат будет вызван и какие параметры следует передать его методу обратного вызова.

К счастью, в классе `System.Delegate` имеются методы, позволяющие создавать и вызывать делегаты даже при отсутствии сведений о них на момент компиляции. Вот как они выглядят:

```
public abstract class Delegate {
    // Создание делегата 'type', служащего оболочкой статического метода
    public static Delegate CreateDelegate(Type type, MethodInfo method);
    public static Delegate CreateDelegate(Type type, MethodInfo method,
        Boolean throwOnBindFailure);

    // Создание делегата 'type', служащего оболочкой экземплярного метода
    public static Delegate CreateDelegate(Type type,
        Object firstArgument, MethodInfo method); // firstArgument значит 'this'
    public static Delegate CreateDelegate(Type type,
        Object firstArgument, MethodInfo method, Boolean throwOnBindFailure);

    // Вызов делегата путем передачи ему параметров
    public Object DynamicInvoke(params Object[] args);
}
```

Все версии метода `CreateDelegate` создают новый объект, наследующий от типа `Delegate`, заданного первым параметром `type`. Параметр `MethodInfo` указывает на обратный вызов нашего метода; для получения значения вам потребуется воспользоваться API отражений (о них речь идет в главе 23). Если делегат предполагается в качестве оболочки для экземплярного метода, методу `CreateDelegate` следует передать также параметр `firstArgument`, указывающий, что объект в экземплярный метод должен передаваться через параметр `this`. Наконец, следует упомянуть, что метод `CreateDelegate` вбрасывает исключение

`ArgumentException`, когда делегат не может связаться с методом, указанным в параметре `method`. Такая ситуация возникает, когда сигнатура метода `method`, заданная в переменной, не соответствует сигнатуре, требуемой делегатом и заданной в параметре `type`. Впрочем, передав параметру `throwOnBindFailure` значение `false`, вы избежите исключения `ArgumentException`; вместо этого будет возвращено значение `null`.

ВНИМАНИЕ

Количество перегруженных версий метода `CreateDelegate` в классе `System.Delegate` намного больше, чем показано здесь. Однако вызывать их вам не нужно. Более того, в Microsoft даже жалеют о том, что эти методы существуют. Дело в том, что они определяют метод привязки с использованием типа `String` вместо параметра `MethodInfo`. В результате возможна неоднозначная привязка, ведущая к непредсказуемому поведению приложения.

Метод `DynamicInvoke` класса `System.Delegate` позволяет задействовать метод обратного вызова делегата, передавая набор параметров, определяемых во время выполнения. При вызове метода `DynamicInvoke` проверяется совместимость переданных параметров с параметрами, ожидаемыми методом обратного вызова. Для совместимых параметров выполняется обратный вызов, в противном случае вбрасывается исключение `ArgumentException`. Данный метод возвращает объект, который вернул метод обратного вызова.

Рассмотрим пример применения методов `CreateDelegate` и `DynamicInvoke`:

```
using System;
using System.Reflection;
using System.IO;

// Несколько разных определений делегатов
internal delegate Object TwoInt32s(Int32 n1, Int32 n2);
internal delegate Object OneString(String s1);
public static class Program {
    public static void Main(String[] args) {
        if (args.Length < 2) {
            String fileName = Path.GetFileNameWithoutExtension(
                Assembly.GetEntryAssembly().Location);
            String usage =
                @"Usage: " +
                "{0}{1} delType methodName [Arg1] [Arg2]" +
                "{0} where delType must be TwoInt32s or OneString"+
                "{0} if delType is TwoInt32s, methodName must be Add or Subtract" +
                "{0} if delType is OneString,
                methodName must be NumChars or Reverse" +
                "{0}" +
                "{0}Examples: " +
```

продолжение »

```

        "{0} {1} TwoInt32s Add 123 321" +
        "{0} {1} TwoInt32s Subtract 123 321" +
        "{0} {1} OneString NumChars \"Hello there\"" +
        "{0} {1} OneString Reverse \"Hello there\"";
    Console.WriteLine(usage, Environment.NewLine, fileName);
    return;
}

// Преобразование аргумента delType в тип делегата
Type delType = Type.GetType(args[0]);
if (delType == null) {
    Console.WriteLine("Invalid delType argument: " + args[0]);
    return;
}

Delegate d;
try {
    // Преобразование аргумента Arg1 в метод
    MethodInfo mi = typeof(Program).GetMethod(args[1],
        BindingFlags.NonPublic | BindingFlags.Static);

    // Создание делегата, служащего оболочкой статического метода
    d = Delegate.CreateDelegate(delType, mi);
}
catch (ArgumentException) {
    Console.WriteLine("Invalid methodName argument: " + args[1]);
    return;
}

// Создание массива, содержащего аргументы,
// передаваемые методу через делегат
Object[] callbackArgs = new Object[args.Length - 2];

if (d.GetType() == typeof(TwoInt32s)) {
    try {
        // Преобразование аргументов типа String в тип Int32
        for (Int32 a = 2; a < args.Length; a++)
            callbackArgs[a - 2] = Int32.Parse(args[a]);
    }
    catch (FormatException) {
        Console.WriteLine("Parameters must be integers.");
        return;
    }
}

if (d.GetType() == typeof(OneString)) {
    // Простое копирование аргумента типа String
    Array.Copy(args, 2, callbackArgs, 0, callbackArgs.Length);
}

```

```
try {
    // Вызов делегата и вывод результата
    Object result = d.DynamicInvoke(callbackArgs);
    Console.WriteLine("Result = " + result);
}
catch (TargetParameterCountException) {
    Console.WriteLine("Incorrect number of parameters specified.");
}
}

// Метод обратного вызова, принимающий два аргумента типа Int32
private static Object Add(Int32 n1, Int32 n2) {
    return n1 + n2;
}

// Метод обратного вызова, принимающий два аргумента типа Int32
private static Object Subtract(Int32 n1, Int32 n2) {
    return n1 - n2;
}

// Метод обратного вызова, принимающий один аргумент типа String
private static Object NumChars(String s1) {
    return s1.Length;
}

// Метод обратного вызова, принимающий один аргумент типа String
private static Object Reverse(String s1) {
    Char[] chars = s1.ToCharArray();
    Array.Reverse(chars);
    return new String(chars);
}
}
```

Глава 18. Настраиваемые атрибуты

В этой главе описывается один из самых инновационных механизмов Microsoft .NET Framework — механизм *настраиваемых атрибутов* (custom attributes). Именно они позволяют снабдить конструкторы декларативными аннотациями, что наделяет код особыми возможностями. Настраиваемые атрибуты дают возможность задать информацию, применимую практически к любой записи таблицы метаданных. Информацию об этих расширяемых метаданных можно запрашивать во время выполнения с целью динамического изменения хода выполнения программы. Настраиваемые атрибуты применяются в различных технологиях .NET Framework (Windows Forms, Web Forms, XML Web services и т. п.), давая разработчикам возможность легко реализовывать свои замыслы. Так что умение работать с настраиваемыми атрибутами требуется всем разработчикам .NET Framework.

Сфера применения настраиваемых атрибутов

Атрибуты `public`, `private`, `static` и им подобные применяются как к типам, так и к членам типов. Их полезность очевидна. А как насчет возможности задания собственных атрибутов? Предположим, нужно не просто определить тип, но и каким-либо образом указать на необходимость его сериализации. Или, к примеру, нужно применить атрибут к методу, но предварительно проверить права доступа.

Разумеется, создавать настраиваемые атрибуты и применять их к типам и методам очень удобно, однако для этого компилятор должен распознавать эти атрибуты и заносить соответствующую информацию в метаданные. Поставщики компиляторов предпочитают не открывать исходный код, поэтому специалисты Microsoft предложили альтернативный способ работы с настраиваемыми атрибутами, которые представляют собой мощный механизм, полезный как при разработке, так и при выполнении приложений. Описать и задействовать настраиваемые атрибуты может кто угодно, а все CLR-совместимые компиляторы должны их распознавать и генерировать соответствующие метаданные.

Следует понимать, что настраиваемые атрибуты представляют собой лишь средство донесения до цели некой дополнительной информации. Компилятор помещает эту информацию в метаданные управляемого модуля. Большая часть

атрибутов для компилятора просто не имеет значения; он обнаруживает их в исходном коде и создает для них соответствующие метаданные.

Библиотека классов .NET Framework (FCL) включает определения сотен настраиваемых атрибутов, которые вы можете использовать в своем коде. Вот несколько примеров:

- ❑ Атрибут `DllImport` при применении к методу информирует CLR о том, что метод реализован в неуправляемом коде указанной DLL-библиотеки.
- ❑ Атрибут `Serializable` при применении к типу информирует механизмы сериализации о том, что экземплярные поля доступны для сериализации и десериализации.
- ❑ Атрибут `AssemblyVersion` при применении к сборке задает версию сборки.
- ❑ Атрибут `Flags` при применении к перечислимому типу превращает перечислимый тип в набор битовых флагов.

Рассмотрим код с множеством примененных к нему атрибутов. В C# имена настраиваемых атрибутов помещаются в квадратные скобки непосредственно перед именем класса, объекта и т. п. Не пытайтесь понять, что именно делает код, так как он написан всего лишь для демонстрации атрибутов:

```
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
internal sealed class OSVERSIONINFO {
    public OSVERSIONINFO() {
        OSVersionInfoSize = (UInt32) Marshal.SizeOf(this);
    }

    public UInt32 OSVersionInfoSize = 0;
    public UInt32 MajorVersion = 0;
    public UInt32 MinorVersion = 0;
    public UInt32 BuildNumber = 0;
    public UInt32 PlatformId = 0;

    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public String CSDVersion = null;
}

internal sealed class MyClass {
    [DllImport("Kernel32", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern Boolean GetVersionEx([In, Out] OSVERSIONINFO ver);
}
```

В данном случае атрибут `StructLayout` применяется к классу `OSVERSIONINFO`, атрибут `MarshalAs` — к полю `CSDVersion`, атрибут `DllImport` — к методу `GetVersionEx`, а атрибуты `In` и `Out` — к параметру `ver` метода `GetVersionEx`. В каждом языке

свой синтаксис применения настраиваемых атрибутов. Например, в Visual Basic .NET вместо квадратных скобок используются угловые (< >).

CLR позволяет применять атрибуты ко всему, что может быть представлено метаданными. Чаще всего они применяются к записям в следующих таблицах определений: `TypeDef` (классы, структуры, перечисления, интерфейсы и делегаты), `MethodDef` (конструкторы), `ParamDef`, `FieldDef`, `PropertyDef`, `EventDef`, `AssemblyDef` и `ModuleDef`. В частности, C# позволяет применять настраиваемые атрибуты только к исходному коду, в котором определены такие элементы, как сборки, модули, типы (класс, структура, перечисление, интерфейс, делегат), поля, методы (в том числе конструкторы), параметры методов, возвращаемые методами значения, свойства, события, параметры обобщенного типа.

Вы можете задать префикс, указывающий, к чему будет применен атрибут. Возможные варианты префиксов представлены в показанном далее фрагменте кода. Впрочем, как понятно из предыдущего примера, компилятор часто способен определить назначение атрибута даже при отсутствии префикса. Обязательные префиксы выделены полужирным шрифтом:

```
using System;

[assembly: SomeAttr]    // Применяется к сборке
[module: SomeAttr]     // Применяется к модулю

[type: SomeAttr]       // Применяется к типу
internal sealed class SomeType<[typevar: SomeAttr] T> { // Применяется
    // к переменной универсального типа

    [field: SomeAttr]    // Применяется к полю
    public Int32 SomeField = 0;

    [return: SomeAttr]   // Применяется к возвращаемому значению
    [method: SomeAttr]   // Применяется к методу
    public Int32 SomeMethod(
        [param: SomeAttr] // Применяется к параметру
        Int32 SomeParam) { return SomeParam; }

    [property: SomeAttr] // Применяется к свойству
    public String SomeProp {
        [method: SomeAttr] // Применяется к механизму чтения get
        get { return null; }
    }

    [event: SomeAttr]    // Применяется к событиям
    [field: SomeAttr]    // Применяется к полям, созданным компилятором
    [method: SomeAttr]   // Применяется к созданным
                        // компилятором методам add и remove
    public event EventHandler SomeEvent;
}
```

Теперь, когда вы знаете, как применять настраиваемые атрибуты, давайте разберемся, что они собой представляют. Настраиваемый атрибут — это всего лишь экземпляр типа. Для соответствия общезыковой спецификации (CLS) он должен прямо или косвенно наследовать от абстрактного класса `System.Attribute`. В C# допустимы только CLS-совместимые атрибуты. В документации на .NET Framework SDK можно обнаружить определения следующих классов из предыдущего примера: `StructLayoutAttribute`, `MarshalAsAttribute`, `DllImportAttribute`, `InAttribute` и `OutAttribute`. Все они находятся в пространстве имен `System.Runtime.InteropServices`, при этом классы атрибутов могут определяться в любом пространстве имен. Можно заметить, что все перечисленные классы являются производными от класса `System.Attribute`, как и следует для CLS-совместимых атрибутов.

ПРИМЕЧАНИЕ

При определении атрибута компилятор позволяет опускать суффикс `Attribute`, что упрощает набор кода и делает его более читабельным. Я активно использую эту возможность в приводимых в книге примерах. Например, пишу `[DllImport(...)]` вместо `[DllImportAttribute(...)]`.

Как уже упоминалось, атрибуты являются экземплярами класса. И этот класс должен иметь открытый конструктор для создания экземпляров. А значит, синтаксис применения атрибутов аналогичен вызову конструктора. Кроме того, используемый язык может поддерживать специальный синтаксис определения открытых полей или свойств класса атрибутов. Рассмотрим это на примере. Вернемся к приложению, в котором атрибут `DllImport` применяется к методу `GetVersionEx`:

```
[DllImport("Kernel32", CharSet = CharSet.Auto, SetLastError = true)]
```

Вряд ли вы будете когда-нибудь использовать подобный синтаксис для вызова конструктора. Согласно описанию класса `DllImportAttribute` в документации, его конструктор требует единственного параметра типа `String`. В рассматриваемом примере в качестве параметра передается строка `"Kernel32"`. Параметры конструктора называются *позиционными* (*positional parameters*). При применении атрибута следует обязательно их указывать.

А что с еще двумя «параметрами»? Показанный особый синтаксис позволяет задавать любые открытые поля или свойства объекта `DllImportAttribute` после его создания. В рассматриваемом примере при создании этого объекта его конструктору передается строка `"Kernel32"`, а открытым экземплярным полям `CharSet` и `SetLastError` присваиваются значения `CharSet.Auto` и `true` соответственно. «Параметры», задающие поля или свойства, называются *именованными* (*named parameters*); они являются необязательными. Чуть позже мы рассмотрим, как инициировать конструирование экземпляра класса `DllImportAttribute`.

Следует заметить, что к одному элементу можно применить несколько атрибутов. Скажем, в приведенном в начале главы фрагменте кода к параметру `ver` метода `GetVersionEx` применяются атрибуты `In` и `Out`. Что интересно, порядок следования атрибутов в такой ситуации не имеет значения. В С# отдельные атрибуты заключаются в квадратные скобки, в то время как наборы атрибутов перечисляются в этих скобках через запятую. Если конструктор класса атрибута не имеет параметров, круглые скобки можно опустить. Ну и, как уже упоминалось, суффикс `Attribute` также является необязательным. Показанные далее строки приводят к одному и тому же результату и демонстрируют все возможные способы применения набора атрибутов:

```
[Serializable][Flags]
[Serializable, Flags]
[FlagsAttribute, SerializableAttribute]
[FlagsAttribute()][Serializable()]
```

Определение класса атрибутов

Вы уже знаете, что любой атрибут наследует от класса `System.Attribute`, и умеете применять атрибуты. Пришло время рассмотреть процесс их создания. Представьте, что вы работаете в Microsoft и получили задание реализовать поддержку битовых флагов в перечислимых типах. Для начала вам нужно определить класс `FlagsAttribute`:

```
namespace System {
    public class FlagsAttribute : System.Attribute {
        public FlagsAttribute() {
        }
    }
}
```

Обратите внимание, что класс `FlagsAttribute` наследует от класса `Attribute`; именно это делает его CLS-совместимым с настраиваемым атрибутом. Вдобавок в имени класса присутствует суффикс `Attribute`. Это соответствует стандарту именования, хотя и не является обязательным. Наконец, все неабстрактные атрибуты должны содержать хотя бы один открытый конструктор. Простейший конструктор `FlagsAttribute` не имеет параметров и не выполняет никаких действий.

ВНИМАНИЕ

Атрибут следует рассматривать как логический контейнер состояния. Иначе говоря, хотя атрибут и является классом, этот класс должен быть крайне простым. Он должен содержать всего один открытый конструктор, принимающий обязательную (или позиционную) информацию о состоянии атрибута. Также класс может содержать открытые поля/свойства, принимающие

дополнительную (или именованную) информацию о состоянии атрибута. В классе не должно быть открытых методов, событий или других членов.

В общем случае я не одобряю использование открытых полей. Атрибутов это тоже касается. Лучше воспользоваться свойствами, так как они обеспечивают большую гибкость в случаях, когда требуется внести изменения в реализацию класса атрибутов.

Получается, что экземпляры класса `FlagsAttribute` можно применять к чему угодно, хотя реально этот атрибут следует применять только к перечислимым типам. Нет смысла применять его к свойству или методу. Чтобы указать компилятору область действия атрибута, применим к классу атрибута экземпляр класса `System.AttributeUsageAttribute`:

```
namespace System {
    [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
    public class FlagsAttribute : System.Attribute {
        public FlagsAttribute() {
        }
    }
}
```

В этой новой версии экземпляр `AttributeUsageAttribute` применен к атрибуту. В конце концов, атрибуты — это всего лишь классы, а значит, к ним, в свою очередь, можно применять другие атрибуты. Атрибут `AttributeUsage` является простым классом, указывающим компилятору область действия настраиваемого атрибута. Все компиляторы имеют встроенную поддержку этого атрибута и при попытке применить его к недопустимому элементу выдают сообщение об ошибке. В рассматриваемом примере атрибут `AttributeUsage` указывает, что экземпляры атрибута `Flags` работают только с перечислимыми типами.

Так как все атрибуты являются типами, понять, как устроен класс `AttributeUsageAttribute`, несложно. Вот исходный код этого класса в FCL:

```
[Serializable]
[AttributeUsage(AttributeTargets.Class, Inherited=true)]
public sealed class AttributeUsageAttribute : Attribute {
    internal static AttributeUsageAttribute Default =
        new AttributeUsageAttribute(AttributeTargets.All);

    internal Boolean m_allowMultiple = false;
    internal AttributeTargets m_attributeTarget = AttributeTargets.All;
    internal Boolean m_inherited = true;

    // Это единственный открытый конструктор
    public AttributeUsageAttribute(AttributeTargets validOn) {
        m_attributeTarget = validOn;
    }

    internal AttributeUsageAttribute(AttributeTargets validOn,
        Boolean allowMultiple, Boolean inherited) {
```

```

    m_attributeTarget = validOn;
    m_allowMultiple = allowMultiple;
    m_inherited = inherited;
}

public Boolean AllowMultiple {
    get { return m_allowMultiple; }
    set { m_allowMultiple = value; }
}

public Boolean Inherited {
    get { return m_inherited; }
    set { m_inherited = value; }
}

public AttributeTargets ValidOn {
    get { return m_attributeTarget; }
}
}

```

Как видите, класс `AttributeUsageAttribute` имеет открытый конструктор, позволяющий передавать битовые флаги, задающие область применения атрибута. Перечислимый тип `System.AttributeTargets` определяется в FCL так:

```

[Flags, Serializable]
public enum AttributeTargets {
    Assembly = 0x0001,
    Module = 0x0002,
    Class = 0x0004,
    Struct = 0x0008,
    Enum = 0x0010,
    Constructor = 0x0020,
    Method = 0x0040,
    Property = 0x0080,
    Field = 0x0100,
    Event = 0x0200,
    Interface = 0x0400,
    Parameter = 0x0800,
    Delegate = 0x1000,
    ReturnValue = 0x2000,
    GenericParameter = 0x4000,
    All = Assembly | Module | Class | Struct | Enum |
    Constructor | Method | Property | Field | Event |
    Interface | Parameter | Delegate | ReturnValue |
    GenericParameter
}

```

У класса `AttributeUsageAttribute` есть два дополнительных открытых свойства, которым при применении атрибута к классу могут быть присвоены значения `AllowMultiple` и `Inherited`.

Большинство атрибутов не имеет смысла применять к одному элементу более одного раза. Например, вам ничего не даст последовательное применение атрибута `Flags` или `Serializable`:

```
[Flags][Flags]
internal enum Color {
    Red
}
```

Более того, при попытке компиляции такого кода появится сообщение об ошибке (ошибка CS0579: дублирование атрибута `Flags`):

```
error CS0579: Duplicate 'Flags' attribute
```

Однако есть и атрибуты, которые имеет смысл применять к одному элементу несколько раз. В FCL это класс атрибутов `ConditionalAttribute`. Но для этого параметру `AllowMultiple` следует присвоить значение `true`. В противном случае вы не сможете применить атрибут несколько раз.

Свойство `Inherited` класса `AttributeUsageAttribute` указывает, будет ли атрибут, применяемый к базовому классу, применяться также к производным классам и переопределенным методам. Суть наследования атрибута демонстрирует следующий код:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    Inherited=true)]
internal class TastyAttribute : Attribute {
}

[Tasty][Serializable]
internal class BaseType {
    [Tasty] protected virtual void DoSomething() { }
}

internal class DerivedType : BaseType {
    protected override void DoSomething() { }
}
```

В этом коде класс `DerivedType` и его метод `DoSomething` снабжены атрибутом `Tasty`, так как класс `TastyAttribute` помечен как наследуемый. Но класс `DerivedType` несериализуемый, ведь класс `SerializableAttribute` в FCL помечен как ненаследуемый атрибут.

Следует помнить, что в .NET Framework наследование атрибутов допустимо только для классов, методов, свойств, событий, полей, возвращаемых значений и параметров. Не забывайте об этом, присваивая параметру `Inherited` значение

true. Кстати, при наличии наследуемых атрибутов дополнительные метаданные в управляемый модуль для производных типов не добавляются. Более подробно мы поговорим об этом чуть позже.

ПРИМЕЧАНИЕ

Если при определении собственного класса атрибутов вы забудете применить атрибут `AttributeUsage`, компилятор и CLR будут рассматривать полученный результат как применимый к любым элементам, но только один раз. Кроме того, он будет наследуемым. Именно такие значения по умолчанию имеют поля класса `AttributeUsageAttribute`.

Конструктор атрибута и типы данных полей и свойств

Определяя класс настраиваемых атрибутов, можно указать конструктор с параметрами, которые должен задавать разработчик, использующий экземпляр атрибута. Кроме того, вы можете определить нестатические открытые поля и свойства своего типа, которые разработчик может задавать по желанию.

Определяя конструктор экземпляров класса атрибутов, а также поля и свойства, следует ограничиться небольшим набором типов данных. Допустимы типы: `Boolean`, `Char`, `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `String`, `Type`, `Object` и перечислимые типы. Можно использовать также одномерные массивы этих типов с нулевой нижней границей, но это не рекомендуется, так как класс настраиваемых атрибутов, конструктор которого умеет работать с массивами, не относится к CLS-совместимым.

Применяя атрибут, следует указывать определенное при компиляции постоянное выражение, совпадающее с типом, заданным классом атрибута. Каждый раз, когда в классе атрибута определяется параметр, поле или свойство типа `Type`, следует использовать оператор `typeof` языка C#, как показано в следующем фрагменте кода. А для параметров, полей и свойств типа `Object` можно передавать значения типа `Int32`, `String` и другие постоянные выражения (в том числе `null`). Если постоянное выражение принадлежит к значимому типу, этот тип будет упакован при создании экземпляра атрибута.

Вот пример применения атрибута:

```
using System;
internal enum Color { Red }
[AttributeUsage(AttributeTargets.All)]
internal sealed class SomeAttribute : Attribute {
    public SomeAttribute(String name, Object o, Type[] types) {
        // 'name' ссылается на String
```

```
// 'o' ссылается на один из легальных типов (упаковка при необходимости)
// 'types' ссылается на одномерный массив Types
// с нулевой нижней границей
}
}
[Some("Jeff", Color.Red, new Type[] { typeof(Math), typeof(Console) })]
internal sealed class SomeType {
}
```

Обнаружив настраиваемый атрибут, компилятор создает экземпляр класса этого атрибута, передавая его конструктору все указанные параметры. Затем он присваивает значения открытым полям и свойствам, используя для этого усовершенствованный синтаксис конструктора. Инициализировав объект, являющийся настраиваемым атрибутом, компилятор сериализует его и сохраняет в таблице метаданных.

ВНИМАНИЕ

Настраиваемый атрибут лучше всего представить следующим образом: экземпляр класса, сериализованный в байтовый поток, находящийся в метаданных. В период выполнения байты из метаданных десериализуются. На самом деле компилятор генерирует информацию, необходимую для создания экземпляра класса атрибутов, и размещает ее в метаданных. Каждый параметр конструктора записывается в однобайтный идентификатор, за которым следует его значение. Завершив сериализацию параметров, компилятор генерирует значения для каждого указанного поля и свойства, записывая их имена и значения в однобайтный идентификатор типа. Для массивов сначала указывается количество элементов.

Выявление настраиваемых атрибутов

Само по себе определение атрибутов бесполезно. Вы можете определить любой класс атрибута и применить его в произвольном месте, но это приведет только к появлению в вашей сборке дополнительных метаданных, никак не сказавшись на работе приложения.

Как показано в главе 15, применение к перечислимому типу `System.Enum` атрибута `Flags` меняет поведение его методов `ToString` и `Format`. Причиной этому является происходящая во время выполнения проверка, не связан ли атрибут `Flags` с перечислимым типом, с которым работают данные методы. Код может анализироваться на наличие атрибутов при помощи технологии, называемой *отражением* (reflection). Подробно она рассматривается в главе 23, здесь же я продемонстрирую ее применение.

Если бы вы отвечали за реализацию метода `Format` типа `Enum`, вы бы сделали это так:

```
public static String Format(Type enumType, Object value, String format) {

    // Применяется ли к перечислимому типу экземпляр типа FlagsAttribute?
    if (enumType.IsDefined(typeof(FlagsAttribute), false)) {
        // Да; выполняем код, трактующий значение как
        // перечислимый тип с битовыми флагами
        ...
    } else {
        // Нет; выполняем код, трактующий значение как
        // обычный перечислимый тип
        ...
    }
}
```

Этот код обращается к методу `IsDefined` типа `Type`, заставляя систему посмотреть метаданные этого перечислимого типа и определить, связан ли с ним экземпляр класса `FlagsAttribute`. Если метод `IsDefined` возвращает значение `true`, значит, экземпляр `FlagsAttribute` связан с перечислимым типом, и метод `Format` будет считать, что переданное значение содержит набор битовых флагов. В противном случае переданное значение будет восприниматься как обычный перечислимый тип.

То есть после определения собственных классов атрибутов нужно также написать код, проверяющий, существует ли экземпляр класса атрибута (для указанных элементов), и в зависимости от результата меняющий порядок выполнения программы. Только таким образом настраиваемый атрибут принесет пользу!

Проверить наличие атрибута в FCL можно разными способами. Для объектов класса `System.Type` имеет смысл использовать метод `IsDefined`, как показано ранее. Но иногда требуется проверить наличие атрибута не для типа, а для сборки, модуля или метода. Остановимся на методах класса `System.Attribute`. Именно он является базовым для CLS-совместимых атрибутов. В этом классе для получения атрибутов имеются три статических метода: `IsDefined`, `GetCustomAttributes` и `GetCustomAttribute`. Каждый из них имеет несколько перегруженных версий. К примеру, одна версия каждого из методов работает с членами типа (классами, структурами, перечислениями, интерфейсами, делегатами, конструкторами, методами, свойствами, полями, событиями и возвращаемыми типами), параметрами, модулями и сборками. Также существуют версии, позволяющие просматривать иерархию наследования и включать в результат наследуемые атрибуты. Краткое описание методов дано в табл. 18.1.

Таблица 18.1. Методы класса `System.Attribute`, определяющие наличие в метаданных CIS-совместимых настраиваемых атрибутов

Метод	Описание
<code>IsDefined</code>	Возвращает значение <code>true</code> при наличии хотя бы одного экземпляра указанного класса, производного от <code>Attribute</code> . Работает быстро, так как не создает (не десериализует) никаких экземпляров класса атрибута
<code>GetCustomAttributes</code>	Возвращает массив, каждый элемент которого является экземпляром указанного класса атрибута. Если методу не передается класс атрибута, массив будет содержать экземпляры всех примененных атрибутов вне зависимости от их класса. Каждый экземпляр создается (десериализуется) с использованием указанных при компиляции параметров, полей и свойств. Если элемент, с которым ведется работа, не имеет экземпляров указанного класса атрибута, метод возвращает пустой массив. Обычно метод работает с атрибутами, параметр <code>AllowMultiple</code> которых имеет значение <code>true</code> , или со списком всех примененных атрибутов
<code>GetCustomAttribute</code>	Возвращает экземпляр указанного класса атрибута. Каждый экземпляр создается (десериализуется) с использованием указанных при компиляции параметров, полей и свойств. Если элемент, с которым ведется работа, не имеет экземпляров указанного класса атрибута, метод возвращает значение <code>null</code> . При наличии более чем одного экземпляра вбрасывается исключение <code>System.Reflection.AmbiguousMatchException</code> . Обычно метод работает с атрибутами, параметр <code>AllowMultiple</code> которых имеет значение <code>false</code>

Если нужно установить только сам факт применения атрибута, используйте метод `IsDefined` как самый быстрый из перечисленных. Однако при применении атрибута, как известно, можно задавать параметры его конструктору и при необходимости определять свойства и поля, а этого метод `IsDefined` делать не умеет.

Для создания атрибутов используйте метод `GetCustomAttributes` или `GetCustomAttribute`. При каждом вызове этих методов создаются экземпляры указанных классов атрибутов, и на основе указанных в исходном коде значений задаются поля и свойства каждого экземпляра. Эти методы возвращают ссылки на сконструированные экземпляры классов атрибутов.

Эти методы просматривают данные управляемого модуля и сравнивают строки в поиске указанного класса настраиваемого атрибута. Эти операции требуют времени, поэтому если вас волнует быстроедействие, подумайте о кэшировании результатов работы методов. В этом случае вам не придется вызывать их раз за разом, запрашивая одну и ту же информацию.

В пространстве имен `System.Reflection` находятся классы, позволяющие анализировать содержимое метаданных модуля: `Assembly`, `Module`, `ParameterInfo`,

MemberInfo, Type, MethodInfo, ConstructorInfo, FieldInfo, EventInfo, PropertyInfo и соответствующие им классы *Builder. Все эти классы содержат методы IsDefined и GetCustomAttributes. Однако только класс System.Attribute предлагает весьма удобный метод GetCustomAttribute.

Версия метода GetCustomAttributes, определенная в классах, связанных с отражением, возвращает массив экземпляров Object[] типа Object вместо массива экземпляров Attribute[] типа Attribute. Дело в том, что классы, связанные с отражением, могут возвращать объекты из классов атрибута, не соответствующих спецификации CLS. К счастью, такие атрибуты встречаются крайне редко. За все время моей работы с .NET Framework я не сталкивался с ними ни разу.

ПРИМЕЧАНИЕ

Имейте в виду, что методы отражения, поддерживающие логический параметр inherit, реализуют только классы Attribute, Type и MethodInfo. Все прочие методы отражения этот параметр игнорируют и иерархию наследования не проверяют. Для проверки наличия унаследованного атрибута в событиях, свойствах, полях, конструкторах или параметрах используйте один из методов класса Attribute.

Есть еще один аспект, о котором следует помнить. После передачи класса методам IsDefined, GetCustomAttribute или GetCustomAttributes они начинают искать этот класс атрибута или производные от него. Для поиска конкретного класса атрибута требуется дополнительная проверка возвращенного значения, которая гарантирует возвращение именно того класса, который вам нужен. Чтобы избежать недоразумений и дополнительных проверок, можно определить класс с модификатором sealed.

Вот пример рассмотрения методов внутри типа и отображения применяемых к каждому их методов атрибутов. Это демонстрационный код; обычно никто не применяет указанные настраиваемые атрибуты подобным образом:

```
using System;
using System.Diagnostics;
using System.Reflection;

[assembly: CLSCompliant(true)]

[Serializable]
[DefaultMemberAttribute("Main")]
[DebuggerDisplayAttribute("Richter", Name = "Jeff",
    Target = typeof(Program))]
public sealed class Program {
    [Conditional("Debug")]
    [Conditional("Release")]
    public void DoSomething() { }
```

```
public Program() {
}

[CLSCompliant(true)]
[STAThread]
public static void Main() {
    // Вывод набора атрибутов, примененных к типу
    ShowAttributes(typeof(Program));

    // Получение и задание методов, связанных с типом
    MemberInfo[] members = typeof(Program).FindMembers(
        MemberTypes.Constructor | MemberTypes.Method,
        BindingFlags.DeclaredOnly | BindingFlags.Instance |
        BindingFlags.Public | BindingFlags.Static,
        Type.FilterName, "");

    foreach (MemberInfo member in members) {
        // Вывод набора атрибутов, примененных к члену
        ShowAttributes(member);
    }
}

private static void ShowAttributes(MemberInfo attributeTarget) {
    Attribute[] attributes = Attribute.GetCustomAttributes(attributeTarget);

    Console.WriteLine("Attributes applied to {0}: {1}",
        attributeTarget.Name, (
            attributes.Length == 0 ? "None" : String.Empty));

    foreach (Attribute attribute in attributes) {
        // Вывод типа всех примененных атрибутов
        Console.WriteLine(" {0}", attribute.GetType().ToString());

        if (attribute is DefaultMemberAttribute)
            Console.WriteLine(" MemberName={0}",
                ((DefaultMemberAttribute) attribute).MemberName);

        if (attribute is ConditionalAttribute)
            Console.WriteLine(" ConditionString={0}",
                ((ConditionalAttribute) attribute).ConditionString);

        if (attribute is CLSCompliantAttribute)
            Console.WriteLine(" IsCompliant={0}",
                ((CLSCompliantAttribute) attribute).IsCompliant);

        DebuggerDisplayAttribute dda = attribute as DebuggerDisplayAttribute;
        if (dda != null) {
            Console.WriteLine(" Value={0}, Name={1}, Target={2}",
```

```

        dda.Value, dda.Name, dda.Target);
    }
}
Console.WriteLine();
}
}

```

Скомпоновав и запустив это приложение, мы получим следующий результат:

```

Attributes applied to Program:
System.SerializableAttribute
System.Diagnostics.DebuggerDisplayAttribute
    Value=Richter, Name=Jeff, Target=Program
System.Reflection.DefaultMemberAttribute
    MemberName=Main

```

```

Attributes applied to DoSomething:
System.Diagnostics.ConditionalAttribute
    ConditionString=Release
System.Diagnostics.ConditionalAttribute
    ConditionString=Debug

```

```

Attributes applied to Main:
System.CLSCompliantAttribute
    IsCompliant=True
System.STAThreadAttribute

```

```

Attributes applied to .ctor: None

```

Сравнение экземпляров атрибута

Теперь, когда вы умеете находить экземпляры атрибутов в коде, имеет смысл рассмотреть процедуру анализа их полей. Можно, к примеру, написать код, явным образом проверяющий значение каждого поля класса атрибута. Однако класс `System.Attribute` переопределяет метод `Equals` класса `Object`, заставляя его сравнивать типы объектов. Если они не совпадают, метод возвращает значение `false`. В случае же совпадения метод `Equals` использует отражения для сравнения полей двух атрибутов (вызывая метод `Equals` для каждого поля). Если все поля совпадают, возвращается значение `true`. Можно переопределить метод `Equals` в вашем собственном классе атрибутов, убрав из него отражения и повысив тем самым производительность.

Класс `System.Attribute` содержит также виртуальный метод `Match`, который вы можете переопределить для получения более богатой семантики. По умолчанию данный метод просто вызывает метод `Equals` и возвращает полученный

результат. Следующий код демонстрирует переопределение методов Equals и Match (значение true возвращается, если один атрибут представляет собой подмножество другого) и применение второго из них:

```
using System;
[Flags]
internal enum Accounts {
    Savings = 0x0001,
    Checking = 0x0002,
    Brokerage = 0x0004
}

[AttributeUsage(AttributeTargets.Class)]
internal sealed class AccountsAttribute : Attribute {
    private Accounts m_accounts;

    public AccountsAttribute(Accounts accounts) {
        m_accounts = accounts;
    }

    public override Boolean Match(Object obj) {
        // Если в базовом классе реализован метод Match и это не
        // класс Attribute, раскомментируйте следующую строку
        // if (!base.Match(obj)) return false;

        // Так как 'this' не равен null, если obj равен null,
        // объекты не совпадают
        // ПРИМЕЧАНИЕ. Эту строку можно удалить, если вы считаете,
        // что базовый тип корректно реализует метод Match
        if (obj == null) return false;

        // Объекты разных типов не могут быть равны
        // ПРИМЕЧАНИЕ. Эту строку можно удалить, если вы считаете,
        // что базовый тип корректно реализует метод Match
        if (this.GetType() != obj.GetType()) return false;

        // Приведение obj к нашему типу для доступа к полям
        // ПРИМЕЧАНИЕ. Это приведение всегда работает,
        // так как объекты принадлежат к одному типу
        AccountsAttribute other = (AccountsAttribute) obj;

        // Сравнение полей
        // Проверка, является ли accounts 'this' подмножеством
        // accounts объекта others
        if ((other.m_accounts & m_accounts) != m_accounts)
            return false;
    }
}
```

```

        return true; // Объекты совпадают
    }

    public override Boolean Equals(Object obj) {
        // Если в базовом классе реализован метод Equals и это
        // не класс Object, раскомментируйте следующую строку
        // if (!base.Equals(obj)) return false;

        // Так как 'this' не равен null, при obj равном null
        // объекты не совпадают
        // ПРИМЕЧАНИЕ. Эту строку можно удалить, если вы считаете,
        // что базовый тип корректно реализует метод Equals
        if (obj == null) return false;

        // Объекты разных типов не могут совпасть
        // ПРИМЕЧАНИЕ. Эту строку можно удалить, если вы считаете,
        // что базовый тип корректно реализует метод Equals
        if (this.GetType() != obj.GetType()) return false;

        // Приведение obj к нашему типу для получения доступа к полям
        // ПРИМЕЧАНИЕ. Это приведение работает всегда,
        // так как объекты принадлежат к одному типу
        AccountsAttribute other = (AccountsAttribute) obj;

        // Сравнение значений полей
        // Проверяем, равен ли accounts 'this'
        // accounts объекта other
        if (other.m_accounts != m_accounts)
            return false;

        return true; // Объекты совпадают
    }

    // Переопределяем GetHashCode, так как Equals уже переопределен
    public override Int32 GetHashCode() {
        return (Int32) m_accounts;
    }
}

[Accounts(Accounts.Savings)]
internal sealed class ChildAccount { }

[Accounts(Accounts.Savings | Accounts.Checking | Accounts.Brokerage)]
internal sealed class AdultAccount { }

public sealed class Program {
    public static void Main() {
        CanWriteCheck(new ChildAccount());
    }
}

```

```
CanWriteCheck(new AdultAccount());

// Это демонстрирует корректность работы метода для
// типа, к которому не был применен атрибут AccountsAttribute
CanWriteCheck(new Program());
}

private static void CanWriteCheck(Object obj) {
    // Создание и инициализация экземпляра типа атрибута
    Attribute checking = new AccountsAttribute(Accounts.Checking);

    // Создание экземпляра атрибута, примененного к типу
    Attribute validAccounts = Attribute.GetCustomAttribute(
        obj.GetType(), typeof(AccountsAttribute), false);

    // Если атрибут применен к типу и указывает на счет "Checking",
    // значит, тип может выписывать чеки
    if ((validAccounts != null) && checking.Match(validAccounts)) {
        Console.WriteLine("{0} types can write checks.", obj.GetType());
    } else {
        Console.WriteLine("{0} types can NOT write checks.", obj.GetType());
    }
}
}
```

Компоновка и запуск этого приложения приведет к следующему результату:

```
ChildAccount types can NOT write checks.
AdultAccount types can write checks.
Program types can NOT write checks.
```

Выявление настраиваемых атрибутов без создания производных от класса Attribute объектов

В этом разделе мы поговорим об альтернативном способе обнаружения настраиваемых атрибутов, примененных к метаданным. В сценариях, требующих повышенной безопасности, данная техника гарантирует, что код класса, производного от класса Attribute, выполняться не будет. Вообще говоря, при вызове методов GetCustomAttribute(s) типа Attribute вызывается конструктор класса атрибута и методы, задающие значения свойств. А первое обращение к типу заставляет CLR вызвать конструктор этого типа (если он, конечно, существует). Конструктор, метод доступа set и методы конструктора типа могут содержать код, выполняющийся при каждом поиске атрибута. Это позволяет выполнить в домене приложения неизвестный код, создавая угрозу безопасности.

Для обнаружения атрибутов без выполнения кода класса атрибута применяется класс `System.Reflection.CustomAttributeData`. В нем определен единственный статический метод `GetCustomAttributes`, позволяющий получить информацию о примененных атрибутах. Этот метод имеет четыре перегруженные версии: одна принимает параметр типа `Assembly`, другая — `Module`, третья — `ParameterInfo`, последняя — `MemberInfo`. Класс определен в пространстве имен `System.Reflection`, о котором речь идет в главе 23. Обычно класс `CustomAttributeData` используется для анализа метаданных сборки, загружаемой статическим методом `ReflectionOnlyLoad` класса `Assembly` (он также рассматривается в главе 23). Пока же только упомянем, что этот метод загружает сборку таким образом, что CLR не может выполнять какой-либо код, в том числе конструкторы типов.

Метод `GetCustomAttributes` класса `CustomAttributeData` можно сравнить с фабрикой. Он возвращает набор объектов `CustomAttributeData` в перечисление `IList<CustomAttributeData>`. Каждому элементу этой коллекции соответствует один настраиваемый атрибут. Для каждого объекта класса `CustomAttributeData` можно запросить предназначенные только для чтения свойства, определив в результате, каким способом *мог бы быть* сконструирован и инициализирован объект. Например, свойство `Constructor` указывает, какой именно конструктор *мог бы быть* вызван. Свойство `ConstructorArguments` возвращает аргументы, которые *могли бы быть* переданы в этот конструктор в качестве экземпляра `IList<CustomAttributeTypedArgument>`. Свойство `NamedArguments` возвращает поля и свойства, которые *могли бы быть* заданы как экземпляры `IList<CustomAttributeNamedArgument>`. Условное наклонение во всех этих предложениях обусловлено тем, что ни конструктор, ни метод доступа `set` на самом деле не вызываются. Для безопасности запрещено выполнение любых методов класса атрибута.

Вот отредактированная версия предыдущего кода, в которой для безопасного получения атрибутов используется класс `CustomAttributeData`:

```
using System;
using System.Diagnostics;
using System.Reflection;
using System.Collections.Generic;
```

```
[assembly: CLSCompliant(true)]
```

```
[Serializable]
```

```
[DefaultMemberAttribute("Main")]
```

```
[DebuggerDisplayAttribute("Richter", Name="Jeff", Target=typeof(Program))]
```

```
public sealed class Program {
```

```
    [Conditional("Debug")]
```

```
    [Conditional("Release")]
```

```
    public void DoSomething() { }
```

```
    public Program() {
    }
```

```
[CLSCompliant(true)]
[STAThread]
public static void Main() {
    // Вывод атрибутов, примененных к данному типу
    ShowAttributes(typeof(Program));

    // Получение набора связанных с типом методов
    MemberInfo[] members = typeof(Program).FindMembers(
        MemberTypes.Constructor | MemberTypes.Method,
        BindingFlags.DeclaredOnly | BindingFlags.Instance |
        BindingFlags.Public | BindingFlags.Static,
        Type.FilterName, "*");

    foreach (MemberInfo member in members) {
        // Вывод атрибутов, примененных к данному члену
        ShowAttributes(member);
    }
}

private static void ShowAttributes(MemberInfo attributeTarget) {
    IList<CustomAttributeData> attributes =
        CustomAttributeData.GetCustomAttributes(attributeTarget);

    Console.WriteLine("Attributes applied to {0}: {1}",
        attributeTarget.Name, (
            attributes.Count == 0 ? "None" : String.Empty));

    foreach (CustomAttributeData attribute in attributes) {
        // Вывод типа каждого примененного атрибута
        Type t = attribute.Constructor.DeclaringType;
        Console.WriteLine(" {0}", t.ToString());
        Console.WriteLine(" Constructor called={0}", attribute.Constructor);

        IList<CustomAttributeTypedArgument> posArgs =
            attribute.ConstructorArguments;
        Console.WriteLine(" Positional arguments passed to constructor:" +
            ((posArgs.Count == 0) ? " None" : String.Empty));
        foreach (CustomAttributeTypedArgument pa in posArgs) {
            Console.WriteLine(" Type={0}, Value={1}",
                pa.ArgumentType, pa.Value);
        }

        IList<CustomAttributeNamedArgument> namedArgs =
            attribute.NamedArguments;
        Console.WriteLine(" Named arguments set after construction:" +
            ((namedArgs.Count == 0) ? " None" : String.Empty));
        foreach (CustomAttributeNamedArgument na in namedArgs) {
```

```

        Console.WriteLine(" Name={0}, Type={1}, Value={2}",
            na.MemberInfo.Name, na.TypedValue.ArgumentType,
            na.TypedValue.Value);
    }

    Console.WriteLine();
}
Console.WriteLine();
}
}

```

Компоновка и запуск этого приложения приведут к следующему результату:

Attributes applied to Program:

```

System.SerializableAttribute
  Constructor called=Void .ctor()
  Positional arguments passed to constructor: None
  Named arguments set after construction: None

```

System.Diagnostics.DebuggerDisplayAttribute

```

  Constructor called=Void .ctor(System.String)
  Positional arguments passed to constructor:
    Type=System.String, Value=Richter
  Named arguments set after construction:
    Name=Name, Type=System.String, Value=Jeff
    Name=Target, Type=System.Type, Value=Program

```

System.Reflection.DefaultMemberAttribute

```

  Constructor called=Void .ctor(System.String)
  Positional arguments passed to constructor:
    Type=System.String, Value=Main
  Named arguments set after construction: None

```

Attributes applied to DoSomething:

```

System.Diagnostics.ConditionalAttribute
  Constructor called=Void .ctor(System.String)
  Positional arguments passed to constructor:
    Type=System.String, Value=Release
  Named arguments set after construction: None

```

System.Diagnostics.ConditionalAttribute

```

  Constructor called=Void .ctor(System.String)
  Positional arguments passed to constructor:
    Type=System.String, Value=Debug
  Named arguments set after construction: None

```

Attributes applied to Main:

```

System.CLSCompliantAttribute

```

```
Constructor called=Void .ctor(Boolean)
Positional arguments passed to constructor:
  Type=System.Boolean, Value=True
Named arguments set after construction: None
```

```
System.SThreadAttribute
Constructor called=Void .ctor()
Positional arguments passed to constructor: None
Named arguments set after construction: None
```

Attributes applied to .ctor: None

Условные атрибуты

Программисты все чаще используют атрибуты благодаря простоте их создания, применения и отражения. Атрибуты позволяют также снабдить код аннотациями, одновременно реализуя другие богатые возможности. В последнее время атрибуты часто используются при написании и отладке кода. Например, инструмент утилиты кода для Microsoft Visual Studio (FxCopCmd.exe) предлагает атрибут `System.Diagnostics.CodeAnalysis.SuppressMessageAttribute`, применимый к типам и членам и подавляющий сообщения о нарушении правила определенного инструмента статического анализа. Этот атрибут ищется только кодом анализирующего инструмента, при нормальном ходе выполнения программы он просто не требуется. Если вы не анализируете код, наличие в метаданных атрибута `SuppressMessage` просто увеличивает объем метаданных, что не лучшим образом сказывается на производительности. Соответственно, хотелось бы, чтобы компилятор задействовал этот атрибут только в случаях, когда вы собираетесь воспользоваться инструментом анализа кода.

К счастью, вам на помощь приходит класс *условных атрибутов* (conditional attribute). Так называют класс, к которому применен атрибут `System.Diagnostics.ConditionalAttribute`. Например:

```
// #define TEST
// #define VERIFY

using System;
using System.Diagnostics;

[Conditional("TEST")] [Conditional("VERIFY")]
public sealed class CondAttribute : Attribute {
}

[Cond]
public sealed class Program {
```

продолжение ➤

```
public static void Main() {  
    Console.WriteLine("CondAttribute is {0}applied to Program type.",  
        Attribute.IsDefined(typeof(Program),  
            typeof(CondAttribute)) ? "" : "not ");  
}
```

Обнаружив, что был применен экземпляр `CondAttribute`, компилятор помещает в метаданные информацию об атрибуте, только если при компиляции кода был определен идентификатор `TEST` или `VERIFY`. При этом метаданные определения и реализации класса атрибута все равно останутся в сборке.

Глава 19. Null-совместимые значимые типы

Как известно, переменная значимого типа не может иметь значения `null`; ее содержимым всегда является значение соответствующего типа. Именно поэтому типы и называют значимыми. Но для некоторых сценариев такой подход является проблемой. Например, при проектировании базы данных тип данных столбца можно определить как 32-разрядное целое, что в FCL соответствует типу `Int32`. Однако в столбце базы может отсутствовать значение, что соответствует значению `null`. И это — вполне стандартная ситуация. А значит, работа с базами данных средствами .NET Framework затруднена. Ведь общезыковая среда (CLR) не позволяет представить значение типа `Int32` как `null`.

ПРИМЕЧАНИЕ

Адаптеры таблиц Microsoft ADO.NET поддерживают типы, допускающие присвоение значений `null`. Но, к сожалению, типы в пространстве имен `System.Data.SqlTypes` не замещаются null-совместимыми типами отчасти из-за отсутствия однозначного соответствия между ними. К примеру, тип `SqlDecimal` допускает максимум 38 разрядов, в то время как обычный тип `Decimal` — только 29. А тип `SqlString` поддерживает собственные региональные стандарты и порядок сравнения, чего не скажешь о типе `String`.

Вот еще один пример: в Java класс `java.util.Date` относится к ссылочным типам, а значит, его переменные допускают присвоение значения `null`. В то же время в CLR тип `System.DateTime` является значимым и подобного присвоения не допускает. Если написанному на Java приложению потребуется передать информацию о дате и времени веб-службе на платформе CLR, возможны проблемы. Ведь если Java-приложение отправит значение `null`, CLR просто не будет знать, что с ним делать.

Чтобы исправить ситуацию, в Microsoft разработали для CLR *null-совместимые значимые типы* (*nullable value type*). Чтобы понять, как они работают, познакомимся с определенным в FCL классом `System.Nullable<T>`. Вот логическое представление реализации этого класса:

```
[Serializable, StructLayout(LayoutKind.Sequential)]  
public struct Nullable<T> where T : struct {
```

продолжение ➤

```
// Эти два поля представляют состояние
private Boolean hasValue = false; // Предполагается наличие null
internal T value = default(T);    // Предполагается, что все биты
                                   // равны нулю

public Nullable(T value) {
    this.value = value;
    this.hasValue = true;
}

public Boolean HasValue { get { return hasValue; } }

public T Value {
    get {
        if (!hasValue) {
            throw new InvalidOperationException(
                "Nullable object must have a value.");
        }
        return value;
    }
}

public T GetValueOrDefault() { return value; }

public T GetValueOrDefault(T defaultValue) {
    if (!HasValue) return defaultValue;
    return value;
}

public override Boolean Equals(Object other) {
    if (!HasValue) return (other == null);
    if (other == null) return false;
    return value.Equals(other);
}

public override int GetHashCode() {
    if (!HasValue) return 0;
    return value.GetHashCode();
}

public override string ToString() {
    if (!HasValue) return "";
    return value.ToString();
}

public static implicit operator Nullable<T>(T value) {
    return new Nullable<T>(value);
}
```

```
public static explicit operator T(Nullable<T> value) {  
    return value.Value;  
}  
}
```

Как видите, этот класс реализует значимый тип, который может принимать значение null. Так как `Nullable<T>` также относится к значимым типам, его экземпляры достаточно производительны, поскольку экземпляры могут размещаться в стеке, а их размер совпадает с размером исходного типа, к которому приплюсован размер поля типа `Boolean`. Имейте в виду, что в качестве параметра `T` типа `Nullable` могут фигурировать только структуры. Дело в том, что переменные ссылочного типа и так могут принимать значение `null`.

Чтобы использовать в коде null-совместимый тип `Int32`, можно написать:

```
Nullable<Int32> x = 5;  
Nullable<Int32> y = null;  
Console.WriteLine("x: HasValue={0}, Value={1}", x.HasValue, x.Value);  
Console.WriteLine("y: HasValue={0}, Value={1}",  
    y.HasValue, y.GetValueOrDefault());
```

После компиляции и запуска этого кода получаем:

```
x: HasValue=True, Value=5  
y: HasValue=False, Value=0
```

Поддержка в C# null-совместимых значимых типов

В приведенном фрагменте кода для инициализации двух переменных `x` и `y` типа `Nullable<Int32>` используется достаточно простой синтаксис. Дело в том, что разработчики C# старались интегрировать в язык null-совместимые значимые типы, сделав их полноправными членами соответствующего семейства типов. В настоящее время C# предлагает достаточно удобный синтаксис для работы с такими типами. Переменные `x` и `y` можно объявить и инициализировать прямо в коде, воспользовавшись знаком вопроса:

```
Int32? x = 5;  
Int32? y = null;
```

В C# запись `Int32?` аналогична записи `Nullable<Int32>`. При этом вы можете выполнять преобразования, а также приведение null-совместимых экземпляров к другим типам. Язык C# поддерживает возможность применения операторов к экземплярам null-совместимых значимых типов. Вот несколько примеров.


```
private static void ConversionsAndCasting() {
    // Неявное преобразование из типа Int32 в Nullable<Int32>
    Int32? a = 5;

    // Неявное преобразование из 'null' в Nullable<Int32>
    Int32? b = null;

    // Явное преобразование Nullable<Int32> в Int32
    Int32 c = (Int32) a;

    // Прямое и обратное приведение элементарного типа
    // в null-совместимый тип
    Double? d = 5; // Int32->Double? (d is 5.0 as a double)
    Double? e = b; // Int32?->Double? (e is null)
}
```

Еще C# позволяет применять операторы к экземплярам null-совместимых типов. Вот несколько примеров:

```
private static void Operators() {
    Int32? a = 5;
    Int32? b = null;

    // Унарные операторы (+ ++ - -- ! ~)
    a++; // a = 6
    b = -b; // b = null

    // Бинарные операторы (+ - * / % & | ^ << >>)
    a = a + 3; // a = 9
    b = b * 3; // b = null;

    // Операторы равенства (== !=)
    if (a == null) { /* no */ } else { /* yes */ }
    if (b == null) { /* yes */ } else { /* no */ }
    if (a != b) { /* yes */ } else { /* no */ }

    // Операторы сравнения (< <= >=)
    if (a < b) { /* no */ } else { /* yes */ }
}
```

Вот как эти операторы интерпретирует C#:

- ❑ **Унарные операторы** (+++, -, --, !, ~). Если операнд равен null, результат тоже равен null.
- ❑ **Бинарные операторы** (+, -, *, /, %, &, |, ^, <<, >>). Результат равен значению null, если этому значению равен хотя бы один операнд. Исключением является случай воздействия операторов & и | на логический операнд ?. В результате поведение этих двух операторов совпадает с троичной логикой

языка SQL. Если ни один из операндов не равен null, операция проходит в обычном режиме, если же оба операнда равны null, в результате получаем null. Особая ситуация возникает в случае, когда значению null равен только один из операндов. В следующей таблице показаны возможные результаты, которые эти операторы дают для всех возможных комбинаций значений true, false и null.

Операнд 1 → Операнд 2 ↓	true	false	null
True	& = true = true	& = false = true	& = null = true
False	& = false = true	& = false = false	& = false = null
Null	& = null = true	& = false = null	& = null = null

- ❑ **Операторы равенства** (==, !=). Если оба операнда имеют значение null, они равны. Если только один из них имеет это значение, операнды не равны. Если ни один из них не равен null, операнды сравниваются на предмет равенства.
- ❑ **Операторы сравнения** (<, >, <=, >=). Если значение null имеет один из операндов, в результате получаем значение false. Если ни один из операндов не имеет значения null, следует сравнить их значения.

Операции с экземплярами null-совместимых типов приводят к разрастанию кода. К примеру, рассмотрим метод:

```
private static Int32? NullableCodeSize(Int32? a, Int32? b) {  
    return a + b;  
}
```

В результате компиляции создается большой объем IL-кода, что отрицательно сказывается на производительности. Вот эквивалент этого кода на C#:

```
private static Nullable<Int32> NullableCodeSize(Nullable<Int32> a,  
  
    Nullable<Int32> b) {  
    Nullable<Int32> nullable1 = a;  
    Nullable<Int32> nullable2 = b;  
    if (!(nullable1.HasValue & nullable2.HasValue)) {  
        return new Nullable<Int32>();  
    }  
    return new Nullable<Int32>(  
        nullable1.GetValueOrDefault() + nullable2.GetValueOrDefault());  
}
```

Напоследок напомним о возможности определения ваших собственных значимых типов, перегружающих упомянутые операторы. О том, как именно это осуществляется, мы говорили в главе 8. Если воспользоваться допускающим null-совместимым экземпляром вашего собственного значимого типа, компилятор поймет это правильно и вызовет перегруженный оператор. Предположим, у нас есть значимый тип `Point`, следующим образом описывающий перегрузку операторов `==` и `!=`:

```
using System;
```

```
internal struct Point {  
    private Int32 m_x, m_y;  
    public Point(Int32 x, Int32 y) { m_x = x; m_y = y; }  
  
    public static Boolean operator==(Point p1, Point p2) {  
        return (p1.m_x == p2.m_x) && (p1.m_y == p2.m_y);  
    }  
  
    public static Boolean operator!=(Point p1, Point p2) {  
        return !(p1 == p2);  
    }  
}
```

Воспользовавшись в этот момент null-совместимыми экземплярами типа `Point`, вы заставите компилятор вызвать перегруженные операторы:

```
internal static class Program {  
    public static void Main() {  
        Point? p1 = new Point(1, 1);  
        Point? p2 = new Point(2, 2);  
  
        Console.WriteLine("Are points equal? " + (p1 == p2).ToString());  
        Console.WriteLine("Are points not equal? " + (p1 != p2).ToString());  
    }  
}
```

После запуска этого кода я получил следующий результат:

```
Are points equal? False  
Are points not equal? True
```

Оператор объединения null-совместимых значений

В C# существует *оператор объединения null-совместимых значений* (null-coalescing operator). Он обозначается знаком `??` и работает с двумя операндами.

Если левый операнд не равен `null`, оператор возвращает его значение. В противном случае возвращается значение правого операнда. Оператор объединения null-совместимых значений удобен при задании предлагаемого по умолчанию значения переменной.

Основным преимуществом этого оператора является поддержка как ссылочных типов, так и null-совместимых значимых типов. Вот код, демонстрирующий его работу:

```
private static void NullCoalescingOperator() {  
    Int32? b = null;  
    // Приведенная далее инструкция эквивалентна следующей:  
    // x = (b.HasValue) ? b.Value : 123  
    Int32 x = b ?? 123;  
    Console.WriteLine(x); // "123"  
    // Приведенная далее в инструкции строка эквивалентна следующему коду:  
    // String temp = GetFilename();  
    // filename = (temp != null) ? temp : "Untitled";  
    String filename = GetFilename() ?? "Untitled";  
}
```

Некоторые пользователи считают оператор объединения null-совместимых значений всего лишь синтаксическим сокращением для оператора `?`. Однако оператор `??` позволяет получить два синтаксических преимущества.

Во-первых, он лучше работает с выражениями:

```
Func<String> f = () => SomeMethod() ?? "Untitled";
```

Прочитать и понять эту строку намного проще, чем следующий фрагмент кода, требующий присвоения переменных и использования нескольких операторов:

```
Func<String> f = () => { var temp = SomeMethod();  
    return temp != null ? temp : "Untitled";};
```

Во-вторых, оператор `??` упрощает сложные сценарии. Например:

```
String s = SomeMethod1() ?? SomeMethod2() ?? "Untitled";
```

Ведь эту строку прочитать и понять гораздо проще, чем следующий фрагмент кода:

```
String s;  
var sm1 = SomeMethod1();  
if (sm1 != null) s = sm1;  
else {  
    var sm2 = SomeMethod2();  
    if (sm2 != null) s = sm2;  
    else s = "Untitled";  
}
```

Поддержка в CLR null-совместимых значимых типов

В CLR существует встроенная поддержка null-совместимых значимых типов. Она предусматривает упаковку и распаковку, а также вызов метода `GetType` и интерфейсных методов. Все это призвано обеспечить более тесную интеграцию null-совместимых значимых типов в CLR. В результате типы ведут себя более естественно и лучше соответствуют ожиданиям большинства разработчиков. Рассмотрим процедуру поддержки этих типов в CLR более подробно.

Упаковка null-совместимых значимых типов

Представим переменную типа `Nullable<Int32>`, которой логически присваивается значение `null`. Для передачи этой переменной методу, ожидающему ссылки на тип `Object`, ее следует упаковать и передать методу ссылку на упакованный тип `Nullable<Int32>`. Однако при этом в метод будет передано отличное от `null` значение, несмотря на то что тип `Nullable<Int32>` содержит `null`. Эта проблема решается в CLR при помощи специального кода, который при упаковке null-совместимых типов создает иллюзию их принадлежности к обычным типам.

При упаковке экземпляра `Nullable<T>` проверяется его равенство значению `null` и в случае положительного результата вместо упаковки возвращается значение `null`. В противном случае CLR упаковывает значение экземпляра. Другими словами, тип `Nullable<Int32>` со значением 5 упаковывается в тип `Int32` с аналогичным значением. Вот код, демонстрирующий такое поведение:

```
// После упаковки Nullable<T> возвращается null или упакованный тип T
Int32? n = null;
Object o = n; // o равно null
Console.WriteLine("o is null={0}", o == null); // "True"
```

```
n = 5;
o = n; // o ссылается на упакованный тип Int32
Console.WriteLine("o's type={0}", o.GetType()); // "System.Int32"
```

Распаковка null-совместимых значимых типов

В CLR упакованный значимый тип `T` распаковывается обратно в `T` или в `Nullable<T>`. Если ссылка на упакованный значимый тип равна `null` и выполняется распаковка в тип `Nullable<T>`, CLR присваивает `Nullable<T>` значение `null`. Вот пример, иллюстрирующий сказанное:

```
// Создание упакованного типа Int32
Object o = 5;
```

```
// Распаковка этого типа в Nullable<Int32> и в Int32
Int32? a = (Int32?) 0; // a = 5
Int32 b = (Int32) 0; // b = 5
```

```
// Создание ссылки, инициализированной значением null
o = null;
```

```
// Распаковка ее в Nullable<Int32> и в Int32
a = (Int32?) 0; // a = null
b = (Int32) 0; // NullReferenceException
```

Вызов метода GetType через null-совместимый значимый тип

При вызове метода GetType для объекта типа Nullable<T> CLR возвращает тип T вместо Nullable<T>. Вот демонстрирующий данное поведение код:

```
Int32? x = 5;
```

```
// Эта строка выводит "System.Int32", а не "System.Nullable<Int32>"
Console.WriteLine(x.GetType());
```

Вызов интерфейсных методов через null-совместимый значимый тип

В приведенном далее фрагменте кода переменная n типа Nullable<Int32> приводится к интерфейсному типу IComparable<Int32>. Но тип Nullable<T> в отличие от типа Int32 не реализует интерфейс IComparable<Int32>. Тем не менее код успешно компилируется, а механизм верификации CLR считает, что код прошел проверку. Это нужно для предоставления более удобного синтаксиса.

```
Int32? n = 5;
Int32 result = ((IComparable) n).CompareTo(5); // Компилируется
                                                // и выполняется
Console.WriteLine(result); // 0
```

Без подобной поддержки со стороны CLR пришлось бы писать громоздкий код вызова интерфейсного метода через null-совместимый значимый тип. Для вызова метода потребовалось бы приведение распакованного значимого типа перед приведением к интерфейсу:

```
Int32 result = ((IComparable) (Int32) n).CompareTo(5); // Громоздкий код
```


ЧАСТЬ IV

Ключевые механизмы

Глава 20. Исключения и управление состоянием	492
Глава 21. Автоматическое управление памятью (сборка мусора)	552
Глава 22. Хостинг CLR и домены приложений	633
Глава 23. Загрузка сборок и отражение	665
Глава 24. Сериализация	709

Глава 20. Исключения и управление состоянием

Эта глава посвящена *обработке исключений* (exception handling), хотя мы будем касаться и других тем. Процедура обработки исключения состоит из нескольких этапов. Для начала следует определить, что ошибка присутствует. Определив это, нужно выяснить обстоятельства ее возникновения и решить, как от нее избавиться.

На этом этапе возникает вопрос о состоянии системы, так как ошибки обычно возникают в самое неподходящее время. Скорее всего, ваш код в этот момент будет находиться в некоем переходном состоянии и вам потребуется вернуть его в состояние, существовавшее до возникновения ошибки. Разумеется, мы также выясним, каким образом код дает понять о том, что с ним что-то не так.

На мой взгляд, обработка исключений является самым слабым местом CLR, и именно поэтому разработчикам бывает так трудно писать управляемый код. В последнее время специалисты Microsoft внесли значительные улучшения в этот аспект, но до хорошей, надежной системы все еще достаточно далеко. О том, что именно было сделано в данном направлении, мы поговорим при рассмотрении необработанных исключений, областей ограниченного выполнения, контрактов кода, средств создания оболочек для исключений во время выполнения, неперехваченных исключений и т. п.

Определение «исключения»

Конструируя тип, мы заранее пытаемся представить, в каких ситуациях он будет использоваться. В качестве имени типа обычно выбирается существительное, например `FileStream` или `StringBuilder`. Затем задаются свойства, события, методы и т. п. Форма определения этих членов (типы данных свойств, параметры методов, возвращаемые значения и т. п.) становится программным интерфейсом типа. Именно члены определяют допустимые действия с типом и его экземплярами. Для их имен обычно выбираются глаголы, например `Read`, `Write`, `Flush`, `Append`, `Insert`, `Remove` и т. п. Если член не может решить возложенную на него задачу, должно вбрасываться исключение.

ВНИМАНИЕ

Исключением называется ситуация, когда член не в состоянии решить возложенную на него задачу.

Рассмотрим следующее определение класса:

```
internal sealed class Account {  
    public static void Transfer(Account from, Account to, Decimal amount) {  
        from -= amount;  
        to += amount;  
    }  
}
```

Метод `Transfer` принимает два объекта `Account` и значение `Decimal`, определяя, сколько средств переводится с одного счета на другой. Очевидно, что этот метод должен вычитать деньги с одного счета и прибавлять их к другому. Но есть ряд обстоятельств, которые могут помешать его работе. Например, аргумент `from` или `to` может иметь значение `null`; аргументы `from` или `to` могут ссылаться на не открытый счет; на счету, с которого предполагается взять деньги, может оказаться недостаточно средств; на целевом счету может оказаться так много денег, что перевод дополнительной суммы станет причиной переполнения; аргумент `amount` может быть равен 0, иметь отрицательное значение или иметь более двух знаков после запятой.

При вызове метода `Transfer` следует учитывать все перечисленные ситуации и при выявлении любой из них оповещать вызывающий код, вбрасывая исключение. Обратите внимание, что возвращаемое методом значение принадлежит к типу `void`. То есть метод `Transfer` просто завершает свою работу, если завершение происходит в обычном режиме, или вбрасывает исключение в противном случае.

Объектно-ориентированное программирование обеспечивает высокую эффективность труда разработчиков, так как позволяет писать, например, такой код:

```
Boolean f = "Jeff".Substring(1, 1).ToUpper().EndsWith("E"); // true
```

Здесь я реализую свои намерения, объединяя несколько операций¹. Этот код легко читается и редактируется, так как его назначение очевидно. Мы берем строку, выделяем ее часть, приводим символы этой части к верхнему регистру и смотрим, заканчивается ли выделенный фрагмент символом "E". При этом делается допущение, что все упомянутые операции успешно завершаются. Хотя, разумеется, от ошибок никто не застрахован. Соответственно, с ними нужно что-то делать. Существует множество объектно-ориентированных средств — конструкторы, инструменты просмотра/задания свойств, добавления/удаления событий, вызовы перегрузки операторов, вызовы операторов преобразования типа, — которые не умеют возвращать код ошибки. Но даже они должны каким-то способом сообщать о ее наличии. В .NET Framework и всех поддерживаемых этой платформой языках программирования для этой цели существует специальный механизм, называемый *обработкой исключений* (exception handling).

¹ Методы расширения в C# позволяют строить цепочки из целого набора методов.

ВНИМАНИЕ

Некоторые разработчики считают, что исключение связано с частотой возникновения некоторого явления. К примеру, разработчик метода чтения файла может сказать: «Читая файл, вы в итоге достигнете его конца. Так как это случается всегда, я заставлю мой метод Read в этот момент возвращать определенное значение. И тогда вбрасывать исключение не понадобится». Но так считает разработчик, создающий метод Read, а не тот, кто этим методом потом пользуется.

На момент создания метода невозможно предугадать все ситуации, в которых он будет вызываться. Соответственно, нельзя предсказать, насколько частыми станут попытки прочитать файл до конца. Более того, так как большинство файлов содержит структурированные данные, вряд ли чтение последних фрагментов будет частым событием.

Механика обработки исключений

В этом разделе рассмотрены конструкции языка C#, предназначенные для обработки исключений, хотя мы и не будем особо вдаваться детали. Вам важно понять, когда и каким образом применяется обработка исключений. Подробную же информацию по данной теме вы найдете в документации на .NET Framework и спецификации языка C#. Следует также упомянуть, что в основе обработки исключений в .NET Framework лежит *структурная обработка исключений* (Structured Exception Handling, SEH) Windows. SEH рассматривается во многих источниках, в том числе в моей книге «Windows via C/C++» (Microsoft Press, 2007).

Рассмотрим код, демонстрирующий стандартное применение механизма обработки исключений. Он дает представление о виде и предназначении блоков обработки исключений. В комментариях дано формальное описание блоков try, catch и finally.

```
private void SomeMethod() {  
    try {  
        // В этот блок помещают код, требующий корректного восстановления  
        // или очистки ресурсов  
    }  
    catch (InvalidOperationException) {  
        // В такие блоки помещают код, который восстанавливают  
        // после исключения InvalidOperationException  
    }  
    catch (IOException) {  
        // В такие блоки помещают код, который восстанавливают  
        // после исключения IOException  
    }  
}
```

```
catch {  
    // Здесь находится код, который восстанавливают после остальных исключений  
  
    // После перехвата исключений их обычно вбрасывают повторно  
    // Этот вопрос будет рассмотрен позже  
    throw;  
}  
finally {  
    // Здесь находится код, выполняющий очистку ресурсов  
    // после операций, начатых в блоке try. Этот код  
    // выполняется ВСЕГДА вне зависимости от наличия исключения  
}  
// Код, следующий за блоком finally, выполняется, если в блоке try  
// не вбрасывалось исключение или если исключение было перехвачено  
// блоком catch, а новое не вбрасывалось  
}
```

Мы рассмотрели один из возможных способов обработки исключений при помощи блоков. Обычно все выглядит намного проще. В большинстве случаев можно обойтись всего двумя блоками, например блоком `try` с соответствующим ему блоком `finally` или же парой `try` и `catch`. Такой сложный пример я выбрал для демонстрации возможных комбинаций.

Блок `try`

В блок `try` помещается код, требующий очистки ресурсов и/или восстановления после исключения. Код очистки содержится в блоке `finally`. В блоке `try` может располагаться также код, приводящий к вбрасыванию исключения. Код же восстановления вставляют в один или несколько блоков `catch`. Один блок `catch` соответствует одному событию, после которого по вашим предположениям может потребоваться восстановление приложения. Блок `try` должен быть связан хотя бы с одним блоком `catch` или `finally`; сам по себе он не имеет смысла.

ВНИМАНИЕ

Иногда разработчики задаются вопросом об объеме кода, которое можно поместить внутрь блока `try`. Ответ на этот вопрос зависит от управления состоянием. Если внутри блока `try` вы собираетесь выполнять набор операций, каждая из которых может стать причиной исключения одного и того же типа, но при этом способы обработки каждого исключения разные, имеет смысл создать для каждой операции собственный блок `try`.

Блок `catch`

В блок `catch` помещают код, который должен выполняться в ответ на исключение. Блок `try` может быть связан как с набором блоков `catch`, так и не

ассоциироваться ни с одним таким блоком. Если код в блоке `try` не становится причиной исключения, CLR никогда не переходит к выполнению кода в соответствующем блоке `catch`. Поток просто пропускает их, сразу переходя к коду блока `finally` (если таковой, конечно, существует). Выполнив код блока `finally`, поток переходит к инструкции, следующей за этим блоком.

Выражение в скобках после ключевого слова `catch` называется *типом исключения* (`catch type`). В C# эту роль играет тип `System.Exception` и его производные. В предыдущем примере первые два блока `catch` обрабатывали исключения типа `InvalidOperationException` (или их производные) и `IOException` (или, опять же, их производные). В последнем блоке (для которого не был явно указан тип исключения) обрабатывались все остальные виды исключений. Это эквивалентно наличию блока `catch` для исключений типа `System.Exception` при условии, что доступ к информации внутри скобок отсутствует.

ПРИМЕЧАНИЕ

При отладке в Microsoft Visual Studio для просмотра вброшенного исключения нужно добавить в окно контрольных значений специальную переменную `$exception`.

Просмотр блоков `catch` в CLR осуществляется сверху вниз, поэтому чем более специализированным является исключение, тем выше его следует поместить. Сначала следуют самые младшие потомки, потом — их базовые классы (если таковые имеются) и, наконец, — класс `System.Exception` (или блок с неуказанным типом исключений). В противном случае компилятор сообщит об ошибке, так как более узкоспециализированные блоки в таких условиях окажутся для него недостижимыми.

Вброшенное при выполнении кода блока `try` (или любого вызванного этим блоком метода) исключение инициирует поиск блоков `catch` соответствующего типа. При отсутствии совпадений CLR продолжает просматривать стек вызовов в поисках фильтра, который мог бы перехватить это исключение. Если при достижении вершины стека не обнаруживается блока `catch` нужного типа, исключение считается необработанным. Эту ситуацию мы рассмотрим чуть позже.

При обнаружении блока `catch` нужного типа CLR исполняет все внутренние блоки `finally`, начиная со связанного с блоком `try`, в котором было вброшено исключение, и заканчивая блоком `catch` нужного типа. При этом ни один блок `finally` не выполняется до завершения действий с блоком `catch`, обрабатывающим исключение.

После того как работа с кодом блоков `finally` будет закончена, исполняется код из обрабатывающего блока `catch`. Здесь выбирается способ восстановления после исключения. Затем можно выбрать один из трех вариантов действий:

- еще раз вбросить то же самое исключение, уведомляя о нем код, расположенный выше в стеке;

- ❑ вбросить исключение другого типа, передавая расположенному выше в стеке коду больше сведений об исключениях;
- ❑ позволить потоку покинуть блок `catch`.

О том, что делать в каждом из этих случаев, мы поговорим немного позже.

При выборе первого или второго варианта действий CLR работает по уже рассмотренной схеме: просматривает стек вызовов в поисках блока `catch`, тип которого соответствует типу сгенерированного исключения.

В последнем же случае происходит переход к блоку `finally` (если он, конечно, существует). После завершения в нем всех операций управление переходит к расположенной после блока `finally` инструкции. Если блок `finally` отсутствует, поток переходит к инструкции, расположенной за последним блоком `catch`.

В С# после типа перехватываемого исключения можно указать имя переменной, которая будет ссылаться на сгенерированный объект, потомок класса `System.Exception`. В коде блока `catch` эту переменную можно использовать для получения информации об исключении (например, о последовательности инструкций, приведших к исключению). Объект, на который ссылается переменная, в принципе можно редактировать, но я рекомендую рассматривать его как предназначенный только для чтения. Впрочем, подробный разговор о типе `Exception` и манипуляциях им вынесен в отдельный раздел.

ПРИМЕЧАНИЕ

Можно создать событие `FirstChanceException` класса `AppDomain` и получать информацию об исключениях еще до того, как CLR начнет искать их обработчики. Подробно эта тема рассматривается в главе 22.

Блок `finally`

Код блока `finally` выполняется всегда¹. Обычно этот код производит очистку после выполнения блока `try`. Если в блоке `try` был открыт некий файл, блок `finally` должен содержать закрывающий этот файл код:

```
private void ReadData(String pathname) {  
  
    FileStream fs = null;  
    try {  
        fs = new FileStream(pathname, FileMode.Open);  
        // Обработка данных в файле  
    }  
}
```

¹ Прерывание потока или выгрузка домена приложений является источником исключения `ThreadAbortException`, обеспечивающего выполнение блока `finally`. Если же поток прерывается функцией `TerminateThread` или методом `FailFast` класса `System.Environment`, блок `finally` выполняется. Разумеется, Windows производит очистку всех ресурсов, которые использовались прерванным процессом.

```
catch (IOException) {  
    // Код восстановления после исключения IOException  
}  
finally {  
    // Файл обязательно следует закрыть  
    if (fs != null) fs.Close();  
}  
}
```

Если код блока `try` выполняется без исключений, файл закрывается. Впрочем, поскольку даже исключение не помешает выполнению код в блоке `finally`, файл не останется открытым. А вот если поместить инструкцию закрытия файла после блока `finally`, в случае неперехваченного исключения файл останется открытым (до следующего прохода сборщика мусора).

Блок `try` может существовать и без блока `finally`, ведь иногда его код просто не требует последующей очистки. Однако если вы решили создать блок `finally`, его следует поместить после всех блоков `catch`. И помните, что одному `try` может соответствовать только один блок `finally`.

Достигнув конца блока `finally`, поток переходит к инструкции, расположенной после этого блока. Еще раз напоминаю, что в блок `finally` помещается код, осуществляющий очистку. И он должен выполнять только те действия, которые необходимы для отмены операций, начатых в блоке `try`. Код блоков `catch` и `finally` следует делать по возможности коротким (ограничиваясь одной или двумя строками) и работающим без исключений. Однако иногда случается так, что источником исключения становится код восстановления или код очистки. Обычно это указывает на наличие серьезных ошибок.

Если источником исключения становятся блоки `catch` или `finally`, CLR продолжает работу как в случае, когда исключение вбрасывается после блока `finally`. Просто при этом теряется информация о первом исключении, брошенном в блоке `try`. Скорее всего, это новое исключение останется необработанным. После этого CLR завершает процесс, ликвидировав все поврежденные состояния. Продолжение работы приложения в подобном случае привело бы к непредсказуемым результатам и, вероятно, к брешам в системе безопасности.

С моей точки зрения, для механизма обработки исключений следовало бы выбрать другие ключевые слова. Ведь программисту нужно всего лишь выполнить фрагмент кода. А если что-то пойдет не так, либо восстановить приложение после ошибки и двигаться дальше, либо вернуться в состояние до возникновения проблем и сообщить о неполадках. Программистам также нужна гарантированная очистка кода. Слева показан код, хороший с точки зрения компилятора, справа — вариант, который предпочел бы видеть я:

<pre>void Method() { try { ... } }</pre>	<pre>void Method() { try { ... } }</pre>
--	--

```
catch (XxxException) {  
    ...  
}  
catch (YyyException) {  
    ...  
}  
catch {  
    ...: throw;  
}  
finally {  
    ...  
}  
}  
  
handle (XxxException) {  
    ...  
}  
handle (YyyException) {  
    ...  
}  
compensate {  
    ...: throw;  
}  
cleanup {  
    ...  
}  
}
```

CLS-совместимые и CLS-несовместимые исключения

Все языки программирования, ориентированные на CLR, должны поддерживать создание объектов класса `Exception`, так как этого требует общеязыковая спецификация (Common Language Specification, CLS). Но на самом деле, CLR разрешает вбрасывать экземпляры любого типа, в результате в некоторых языках появляются несовместимые с CLS исключения типа `String`, `Int32` или `DateTime`. Компилятор C# разрешает вбрасывать только объекты, производные от класса `Exception`, в то время как в других языках это ограничение отсутствует.

Многие программисты не знают, что для получения исключения можно вбрасывать объект любого типа, поэтому они пользуются только объектами, производными от класса `Exception`. До выхода версии CLR 2.0 при помощи блоков `catch` перехватывались только CLS-совместимые исключения. Если метод на C# вызывал метод, написанный на другом языке, и тот вбрасывал CLS-несовместимое исключение, его было невозможно перехватить, что чревато нарушением защиты.

Начиная с версии 2.0, в CLR появился класс `RuntimeWrappedException`, определенный в пространстве имен `System.Runtime.CompilerServices`. Являясь производным от класса `Exception`, он представляет собой CLS-совместимый тип исключений. Этот класс обладает закрытым полем типа `Object`, к которому можно обратиться через предназначенное только для чтения свойство `WrappedException` того же класса. В CLR 2.0 при вбрасывании CLS-несовместимого исключения автоматически создается экземпляр класса `RuntimeWrappedException`, закрытому полю которого присваивается ссылка на брошенный объект. Таким способом несовместимые с CLS исключения превращаются в CLS-совместимые. В итоге любой код, умеющий перехватывать исключения типа `Exception`, будет перехватывать и все остальные исключения, что устраняет угрозу безопасности.

До версии 2.0 перехват CLS-несовместимых исключений осуществлялся с помощью примерно такого кода:

```
private void SomeMethod() {
    try {
        // Внутри блока try помещают код, требующий корректного
        // восстановления работоспособности или очистки ресурсов
    }
    catch (Exception e) {
        // До C# 2.0 этот блок перехватывал только CLS-совместимые исключения
        // В C# 2.0 этот блок научился перехватывать также
        // CLS-несовместимые исключения
        throw; // Повторное вбрасывание перехваченного исключения
    }
    catch {
        // Во всех версиях C# этот блок перехватывает
        // и совместимые, и не совместимые с CLS исключения
        throw; // Повторное вбрасывание перехваченного исключения
    }
}
```

Узнав, что CLR поддерживает теперь оба вида исключений, некоторые разработчики стали писать два блока catch (как показано в предыдущем фрагменте кода), чтобы перехватывать исключения обоих видов. Если этот код перекомпилировать для CLR 2.0, второй блок catch никогда не будет выполняться, а компилятор выдаст предупреждение (CS1058: предыдущий блок catch уже перехватывает все исключения. Все брошенные исключения окажутся в оболочке класса System.Runtime.CompilerServices.RuntimeWrappedException):

CS1058: A previous catch clause already catches all exceptions. All non-exceptions thrown will be wrapped in a System.Runtime.CompilerServices.RuntimeWrappedException

Есть два пути переноса кода более ранних версий в .NET Framework 2.0. Во-первых, можно объединить два блока catch. Именно так рекомендуется действовать. Однако можно также сообщить CLR, что код вашей сборки будет работать по «старым» правилам, то есть что блоки catch (Exception) не должны перехватывать экземпляры нового класса RuntimeWrappedException. Вместо этого, среда CLR должна изъять из оболочки CLS-несовместимый объект и вызывать ваш код только при наличии в нем блока catch, в котором не определено никакого типа. Чтобы сообщить CLR об этом, к сборке нужно применить экземпляр RuntimeCompatibilityAttribute, например, так:

```
using System.Runtime.CompilerServices;
[assembly:RuntimeCompatibility(WrapNonExceptionThrows = false)]
```

ПРИМЕЧАНИЕ

Этот атрибут оказывает влияние на всю сборку. В одной сборке нельзя совмещать исключения в оболочке и без нее. Нужно соблюдать особую осторожность при добавлении в сборку нового кода (который ожидает от CLR исключений в оболочке класса System.Runtime.CompilerServices.RuntimeWrappedException), где есть старый код (в котором CLR не помещает исключения в оболочку).

Класс System.Exception

CLR позволяет вбрасывать в качестве исключений экземпляры любого типа — от Int32 до String. Но в Microsoft решили, что не стоит заставлять все языки вбрасывать и перехватывать исключения произвольного типа. Соответственно, был создан тип System.Exception. Именно исключения этого типа и его производных должны перехватываться во всех CLS-совместимых языках программирования. CLS-совместимыми называются типы исключений, производные от типа System.Exception. Компиляторы C# и многих других языков позволяют коду вбрасывать только CLS-совместимые исключения.

Свойства типа System.Exception описаны в табл. 20.1. Писать код доступа к этим свойствам вам, скорее всего, никогда не придется. Их ищут в отчете отладчика или в аварийном дампе памяти после прекращения работы приложения из-за необработанного исключения.

Таблица 20.1. Открытые свойства типа System.Exception

Свойство	Доступ	Тип	Описание
Message	Только для чтения	String	Содержит текст, сообщающий о причине исключения. Если исключение не удастся обработать, сообщение записывается в журнал. Конечным пользователям это сообщение недоступно, поэтому его максимально насыщают техническими подробностями, которые могут оказаться полезными для разработчиков
Data	Только для чтения	IDictionary	Ссылка на набор пар «параметр-значение». Обычно, перед тем как вбросить исключение, код добавляет запись в этот набор. Перехватывающий исключение код запрашивает эти записи и использует полученную информацию для своей работы

продолжение ➤

Таблица 20.1 (продолжение)

Свойство	Доступ	Тип	Описание
Source	Чтение/ запись	String	Содержит имя сборки, вбросившей исключение
StackTrace	Только для чтения	String	Содержит имена и сигнатуры методов, вызов которых стал источником исключения. Очень полезное для отладки свойство
TargetSite	Только для чтения	MethodBase	Содержит имя метода, ставшего источником исключения
HelpLink	Только для чтения	String	Содержит адрес документации с информацией об исключении (например, file:///C:/MyApp/Help.htm#MyExceptionHelp). С точки зрения безопасности пользователи не должны видеть сведений о необработанных исключениях, поэтому данное свойство требуется редко
InnerException	Только для чтения	Exception	Если текущее исключение было вброшено в ходе обработки предыдущего, указывает на это предыдущее исключение. Обычно данное поле содержит значение null. Тип Exception содержит также открытый метод GetBaseException, просматривающий связанный список внутренних исключений и возвращающий самое первое из них

Хотелось бы подробнее поговорить о предназначенном только для чтения свойстве `StackTrace` класса `System.Exception`. Его может читать блок `catch` для получения информации о том, какой именно метод стал источником исключения. Эта информация может быть весьма ценной для поиска объекта, ставшего источником исключения, и последующего исправления кода. При обращении к этому свойству вы фактически обращаетесь к коду в CLR, поскольку свойство не просто возвращает строку. При создании объекта типа, производного от `Exception`, свойству `StackTrace` присваивается значение `null`. И соответственно, при попытке прочитать свойство вы получаете не результат трассировки стека, а `null`.

При появлении исключения CLR делает запись с указанием места его возникновения. Когда блок `catch` получает исключение, CLR записывает, где именно оно было обнаружено. Если внутри блока `catch` обратиться к свойству `StackTrace` объекта, сгенерированного при появлении исключения, реализующий это свойство код обратится к CLR, где и будет создана строка, содержащая имена всех методов от точки, в которой было вброшено исключение, до точки, где оно было перехвачено.

ВНИМАНИЕ

При вбрасывании исключения CLR обнуляет его начальную точку. То есть CLR запоминает только место появления самого последнего исключения.

Вот код, вбрасывающий то же исключение, которое было перехвачено, и заставляющий CLR обнулить начальную точку:

```
private void SomeMethod() {
    try { ... }
    catch (Exception e) {
        ...
        throw e; // CLR считает, что исключение возникло тут
                // FxCop сообщает об ошибке
    }
}
```

В противоположность этому, при повторном вбрасывании перехваченного исключения с помощью ключевого слова `throw` удаления из стека информации о начальной точке не происходит. Вот код, демонстрирующий это на практике:

```
private void SomeMethod() {
    try { ... }
    catch (Exception e) {
        ...
        throw; // CLR не меняет информацию о начальной точке исключения
              // FxCop НЕ сообщает об ошибке
    }
}
```

Разница между этими фрагментами кода только в восприятии CLR места вбрасывания исключения. К сожалению, при первом или повторном вбрасывании исключения Windows обнуляет стек с информацией о начальной точке. И в случае необработанного исключения в систему сбора информации об ошибках Windows уходят сведения о последнем вброшенном исключении, даже если CLR «знает», где именно было вброшено самое первое исключение. Это серьезно усложняет отладку приложений. Некоторым разработчикам подобная ситуация кажется совершенно недопустимой, поэтому они выбирают другой способ реализации кода, гарантирующий истинность информации о первоначальной точке возникновения исключения:

```
private void SomeMethod() {
    Boolean trySucceeds = false;
    try {
        ...
        trySucceeds = true;
    }
}
```

продолжение ➤

```
finally {  
    if (!trySucceeds) { /* код перехвата исключения */ }  
}  
}
```

Строка, возвращаемая свойством `StackTrace`, не включает в себя имен методов, расположенных в стеке вызова выше точки принятия исключения блоком `catch`. Для отслеживания всего стека с самого начала до момента обработки исключения используйте тип `System.Diagnostics.StackTrace`. Он содержит свойства и методы, дающую разработчикам возможность программно управлять трассировкой стека и составляющими его кадрами.

Существует несколько конструкторов, позволяющих получить объект `StackTrace`. Некоторые из них строят кадры от начала потока до момента появления объекта `StackTrace`. Другие — инициализируют кадры объекта `StackTrace`, передавая ему в качестве аргумента объект, производный от типа `Exception`.

Если CLR обнаруживает для ваших сборок символы отладки (находящиеся в файлах с расширением `pdb`) свойство `StackTrace` класса `System.Exception` или `System.Diagnostics` возвращает строку. Метод `ToString` свойства `StackTrace` содержит пути к файлу в исходном коде и номера строк. А это крайне полезная для отладки информация.

В результатах трассировки стека можно обнаружить, что имена некоторых из вызывавшихся методов отсутствуют. Такая ситуация может возникнуть по двум причинам. Во-первых, в стек записывается, куда вернулся поток, а не откуда он вышел. Во-вторых, JIT-компилятор может выполнять подстановку (`inline`) кода методов в вызывающий код, чтобы избежать слишком большого числа вызовов, и возвращать результат вызова только одного метода. Многие компиляторы (в том числе C#) предлагают переключатель командной строки `/debug`. При его использовании компилятор включает в результирующую сборку информацию, заставляющую JIT-компилятор прекратить подстановку методов. В результате трассировка стека становится более полной и полезной как инструмент отладки.

ПРИМЕЧАНИЕ

JIT-компилятор проверяет назначенный сборке атрибут `System.Diagnostics.DebuggableAttribute`. Компилятор C# назначает этот атрибут автоматически. Установка флага `DisableOptimizations` заставляет JIT-компилятор прекратить подстановку методов сборки. В C# флаг устанавливается переключателем командной строки `/debug`. Применяв к методу настраиваемый атрибут `System.Runtime.CompilerServices.MethodImplAttribute`, вы можете запретить подстановку как для отладочной, так и для рабочей конфигурации. Вот пример определения метода, запрещающего подстановку:

```
using System;  
using System.Runtime.CompilerServices;
```

```
internal sealed class SomeType {  
    [MethodImpl(MethodImplOptions.NoInlining)]  
    public void SomeMethod() {  
        ...  
    }  
}
```

Классы исключений, определенные в FCL

В библиотеке классов Framework Class Library определено множество типов исключений (являющихся потомками класса `System.Exception`). Типы, определенные в сборке `mscorlib.dll`, иллюстрирует показанная далее иерархия; другие сборки содержат еще больше типов исключений (эта иерархия получена при помощи приложения, демонстрирующегося в главе 23).

```
System.Exception  
    System.AggregateException  
    System.ApplicationException  
        System.Reflection.InvalidFilterCriteriaException  
        System.Reflection.TargetException  
        System.Reflection.TargetInvocationException  
        System.Reflection.TargetParameterCountException  
        System.Threading.WaitHandleCannotBeOpenedException  
    System.InvalidTimeZoneException  
    System.IO.IsolatedStorage.IsolatedStorageException  
    System.Runtime.CompilerServices.RuntimeWrappedException  
    System.SystemException  
        System.AccessViolationException  
        System.AppDomainUnloadedException  
        System.ArgumentException  
            System.ArgumentNullException  
            System.ArgumentOutOfRangeException  
        System.DuplicateWaitObjectException  
        System.Globalization.CultureNotFoundException  
        System.Text.DecoderFallbackException  
        System.Text.EncoderFallbackException  
    System.ArithmeticException  
        System.DivideByZeroException  
        System.NotFiniteNumberException  
        System.OverflowException  
    System.ArrayTypeMismatchException  
    System.BadImageFormatException  
    System.CannotUnloadAppDomainException
```

System.Collections.Generic.KeyNotFoundException
System.ContextMarshalException
System.DataMisalignedException
System.ExecutionEngineException
System.FormatException
 System.Reflection.CustomAttributeFormatException
System.IndexOutOfRangeException
System.InsufficientExecutionStackException
System.InvalidCastException
System.InvalidOperationException
 System.ObjectDisposedException
System.InvalidProgramException
System.IO.IOException
 System.IO.DirectoryNotFoundException
 System.IO.DriveNotFoundException
 System.IO.EndOfStreamException
 System.IO.FileLoadException
 System.IO.FileNotFoundException
 System.IO.PathTooLongException
System.MemberAccessException
 System.FieldAccessException
 System.MethodAccessException
 System.MissingMemberException
 System.MissingFieldException
 System.MissingMethodException
System.MulticastNotSupportedException
System.NotImplementedException
System.NotSupportedException
 System.PlatformNotSupportedException
System.NullReferenceException
System.OperationCanceledException
 System.Threading.Tasks.TaskCanceledException
System.OutOfMemoryException
 System.InsufficientMemoryException
System.RankException
System.Reflection.AmbiguousMatchException
System.Reflection.ReflectionTypeLoadException
System.Resources.MissingManifestResourceException
System.Resources.MissingSatelliteAssemblyException
System.Runtime.InteropServices.ExternalException
 System.Runtime.InteropServices.COMException
 System.Runtime.InteropServices.SEHException
System.Runtime.InteropServices.InvalidComObjectException
System.Runtime.InteropServices.InvalidOleVariantTypeException
System.Runtime.InteropServices.MarshalDirectiveException
System.Runtime.InteropServices.SafeArrayRankMismatchException
System.Runtime.InteropServices.SafeArrayTypeMismatchException
System.Runtime.Remoting.RemotingException

System.Runtime.Remoting.RemotingTimeoutException
System.Runtime.Remoting.ServerException
System.Runtime.Serialization.SerializationException
System.Security.Cryptography.CryptographicException
System.Security.Cryptography.CryptographicUnexpectedOperationException
System.Security.HostProtectionException
System.Security.Policy.PolicyException
System.Security.Principal.IdentityNotMappedException
System.Security.SecurityException
System.Security.VerificationException
System.Security.XmlSyntaxException
System.StackOverflowException
System.Threading.AbandonedMutexException
System.Threading.SemaphoreFullException
System.Threading.SynchronizationLockException
System.Threading.ThreadAbortException
System.Threading.ThreadInterruptedException
System.Threading.ThreadStartException
System.Threading.ThreadStateException
System.TimeoutException
System.TypeInitializationException
System.TypeLoadException
System.DllNotFoundException
System.EntryPointNotFoundException
System.TypeUnloadedException
System.UnauthorizedAccessException
System.Security.AccessControl.PrivilegeNotHeldException
System.Threading.LockRecursionException
System.Threading.Tasks.TaskSchedulerException
System.TimeZoneNotFoundException

Специалисты Microsoft хотели сделать тип `System.Exception` базовым для всех исключений, а два других типа, `System.SystemException` и `System.ApplicationException`, стали бы его непосредственными потомками. Кроме того, исключения, вброшенные CLR, стали бы производными от типа `SystemException`, в то время как исключения, появившиеся в приложениях, должны были наследовать от `ApplicationException`. Это дало бы возможность написать блок `catch`, перехватывающий как все CLR-исключения, так и все исключения приложений.

Однако на практике это правило соблюдается не полностью; некоторые исключения являются прямыми потомками типа `Exception` (`IsolatedStorageException`), некоторые CLR-исключения наследуют от типа `ApplicationException` (`TargetInvocationException`), а некоторые исключения приложений — от типа `SystemException` (`FormatException`). Из-за этой путаницы типы `SystemException` и `ApplicationException` не несут никакой особой смысловой нагрузки. В настоящее время в Microsoft подумывают вообще убрать их из иерархии классов исключений, но это невозможно, так как приведет к нарушению работы уже имеющихся приложений, в которых используются эти классы.

Вбрасывание исключений

При реализации своего метода нужно вбрасывать исключение, если метод не в состоянии решить возложенную на него задачу. При этом следует учесть два фактора.

Во-первых, следует понять, к какому производному от типа `Exception` типу будет относиться ваше исключение. Этот выбор должен быть осмысленным. Подумайте о том, каким образом код, расположенный выше по стеку вызовов, сможет получать информацию о неудачной работе метода, чтобы выполнить восстановительные операции. Можно воспользоваться для этой цели одним из типов, определенных в `FCL`, но может оказаться и так, что там пока отсутствует подходящий тип. А значит, вам потребуется определить собственный тип, производный от класса `System.Exception`.

Если вы собираетесь создать иерархию, постарайтесь, чтобы она содержала как можно меньшее число базовых классов. Дело в том, что базовые классы зачастую обрабатывают множество ошибок как одну, а это может быть опасно. Соответственно, никогда не следует создавать объекты `System.Exception`¹ и всегда нужно соблюдать максимальную осторожность при вбрасывании исключений базовых классов.

ВНИМАНИЕ

В данном случае приходится также иметь дело с ограничениями, связанными с поддержкой версий. Если определить новый тип исключения как производный от существующего, код, перехватывавший исключения старого типа, будет работать и с новым типом. В некоторых сценариях такое поведение требуется, в других — нет. Весь вопрос в том, каким образом код, перехватывающий исключения базового класса, реагирует на тип исключения и производные от него типы. Не ожидавший появления новых типов код может повести себя непредсказуемо и даже стать причиной бреши в системе безопасности. Определяющий новый тип исключения программист не может знать всех мест, в которых окажется перехваченное базовое исключение. Не осведомлен он и о способах его обработки. Поэтому принять однозначно верное решение в подобной ситуации, увы, невозможно.

Во-вторых, следует решить, какое строковое сообщение должно быть передано конструктору исключения. Вбрасывание исключения должно сопровождаться детальной информацией о том, почему метод не смог решить свою задачу. При обработке перехваченного исключения этого сообщения не видно. А вот для необработанных исключений эти сообщения регистрируются в журнале. Необработанное исключение указывает на дефект приложения, об искоренении которого должен позаботиться разработчик. Конечные пользователи не

¹ На самом деле класс `System.Exception` никогда не рассматривался как абстрактный, и код, пытающийся вбросить исключение данного типа, не будет компилироваться.

имеют доступа к исходному коду и не могут перекомпилировать программу. Соответственно, не видят они и данного сообщения. Поэтому туда смело можно помещать всю техническую информацию, необходимую для устранения дефекта. Более того, так как все разработчики должны понимать английский язык (ведь языки программирования, а также FCL-классы и FCL-методы написаны на английском), не имеет смысла локализовывать текст сообщения. Впрочем, если вы создаете библиотеку классов для разработчиков, говорящих на других языках, ничто не запрещает выполнить локализацию. Именно по этой причине Microsoft локализует сообщения исключений, вбрасываемых FCL.

Создание классов исключений

К сожалению, процедура создания классов исключений трудоемка и часто сопровождается ошибками. Дело в том, что все типы, производные от типа `Exception`, должны иметь возможность сериализоваться, а это означает, что они не могут выйти за границы домена приложений и их нельзя записывать в журнал или базу данных. Впрочем, особенности сериализации рассматриваются в главе 24, а пока для простоты я создал обобщенный класс `Exception<TExceptionArgs>`:

```
[Serializable]
public sealed class Exception<TExceptionArgs> : Exception, ISerializable
where TExceptionArgs : ExceptionArgs {

    private const String c_args = "Args"; // Для (де)сериализации
    private readonly TExceptionArgs m_args;

    public TExceptionArgs Args { get { return m_args; } }
    public Exception(String message = null, Exception innerException = null)
        : this(null, message, innerException) { }

    public Exception(TExceptionArgs args, String message = null,
        Exception innerException = null): base(message, innerException) {
        m_args = args; }

    // Конструктор для десериализации; так как класс запечатан, конструктор
    // закрыт. Для незапечатанного класса конструктор должен быть защищенным
    [SecurityPermission(SecurityAction.LinkDemand,
        Flags=SecurityPermissionFlag.SerializationFormatter)]
    private Exception(SerializationInfo info, StreamingContext context)
        : base(info, context) {
        m_args = (TExceptionArgs)info.GetValue(
            c_args, typeof(TExceptionArgs));
    }
}
```

продолжение ➤

```
// Метод для сериализации; он открыт из-за интерфейса ISerializable
[SecurityPermission(SecurityAction.LinkDemand,
Flags=SecurityPermissionFlag.SerializationFormatter)]
public override void GetObjectData(
SerializationInfo info, StreamingContext context) {
info.AddValue(c_args, m_args);
base.GetObjectData(info, context);
}

public override String Message {
get {
String baseMsg = base.Message;
return (m_args == null) ? baseMsg : baseMsg + " (
" + m_args.Message + ")";
}
}

public override Boolean Equals(Object obj) {
Exception<TExceptionArgs> other = obj as Exception<TExceptionArgs>;
if (obj == null) return false;
return Object.Equals(m_args, other.m_args) && base.Equals(obj);
}
public override int GetHashCode() { return base.GetHashCode(); }
}
```

Простым является и базовый класс `ExceptionArgs`, которым ограничен класс `TExceptionArgs`. Вот как он выглядит:

```
[Serializable]
public abstract class ExceptionArgs {
public virtual String Message { get { return String.Empty; } }
}
```

Имея эти два класса, я могу легко определять дополнительные классы исключений. Скажем, вот как выглядит определение типа исключения, указывающее на то, что диск заполнен:

```
[Serializable]
public sealed class DiskFullExceptionArgs : ExceptionArgs {
private readonly String m_diskpath; // закрытое поле, задается
// во время создания

public DiskFullExceptionArgs(String diskpath) { m_diskpath = diskpath; }

// Открытое предназначенное только для чтения свойство,
// которое возвращает поле
public String DiskPath { get { return m_diskpath; } }
```

```
// Переопределение свойства Message для включения в него нашего поля
public override String Message {
    get {
        return (m_diskpath == null) ? base.Message : "DiskPath=" + m_diskpath;
    }
}
```

И в случае если мне не потребуется включать в этот класс дополнительные данные, он будет выглядеть просто:

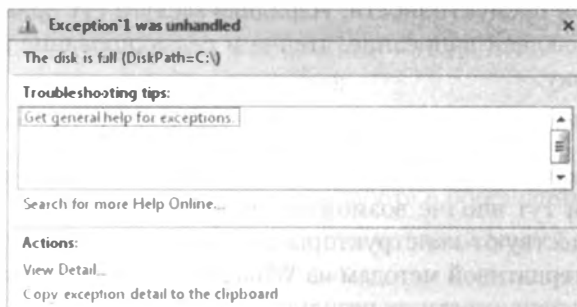
```
[Serializable]
public sealed class DiskFullExceptionArgs : ExceptionArgs { }
```

Теперь можно написать код, вбрасывающий и перехватывающий одно из исключений:

```
public static void TextException() {
    try {
        throw new Exception<DiskFullExceptionArgs>(
            new DiskFullExceptionArgs(@"C:\", "The disk is full");
    }
    catch (Exception<DiskFullExceptionArgs> e) {
        Console.WriteLine(e.Message);
    }
}
```

ПРИМЕЧАНИЕ

Хотелось бы сделать пару замечаний по поводу класса `Exception<TEExceptionArgs>`. Во-первых, любой определенный с его помощью тип исключения будет производным от `System.Exception`. Это не проблема для большинства сценариев, более того, даже предпочтительно использовать широкую иерархию типов. Во-вторых, диалоговое окно, появляющееся в Visual Studio при наличии необработанных исключений, не отображает параметр обобщенного типа `Exception<T>`, как показано на следующем рисунке.



Продуктивность вместо надежности

Я начал заниматься программированием в 1975 году. Написав изрядное количество программ на языке BASIC, я заинтересовался аппаратным обеспечением и перешел на язык ассемблера. Еще через некоторое время я переключился на язык C, дающий доступ к аппаратному обеспечению на более высоком уровне абстракции и облегчающий программирование. Я писал код для операционных систем, для платформ, для библиотек. И всегда старался сделать мой код как можно менее объемным и как можно более быстрым, тем более что для хорошей работы приложения качественным должно быть не только само приложение, но и используемые им операционная система и библиотеки.

Также внимательно я относился к восстановлению после ошибок. Выделяя память (при помощи оператора `new` в C++ или методов `malloc`, `HeapAlloc`, `VirtualAlloc` и т. п.), я всегда проверял возвращаемое значение, чтобы убедиться, что памяти действительно хватает. При неудовлетворительном результате запроса я программировал обходной путь, гарантируя, что состояние остальной части программы останется незатронутым и что вызывающая сторона получит сообщение об ошибке и сможет принять меры по ее коррекции.

По причинам, в которые я не могу вдаваться, при написании кода .NET Framework подобный подход не практиковался. Поэтому всегда есть вероятность столкнуться с недостаточным объемом памяти, к тому же я практически никогда не видел блока `catch` с кодом восстановления после исключения `OutOfMemoryException`. Более того, мне встречались разработчики, утверждавшие, что CLR не позволяет программам перехватывать это исключение. Разумеется, это неправда. На самом деле при выполнении управляемого кода возможны различные ошибки, но я еще не сталкивался с разработчиками, которые писали бы код для восстановления после потенциальных сбоев. В этом разделе мы поговорим как раз о таких сбоях. А также о том, почему приемлемым считается игнорировать такие ситуации. Кроме того, я опишу несколько проблем, которые могут возникнуть, если не обращать внимания на эти сбои, и предложу пути решения.

Объектно-ориентированное программирование позволяет добиться от разработчиков высокой продуктивности. Изрядная заслуга тут принадлежит совместимости, облегчающей написание, чтение и редактирование кода. Например, рассмотрим строку:

```
Boolean f = "Jeff".Substring(1, 1).ToUpper().EndsWith("E");
```

При ее написании мы предположили, что никаких ошибок не возникнет. Тем не менее ошибки тут вполне возможны, и нам нужен способ борьбы с ними. Для этого и существуют конструкторы и механизмы обработки исключений, являющиеся альтернативой методам из Win32 и COM, возвращающим значение `true` или `false` в зависимости от результата своей работы.

Продуктивность достигается не только благодаря совместимости, но и благодаря некоторым возможностям компиляторов. С их помощью вы можете:

- ❑ вставлять в вызываемый метод необязательные параметры;
- ❑ упаковывать экземпляры значимого типа;
- ❑ создавать и инициализировать массивы параметров;
- ❑ связываться с членами динамических переменных и выражений;
- ❑ связываться с методами расширения;
- ❑ связываться с перегруженными операторами и вызывать их;
- ❑ создавать делегаты;
- ❑ автоматически определять тип при вызове обобщенных методов, объявлении локальных переменных и использовании лямбда-выражений;
- ❑ определять и создавать замкнутые классы для лямбда-выражений и итераторов;
- ❑ определять, создавать и инициализировать анонимные типы и их экземпляры;
- ❑ писать код, поддерживающий язык встроенных запросов (Language Integrated Queries, LINQ).

Да и CLR делает многое для облегчения жизни программистов. К примеру, CLR позволяет:

- ❑ вызывать виртуальные и интерфейсные методы;
- ❑ загружать сборки и JIT-компилируемые методы, которые могут стать источником исключений `FileLoadException`, `BadImageFormatException`, `InvalidProgramException`, `FieldAccessException`, `MethodAccessException`, `MissingFieldException`, `MissingMethodException` и `VerificationException`;
- ❑ пересекать границы домена приложений для доступа к объектам типа, производного от `MarshalByRefObject`, которые могут стать источником исключения `AppDomainUnloadedException`;
- ❑ сериализовать и десериализовать объекты при пересечении границ домена приложений;
- ❑ заставлять поток вбрасывать исключение `ThreadAbortException` при вызове методов `Thread.Abort` и `AppDomain.Unload`;
- ❑ вызывать методы `Finalize`, чтобы сборщик мусора до освобождения памяти объекта выполнил завершающие операции;
- ❑ создавать типы в куче загрузчика при работе с обобщенными типами;
- ❑ вызывать статический конструктор типа, который может стать источником исключения `TypeInitializationException`;
- ❑ вбрасывать различные исключения, в том числе `OutOfMemoryException`, `DivideByZeroException`, `NullReferenceException`, `RuntimeWrappedException`,

TargetInvocationException, OverflowException, NotFiniteNumberException, ArrayTypeMismatchException, DataMisalignedException, IndexOutOfRangeException, InvalidCastException, RankException, SecurityException и многие другие.

И, разумеется, .NET Framework поставляется с объемной библиотекой классов, содержащих десятки тысяч типов, каждый из которых поддерживает общую, многократно используемую функциональность. Многие из этих типов предназначены для создания веб-приложений, веб-служб, приложений с богатым пользовательским интерфейсом, приложений для работы с системой безопасности, для управления изображениями, для распознавания речи. Этот список может быть бесконечным. И любой из кодов этих приложений может стать источником ошибки. В следующих версиях могут появиться новые типы исключений, наследующие от уже существующих. И ваши блоки catch начнут перехватывать исключения, о которых раньше и не подозревали.

Все это вместе — объектно-ориентированное программирование, средства компилятора, функциональность CLR и грандиозная библиотека классов — делают .NET Framework незаменимой платформой для разработки программного обеспечения¹. И с моей точки зрения, все это является потенциальным источником ошибок, которые мы практически не можем контролировать. Пока программа работает, все хорошо: мы легко пишем код, который также легко читается и редактируется. Но как только случается сбой, оказывается, что понять, где именно произошла ошибка и в чем ее причина, практически невозможно. Вот пример, иллюстрирующий вышесказанное:

```
private static Object OneStatement(Stream stream, Char charToFind) {
    return (charToFind + ": " + stream.GetType()
        + String.Empty + (stream.Position + 512M))
        .Where(c=>c == charToFind).ToArray();
}
```

Этот немного неестественный метод содержит всего одну инструкцию на C#, но эта инструкция призвана решить множество задач. Вот IL-код, сгенерированный компилятором C# для этого метода (места потенциальных сбоев, обусловленные неявно выполняемыми операциями, выделены полужирным шрифтом):

```
.method private hidebysig static object OneStatement(
    class [mscorlib]System.IO.Stream stream, char charToFind) cil managed {
    .maxstack 5
    .locals init (
        [0] class Program/<c__DisplayClass1 CS$<8__locals2,
        [1] object[] CS$0$0000)
```

¹ Мне следовало сказать, что столь привлекательной для разработчиков платформу делают еще и редактор Visual Studio, механизм IntelliSense, возможность работы с фрагментами кода, шаблоны, расширяемость, отладчик и многое другое. Но я вынес это за пределы основного обсуждения, так как эти средства не влияют на поведение кода во время выполнения.

```

L_0000: newobj instance void Program/(<>c__DisplayClass1::ctor()
L_0005: stloc.0
L_0006: ldloc.0
L_0007: ldarg.1
L_0008: stfld char Program/(<>c__DisplayClass1::charToFind
L_000d: ldc.i4.5
L_000e: newarr object
L_0013: stloc.1
L_0014: ldloc.1
L_0015: ldc.i4.0
L_0016: ldloc.0
L_0017: ldffd char Program/(<>c__DisplayClass1::charToFind
L_001c: box char
L_0021: stelem.ref
L_0022: ldloc.1
L_0023: ldc.i4.1
L_0024: ldstr ": "
L_0029: stelem.ref
L_002a: ldloc.1
L_002b: ldc.i4.2
L_002c: ldarg.0
L_002d: callvirt instance class [
mscorlib]System.Type [mscorlib]System.Object::GetType()
L_0032: stelem.ref
L_0033: ldloc.1
L_0034: ldc.i4.3
L_0035: ldsfld string [mscorlib]System.String::Empty
L_003a: stelem.ref
L_003b: ldloc.1
L_003c: ldc.i4.4
L_003d: ldc.i4 0x200
L_0042: newobj instance void [mscorlib]System.Decimal::ctor(int32)
L_0047: ldarg.0
L_0048: callvirt instance int64 [mscorlib]System.IO.Stream::get_Position()
L_004d: call valuetype [mscorlib]System.Decimal
[mscorlib]System.Decimal::op_Implicit(int64)
L_0052: call valuetype [mscorlib]System.Decimal [
mscorlib]System.Decimal::op_Addition (
valuetype [mscorlib]System.Decimal, valuetype [mscorlib]System.Decimal)
L_0057: box [mscorlib]System.Decimal
L_005c: stelem.ref
L_005d: ldloc.1
L_005e: call string [mscorlib]System.String::Concat(object[])
L_0063: ldloc.0
L_0064: ldftn instance bool Program/(<>c__DisplayClass1::<M>b__0(char)

```

продолжение ➤


```

L_006a: newobj instance
    void [mscorlib]System.Func`2<char, bool>::ctor(object, native int)
L_006f: call class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>
    [System.Core]System.Linq.Enumerable::Where<char>(
        class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>,
        class [mscorlib]System.Func`2<!!0, bool>)
L_0074: call !!0[] [System.Core]System.Linq.Enumerable::ToArray<char>
    (class [mscorlib]System.Collections.Generic.IEnumerable`1<!!0>)
L_0079: ret
}

```

Как видите, исключение `OutOfMemoryException` может быть выброшено в процессе создания класса `<c__DisplayClass1` (тип генерируется компилятором), массива `Object[]`, делегата `Func`, а также при упаковке типов `char` и `Decimal`. Внутреннее выделение памяти происходит при вызове методов `Concat`, `Where` и `ToArray`. Создание экземпляров типа `Decimal` может сопровождаться вызовом конструктора этого типа, что может стать причиной исключения `TypeInitializationException`¹. Затем неявно вызываются операторные методы `op_Implicit` и `op_Addition` типа `Decimal`, которые могут делать что угодно, в том числе и вбрасывать исключение `OverflowException`.

Интересен вызов свойства `Position` класса `Stream`. Начнем с того, что это виртуальное свойство, поэтому метод `OneStatement` не «знает», какой код на самом деле следует выполнять, что может привести к любому исключению. Кроме того, так как класс `Stream` происходит от класса `MarshalByRefObject`, аргумент `stream` может ссылаться на объект-представитель, сам являющийся ссылкой на объект другого домена приложений. Но другой домен приложений может оказаться выгруженным, что ведет к вбрасыванию исключения `AppDomainUnloadedException`.

При всем при этом я не имею возможности редактировать вызываемые мною методы, ведь они созданы в Microsoft. Зато не исключено, что в будущем специалисты Microsoft поменяют их реализацию, а это приведет к новым типам исключений, о которых я на момент написания метода `OneStatement` не подозревал. Существует ли способ сделать этот метод устойчивым по отношению к возможным ошибкам? Кстати, проблемной является и противоположная ситуация: блок `catch` способен перехватывать типы исключений, порожденные конкретным типом, поэтому возможно выполнение кода восстановления совсем для другой ошибки.

Теперь, получив информацию о возможных ошибках, вы, скорее всего, сами можете ответить на вопрос, почему стало возможным написание неустойчивого и ненадежного кода — просто это непрактично. Более того, порой это вообще невозможно. Кроме того, следует учитывать тот факт, что ошибки возникают

¹ Кстати, конструкторы классов для типов `System.Char`, `System.String`, `System.Type` и `System.IO.Stream` в этом приложении также могут стать причиной исключения `TypeInitializationException`, причем при аналогичных обстоятельствах.

относительно редко. И поэтому было решено пожертвовать надежностью кода в угоду продуктивности работы программистов.

Необработанные исключения заставляют приложение прекратить работу. И это здорово, так как позволяет выявить проблемы еще на стадии тестирования. Информации, которую вы при этом получаете (сообщение об ошибке и трассировка стека), обычно достаточно для внесения нужных исправлений. Разумеется, есть и фирмы, не желающие, чтобы их приложения закрывались в ходе тестирования или использования, поэтому разработчики вставляют код, перехватывающий исключение типа `System.Exception`, базового для всех типов исключений. Другое дело, что при таком подходе возможны ситуации, когда приложение продолжит работу с испорченным состоянием.

Чуть позже мы рассмотрим класс `Account`, определяющий метод `Transfer` для перевода денег с одного счета на другой. Представьте, что при вызове этого метода деньги успешно вычитаются с одного счета, но перед зачислением их на другой счет вбрасывается исключение. Если вызывающий код перехватывает исключения типа `System.Exception` и продолжает работу, состояние приложения окажется испорченным: как на счете `from`, так и на счете `to` будет меньше денег, чем там должно быть. А так как в данном случае мы говорим о деньгах, порча состояния рассматривается уже не как обычная ошибка, а как проблема системы безопасности приложения. Продолжая работу, приложение выполнит еще ряд переводов с одного счета на другой, а значит, порча состояния в рамках приложения продолжится.

Можно задаться вопросом, почему метод `Transfer` сам не может перехватывать исключения типа `System.Exception` и возвращать деньги на счет `from`. Это действительно сработает для более-менее простого метода `Transfer`. Но если он производит контрольный отчет изъятых денег или если со счетом одновременно работает несколько потоков, попытка отменить операцию уже не будет иметь успеха, а только приведет к новому исключению. А это значит, что ситуация с порчей состояния не улучшится, а только ухудшится.

ПРИМЕЧАНИЕ

Можно возразить, что информация о месте возникновения ошибки важнее сведений о ее содержании. Например, полезнее было бы знать, что перевод денег со счета не произошел, а не то, что метод `Transfer` не сработал из-за исключения `SecurityException` или `OutOfMemoryException`. На самом деле, модель обработки ошибок Win32 действует следующим способом: методы возвращают значение `true` или `false`, указывая на результативность своей работы. Это дает вам информацию о том, какой именно метод стал причиной проблемы. Затем, если в программе предусмотрен поиск информации о причинах сбоя, вызывается метод `GetLastError`. В классе `System.Exception` присутствует свойство `Source`, указывающее имя незавершенного метода. Но это свойство принадлежит к типу `String`, поэтому вам придется самостоятельно заниматься анализом полученных данных. К тому же в ситуации, когда два метода вызывают один и тот же метод, свойство `Source` не поможет вам понять, где именно произошел сбой. Вместо этого

вам придется анализировать строку, возвращенную свойством `StackTrace` класса `Exception`. Так как это — сложная задача, я пока еще не встречал программиста, написавшего подобный код.

Существуют несколько подходов, способных *помочь* сгладить проблему испорченного состояния:

- ❑ CLR запрещает аварийно завершать потоки во время выполнения кода блоков `catch` и `finally`. Поэтому сделать метод `Transfer` более устойчивым можно следующим способом:

```
public static void Transfer(Account from, Account to, Decimal amount) {  
    try { /* здесь ничего не делается */ }  
    finally {  
        from -= amount;  
        // Прерывание потока (из-за Thread.Abort/AppDomain.Unload)  
        // здесь невозможно  
        to += amount;  
    }  
}
```

Тем не менее я настоятельно не рекомендую помещать весь код внутри блоков `finally`! Этот прием можно использовать только для изменения самых чувствительных состояний.

- ❑ Класс `System.Diagnostics.Contracts.Contract` позволяет применять к методам контракты кода. Именно они позволяют проверять аргументы и другие переменные перед модификацией состояния с использованием этих аргументов/переменных. В случае соответствия контракту вероятность порчи состояния минимальна (но не невозможна). Если же проверка не проходит, сразу вбрасывается исключение. Впрочем, подробно эта тема рассматривается чуть позже.
- ❑ Области ограниченного исполнения (CER) дают возможность избежать имеющихся в CLR неоднозначностей. К примеру, перед входом в блок `try` можно загрузить все требуемые кодом соответствующих блоков `catch` и `finally` сборки. Кроме того, CLR скомпилирует весь код блоков `catch` и `finally`, включая вызываемые внутри этих блоков методы. Таким способом можно устранить множество потенциальных исключений (в том числе `FileLoadException`, `BadImageFormatException`, `InvalidProgramException`, `FieldAccessException`, `MethodAccessException`, `MissingFieldException` и `MissingMethodException`), которые могут возникнуть при попытке выполнения кода восстановления после ошибок (в блоках `catch`) или кода очистки (в блоке `finally`). Также это снизит вероятность появления исключения `OutOfMemoryException` и некоторых других. Области ограниченного исполнения также подробно обсуждаются в этой главе.
- ❑ В зависимости от местоположения состояния можно использовать транзакции, гарантирующие, что редактированию подвергаются либо все состояния,

либо ни одно из них. К примеру, если данные размещены в базе, транзакции работают удовлетворительно. Windows в настоящее время также поддерживает реестр транзакций и операции с файлами (только на разделе NTFS), так что вы можете воспользоваться этими функциями. К сожалению, в настоящее время в .NET Framework данная функциональность не представлена. Для приведения их в действие вам потребуется прибегнуть к явному вызову. Дополнительную информацию по данной теме вы найдете, читая документацию на класс `System.Transactions.TransactionScope`.

- ❑ Можно сделать методы более явными. К примеру, класс `Monitor` обычно используется для включения/отключения режима записи при синхронизации потока:

```
public static class SomeType {
    private static Object s_myLockObject = new Object();
    public static void SomeMethod () {
        Monitor.Enter(s_myLockObject); // В случае исключения произойдет ли
                                        // записи? Если да, то этот режим
                                        // будет невозможно отключить!

        try {
            // Безопасная в отношении потоков операция
        }
        finally {
            Monitor.Exit(s_myLockObject);
        }
    }
    // ...
}
```

Из-за описанных проблем не стоит использовать этот вариант перегрузки метода `Enter` класса `Monitor`. Лучше переписать код следующим образом:

```
public static class SomeType {
    private static Object s_myLockObject = new Object();
    public static void SomeMethod () {
        Boolean lockTaken = false; // Предполагаем, что записи нет
        try {
            // Это работает вне зависимости от наличия исключения!
            Monitor.Enter(s_myLockObject, ref lockTaken);
            // Потокбезопасная операция
        }
        finally {
            // Если режим записи включен, отключаем его
            if (lockTaken) Monitor.Exit(s_myLockObject);
        }
    }
    // ...
}
```

Хотя этот код стал более прозрачным, в случае запираания в рамках синхронизации потоков лучше вообще не прибегать к обработке исключений. Причины этого рассматриваются в главе 29.

Если обнаружится, что состояние осталось испорченным даже после восстановительных операций, его надлежит удалить, чтобы не создавать новых проблем. После этого приложение перезагружается, чтобы состояние инициализировалось нормально. Можно надеяться, что повреждений больше не будет. Управляемое состояние не может выходить за границы домена приложений. Поэтому для устранения испорченных состояний достаточно выгрузить домен приложения, воспользовавшись методом `Unload` класса `AppDomain` (детали см. в главе 22).

Если состояние кажется вам настолько плохим, что имеет смысл вообще завершить работу приложения, используйте статический метод `FailFast` класса `Environment`:

```
public static void FailFast(String message);  
public static void FailFast(String message, Exception exception);
```

Этот метод завершает процесс без выполнения активных блоков `try/finally` и без вызовов метода `Finalize`. Ведь выполнение дополнительного кода в поврежденном состоянии может ухудшить положение дел. Метод `FailFast` дает возможность очистить объекты, производные от класса `CriticalFinalizerObject`, с которыми мы познакомимся в главе 21. Это нормально, так как эти объекты стремятся просто закрыть исходные ресурсы. К тому же состояние `Windows`, скорее всего, останется в порядке даже при повреждении состояния `CLR` или вашего приложения. Метод `FailFast` записывает сообщение в журнал событий `Windows`, после чего включает это сообщение в отчет об ошибках. Затем он создает дампы памяти вашего приложения и завершает его работу.

ВНИМАНИЕ

По большей части `FCL`-код не гарантирует сохранения нормального состояния после неожиданного исключения. При обнаружении исключения, прошедшего через `FCL`-код, продолжение работы с `FCL`-объектами повышает вероятность их непредсказуемого поведения.

В этом разделе я попытался познакомить вас с возможными проблемами, связанными с механизмом обработки исключений в `CLR`. Большинство приложений не стоит использовать при повреждении их состояния, так как это ведет к появлению некорректных данных и даже брешам в системе безопасности. Если вы пишете приложение, которое не может аварийно завершать свою работу (например, операционную систему или ядро базы данных), не стоит использовать управляемый код. И так как система `Microsoft Exchange Server`

написана в основном средствами управляемого кода, для хранения электронной почты она задействует собственную базу данных. Эта база называется Extensible Storage Engine, она поставляется вместе с Windows и обычно располагается по адресу `C:\Windows\System32\EseNT.dll`. При желании вы можете воспользоваться этой базой для своих приложений.

Управляемый код желательно выбирать для приложений, которые могут прекращать работу при возникновении дефектов состояния. В эту категорию попадает множество приложений. Написать собственную устойчиво работающую библиотеку классов или приложение крайне сложно, именно поэтому в большинстве случаев разработчики предпочитают обходиться управляемым кодом, повышая продуктивность своей работы.

Приемы работы с исключениями

Понимать механизм обработки исключений важно, но не менее важно понимать, как их разумно использовать. Слишком часто я встречал библиотеки, перехватывающие все исключения без разбора и оставляя разработчика приложения в неведении о возникшем сбое. В этом разделе я предлагаю правила использования исключений, которые должен знать каждый разработчик.

ВНИМАНИЕ

Если вы — разработчик библиотек классов и занимаетесь созданием типов, которые будут использовать другие разработчики, отнеситесь к этим правилам очень серьезно. На вас лежит огромная ответственность за разработку интерфейса, применимого к широкому спектру приложений. Помните: вы не знаете всех тонкостей кода, который вызываете через делегаты, а также через виртуальные или интерфейсные методы.

Вы также не знаете, какой код будет вызывать вашу библиотеку. Нельзя предвидеть все ситуации, в которых может применяться ваш тип, поэтому не принимайте никаких политических решений. Ваш код не должен решать, что есть ошибка, а что нет — оставьте это решение вызывающему коду. Следуйте правилам, изложенным в этой главе, иначе разработчикам приложений при использовании типов вашей библиотеки классов придется не сладко.

Если вы — разработчик приложений, определяйте любую политику, какую сочтете нужной. Придерживаясь правил разработки, вы сможете быстрее выявлять и исправлять ошибки в своем коде, что повысит устойчивость ваших приложений. Однако вы вольны отходить от этих правил, если после тщательного обдумывания сочтете это необходимым. Политику приложения (например, более агрессивный перехват исключений кодом приложения) определяете именно вы.

Активно используйте блоки `finally`

По-моему, блоки `finally` — прекрасное средство! Они позволяют определять код, который будет гарантированно исполнен независимо от вида выброшенного потоком исключения. Блоки `finally` нужны, чтобы выполнить очистку после любой успешно начатой операции, прежде чем вернуть управление или продолжить исполнение кода, расположенного после них. Блоки `finally` также часто используют для явного уничтожения любых объектов во избежание утечки ресурсов. Вот пример, где в этом блоке размещен весь код, выполняющий очистку (закрывающий файл):

```
using System;
using System.IO;

public sealed class SomeType {
    private void SomeMethod() {
        // Открытие файла
        FileStream fs = new FileStream(@"C:\Data.bin ", FileMode.Open);
        try {
            // Вывод частного от деления 100 на первый байт файла
            Console.WriteLine(100 / fs.ReadByte());
        }
        finally {
            // В блоке finally размещается код очистки, гарантирующий
            // закрытие файла независимо от того, выброшено исключение
            // (например, если первый байт файла равен 0) или нет
            fs.Close();
        }
    }
}
```

Гарантировать исполнение кода очистки при любых обстоятельствах настолько важно, что большинство языков поддерживает соответствующие программные конструкции. Например, в C# при использовании инструкций `lock`, `using` и `foreach` блоки `try/finally` создаются автоматически. Компилятор строит эти блоки и при перекрытии деструктора класса (метод `Finalize`). При работе с упомянутыми конструкциями написанный вами код помещается в блок `try`, а код очистки — в блок `finally`. А именно:

- ❑ если вы используете инструкцию `lock`, то внутри блока `finally` отключается записание;
- ❑ если вы используете инструкцию `using`, то внутри блока `finally` для объекта вызывается метод `Dispose`;
- ❑ если вы используете инструкцию `foreach`, то внутри блока `finally` для объекта `IEnumerator` вызывается метод `Dispose`;
- ❑ если вы определяете деструктор, то внутри блока `finally` вызывается метод `Finalize` базового класса.

Например, в следующем коде на C# используются возможности инструкции `using`. Хотя этот фрагмент короче предыдущего, при обработке исходного текста этого и предыдущего примеров компилятор генерирует идентичный код:

```
using System;
using System.IO;

internal sealed class SomeType {
    private void SomeMethod() {
        using (FileStream fs =
            new FileStream(@"C:\Data.bin", FileMode.Open)) {
            // Вывод частного от деления 100 на первый байт файла
            Console.WriteLine(100 / fs.ReadByte());
        }
    }
}
```

Подробнее об инструкции `using` мы поговорим в главе 21, а об инструкции `lock` — в главе 29.

Не надо перехватывать все исключения

Распространенной ошибкой является слишком частое и неуместное использование блоков `catch`. Перехватывая исключение, вы тем самым заявляете, что ожидали его, понимаете его причины и знаете, как с ним разобраться. Другими словами, вы определяете политику для приложения. Эта тема подробно раскрыта в разделе «Продуктивность вместо надежности».

Тем не менее слишком часто приходится видеть примерно такой код:

```
try {
    // Попытка выполнить код, который, как считает программист,
    // может привести к сбою...
}
catch (Exception) {
    ...
}
```

Этот код демонстрирует, что в нем предусмотрены *все* исключения *любых* типов и он способен восстанавливаться после *любых* исключений в *любых* ситуациях. Разве это возможно? Тип из библиотеки классов ни в коем случае не должен перехватывать все исключения подряд: ведь он не может знать наверняка, как приложение должно реагировать на исключения. Кроме того, такой тип будет часто вызывать код приложения через делегат или виртуальный метод. Если в одной части приложения возникает исключение, то в другой части, вероятно, есть код, способный перехватить его. Исключение должно пройти через фильтр перехвата и быть передано вверх по стеку вызовов, чтобы код приложения смог обработать его как надо.

Если исключение осталось необработанным, CLR завершает процесс. О необработанных исключениях мы поговорим чуть позже. Большинство из них обнаруживаются на стадии тестирования. Для борьбы с ними следует либо заставить код реагировать на определенное исключение, либо переписать его, устранив условия, ставшие причиной сбоя. Число необработанных исключений в окончательной версии программы, предназначенной для выполнения в производственной среде, должно быть минимальным, а сама программа должна быть исключительно устойчивой.

ПРИМЕЧАНИЕ

В некоторых случаях метод, не способный решить поставленную задачу, обнаруживает, что состояние некоторых объектов испорчено и не поддается восстановлению. Если разрешить приложению продолжить работу, результаты могут оказаться плачевными, в том числе возможно нарушение безопасности. При обнаружении такой ситуации метод должен не вбрасывать исключение, а немедленно выполнять принудительное завершение процесса вызовом метода `FailFast` типа `System.Environment`.

Кстати, вполне допустимо перехватить исключение `System.Exception` и выполнить определенный код внутри блока `catch` при условии, что в конце этого кода исключение вбрасывается снова. Перехват и «проглатывание» (без повторного вбрасывания) исключения `System.Exception` недопустимо, так как приводит к сокрытию факта сбоя и продолжению работы приложения с непредсказуемыми результатами, что означает нарушение безопасности. Предоставляемая компанией Microsoft утилита `FxCop` позволяет находить блоки `catch (Exception)`, в коде которых отсутствует инструкция `throw`. Подробнее мы обсудим это далее в этой главе.

Наконец, допускается перехватывать исключение, возникшее в одном потоке, и повторно вбрасывать его в другом потоке. Такое поведение поддерживает модель асинхронного программирования (см. главу 27). Например, если поток из пула потоков выполняет код, который вызывал исключение, CLR перехватывает и игнорирует исключение, позволяя потоку вернуться в пул. Позже один из потоков должен вызвать метод `EndXxx`, чтобы выяснить результат асинхронной операции. Метод `EndXxx` вбрасывает такое же исключение, что и поток из пула, выполнявшего заданную работу. В данной ситуации исключение игнорируется первым потоком, но повторно вбрасывается потоком, вызывавшим метод `EndXxx`, в результате ошибка не оказывается скрытой от приложения.

Корректное восстановление после исключения

Иногда заранее известно, источником какого исключения может стать метод. Поскольку это — ожидаемое исключение, нужен код, обеспечивающий кор-

ректное восстановление приложения в такой ситуации и позволяющий ему продолжить работу. Вот пример (на псевдокоде):

```
public String CalculateSpreadsheetCell(Int32 row, Int32 column) {  
    String result;  
    try {  
        result = /* Код для расчета значения ячейки электронной таблицы */  
    }  
    catch (DivideByZeroException) {  
        result = "Нельзя отобразить значение: деление на ноль";  
    }  
    catch (OverflowException) {  
        result = "Нельзя отобразить значение: оно слишком большое";  
    }  
    return result;  
}
```

Этот псевдокод рассчитывает содержимое ячейки электронной таблицы и возвращает строку с ее значением вызывающему коду, который показывает его в окне приложения. Однако содержимое ячейки может быть частным от деления значений двух других ячеек. И если ячейка со знаменателем содержит 0, то CLR вбрасывает исключение `DivideByZeroException`. Тогда метод перехватывает именно это исключение и возвращает специальную строку, которая выводится пользователю. Аналогично содержимое ячейки может быть произведением двух других ячеек. Если полученное значение не умещается в отведенное число битов, CLR генерирует объект `OverflowException`, а также специальную строку для показа ее пользователю.

Перехватывая конкретные исключения, нужно полностью осознавать вызывающие их обстоятельства и знать типы исключений, производные от перехватываемого типа. Не следует перехватывать и обрабатывать тип `System.Exception` (без повторного вбрасывания), так как нельзя знать все возможные исключения, которые могут быть брошены внутри вашего блока `try` (особенно это касается типов `OutOfMemoryException` и `StackOverflowException`).

Отмена незавершенных операций при невозстановимых исключениях

Обычно для выполнения единственной абстрактной операции методу приходится вызывать несколько других методов, одни из которых могут завершаться успешно, а другие — нет. Допустим, происходит сериализация набора объектов в файл. После сериализации 10 объектов вбрасывается исключение (например, из-за переполнения диска или из-за отсутствия атрибута `Serializable` у следующего сериализуемого объекта). После этого исключение фильтруется и передается вызывающему методу, но в каком состоянии остается файл? Он оказывается поврежденным, так как в нем находится граф частично сериализованного объекта. Было бы здорово, если бы приложение могло отменить неза-

вершенные операции и вернуть файл в состояние, в котором он был до записи сериализованных объектов. Вот нужный фрагмент кода:

```
public void SerializeObjectGraph(FileStream fs,
    IFormatter formatter, Object rootObj) {

    // Сохранение текущей позиции в файле
    Int64 beforeSerialization = fs.Position;

    try {
        // Попытка сериализовать граф объекта и записать его в файл
        formatter.Serialize(fs, rootObj);
    }
    catch { // Перехват всех исключений
        // При ЛЮБОМ повреждении файл возвращается в нормальное состояние
        fs.Position = beforeSerialization;

        // Обрезаем файл
        fs.SetLength(fs.Position);

        // ПРИМЕЧАНИЕ: предыдущий код не помещен в блок finally,
        // так как сброс потока требуется только при сбое сериализации

        // Уведомляем вызывающий код о происходящем.
        // снова вбрасывая ТО ЖЕ САМОЕ исключение
        throw;
    }
}
```

Для корректной отмены незавершенных операций требуется код, перехватывающий все исключения. Да, здесь нужно перехватывать все исключения, так как важен не тип ошибки, а возвращение структур данных в согласованное состояние. Перехватив и обработав исключение, не «проглатывайте» его — вызывающий код следует проинформировать о ситуации. Это делается путем повторного вбрасывания того же исключения. В C# и многих других языках это осуществляется просто. Достаточно указать ключевое слово `throw`, как показано в предыдущем фрагменте кода.

Обратите внимание, что в предыдущем примере не указан тип исключения в блоке `catch`, поскольку здесь требуется перехватывать абсолютно все исключения. К счастью, в C# достаточно опустить тип исключения, и инструкция `throw` повторно вбросит это исключение.

Скрытие деталей реализации для сохранения контракта

Иногда бывает полезно после перехвата одного исключения вбросить исключение другого типа. Это требуется с целью сохранения значения контракта метода.

Тип нового исключения должен быть особым (не использоваться в качестве базового или любого другого типа исключений). Рассмотрим псевдокод для типа PhoneBook, определяющий метод поиска номера телефона по имени:

```
internal sealed class PhoneBook {
    private String m_pathname; // Путь к файлу с телефонами

    // Выполнение других методов

    public String GetPhoneNumber(String name) {
        String phone;
        FileStream fs = null;
        try {
            fs = new FileStream(m_pathname, FileMode.Open);
            // Чтение переменной fs до обнаружения нужного имени
            phone = /* номер телефона найден */
        }
        catch (FileNotFoundException e) {
            // Вбрасывание другого исключения, содержащего имя абонента.
            // с заданием исходного исключения в качестве внутреннего
            throw new NameNotFoundException(name, e);
        }
        catch (IOException e) {
            // Вбрасывание другого исключения, содержащего имя абонента.
            // с заданием исходного исключения в качестве внутреннего
            throw new NameNotFoundException(name, e);
        }
        finally {
            if (fs != null) fs.Close();
        }
        return phone;
    }
}
```

Данные телефонного справочника получают из файла (а не из сетевого соединения или базы данных), но пользователю типа PhoneBook это неизвестно, так как это — особенность реализации, которая может измениться в будущем. Поэтому, если почему-либо файл не найден или не может быть прочитан, вызывающий код получит исключение `FileNotFoundException` или `IOException`, которое он не ожидает. Иначе говоря, в неявном контракте метода ничего не говорится о существовании файла и возможности его чтения. Но тот, кто вызвал этот метод, не может знать заранее, что эти допущения не будут нарушены. Поэтому метод `GetPhoneNumber` перехватывает эти два типа исключений и вбрасывает вместо них новое исключение `NameNotFoundException`.

При использовании этого приема следует перехватывать определенные исключения, обстоятельства возникновения которых вы хорошо понимаете. Кроме того, вы должны знать, какие типы исключений являются производными от перехватываемого типа.

Повторное вбрасывание исключения позволяет вызывающему коду узнать о том, что метод не в состоянии решить свою задачу, а тип `NameNotFoundException` дает ему абстрактное представление о причине сбоя. Важно задать внутреннее исключение нового исключения как имеющее тип `FileNotFoundException` или `IOException`, чтобы не потерять его реальную причину, знание которой может быть полезно разработчику типа `PhoneBook`.

ВНИМАНИЕ

Используя описанный подход, мы обманываем вызывающий код в двух отношениях. Во-первых, мы сообщаем неправду о том, что пошло не так. В нашем примере файл не удалось найти, но мы сообщили о невозможности найти имя. Во-вторых, мы сообщаем неправду о месте сбоя. Если бы исключение `FileNotFoundException` могло пробиться наверх стека вызовов, его свойство `StackTrace` говорило бы, что ошибка произошла в конструкторе `FileStream`. Но когда мы «проглатываем» это исключение и вбрасываем новое `NameNotFoundException`, трассировка стека укажет, что ошибка произошла в блоке `catch`, то есть на расстоянии нескольких строк кода от места вбрасывания исключения. Это может серьезно затруднить отладку, поэтому описанный подход следует использовать с осторожностью.

А теперь предположим, что тип `PhoneBook` был реализован чуть иначе. Пусть он поддерживает открытое свойство `PhoneBookPathname`, позволяющее пользователю задавать или получать имя и путь к файлу, в котором нужно искать номер телефона. Поскольку пользователь знает, что данные телефонного справочника берутся из файла, я модифицирую метод `GetPhoneNumber` так, чтобы он не перехватывал никакие исключения, а выпускал их за пределы метода. Заметьте: я меняю не параметры метода `GetPhoneNumber`, а степень его абстрагированности от пользователей типа `PhoneBook`. В результате пользователи будут ожидать, что путь предусмотрен контрактом `PhoneBook`.

Иногда вбрасывание нового исключения после перехвата уже имеющегося преследует целью добавление к исключению новых данных или контекста. Однако эта цель достигается намного проще. Достаточно перехватить исключение нужного вам типа, добавить в коллекцию его свойства `Data` требуемую информацию и повторно его вбросить:

```
private static void SomeMethod(String filename) {  
    try {  
        // Какие-то операции  
    }  
    catch (IOException e) {  
        // Добавление имени файла к объекту IOException  
        e.Data.Add("Filename", filename);  
        throw; // повторное вбрасывание того же исключения  
    }  
}
```

Вот хороший пример этого подхода. Если конструктор типа вбрасывает исключение, которое не перехватывается внутри его

перехватывает своими средствами CLR, а вместо него вбрасывает исключение `TypeInitializationException`. Это полезно, так как CLR создает внутри методов код неявного вызова конструкторов типа¹. Если конструктор типа вбрасывает исключение `DivideByZeroException`, код может попытаться перехватить его и восстановиться. При этом вы даже не узнаете о том, что был задействован конструктор типа. А так как CLR преобразует исключение `DivideByZeroException` в `TypeInitializationException`, вы четко видите, что причиной исключения стали проблемы с конструктором типа, а не с вашим кодом.

Однако этот подход имеет и свои недостатки. При вызове метода через отражение CLR автоматически перехватывает все вбрасываемые этим методом исключения и преобразует их тип в `TargetInvocationException`. В результате для поиска сведений о причинах исключения требуется перехватить объект `TargetInvocationException` и проанализировать его свойство `InnerException`. Более того, при работе с отражениями, как правило, приходится иметь дело примерно с таким кодом:

```
private static void Reflection(Object o) {
    try {
        // Вызов метода DoSomething для этого объекта
        var mi = o.GetType().GetMethod("DoSomething");
        mi.Invoke(o, null); // Метод DoSomething может вбросить исключение
    }
    catch (System.Reflection.TargetInvocationException e) {
        // CLR преобразует его в TargetInvocationException
        throw e.InnerException; // Повторное вбрасывание исходного исключения
    }
}
```

Тем не менее закончить хотелось бы на позитивной ноте. Если для вызова члена вы используете примитивные динамические типы C# (о них мы говорили в главе 5), сгенерированный компилятором код не перехватывает вообще никаких исключений и вбрасывает объект `TargetInvocationException`. Исходно же брошенное исключение просто оказывается наверху стека. Именно поэтому многие разработчики вместо отражений предпочитают использовать примитивные динамические типы.

Необработанные исключения

Итак, при вбрасывании исключения CLR начинает в стеке вызовов поиск блока `catch`, тип которого соответствует типу исключения. Если ни один из блоков `catch` не отвечает типу исключения, возникает *необработанное исключение* (*unhandled exception*). Обнаружив в процессе поток с необработанным исключением, CLR немедленно уничтожает этот поток. Необработанное исключение

¹ Дополнительную информацию по этой теме вы найдете в соответствующем разделе главы 8.

указывает на ситуацию, которую не предвидел программист, и должно считаться признаком серьезной ошибки в приложении. На этом этапе о недостатке следует уведомить компанию, где разработано приложение, чтобы авторы могли устранить неполадку и выпустить исправленную версию.

Разработчикам библиотек классов даже думать нельзя о необработанных исключениях. О них должны заботиться только разработчики приложений, которым в приложении следует реализовать политику, определяющую порядок обработки необработанных исключений. Microsoft рекомендует разработчикам приложений просто принять политику CLR, предлагаемую по умолчанию. То есть в случае необработанного исключения Windows делает запись в журнал системных событий. Увидеть его можно, открыв приложение **Event Viewer** и перейдя к узлу **Windows Logs**→**Application**, как показано на рис 20.1.

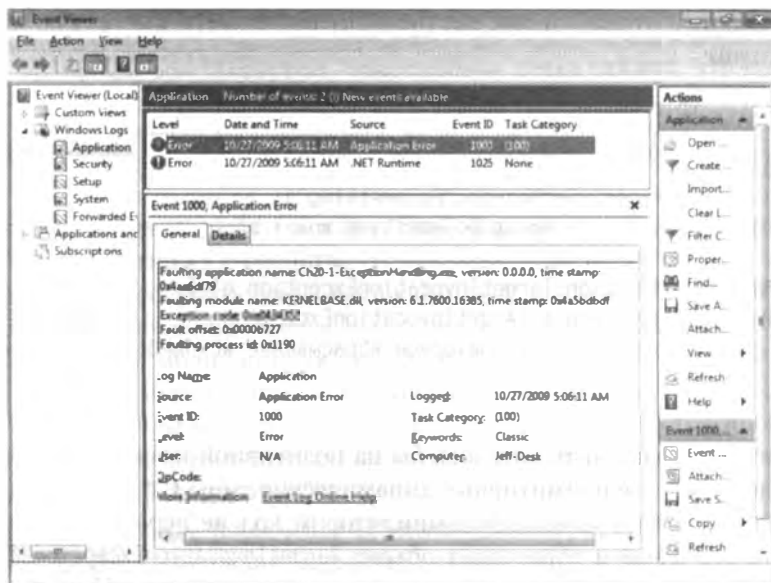


Рис. 20.1. Журнал Windows Event предоставляет информацию о том, работа какого приложения была завершена из-за необработанного исключения

Дополнительную информацию о проблеме можно получить через центр поддержки. Для его запуска нужно щелкнуть на значке с изображением флага на панели задач, выбрать в появившемся меню команду **Open Action Center** (Открыть центр поддержки), перейти на вкладку **Maintenance** (Обслуживание) и выбрать ссылку **View reliability history** (Показать журнал стабильности работы). В нижней части этого окна, как показано на рис. 20.2, можно увидеть список приложений, закрывшихся из-за необработанных исключений.

Для получения еще более подобной информации следует дважды щелкнуть на имени приложения в нижней части журнала **View reliability history**. Пример полученных таким способом сведений показан на рис. 20.3, а расшифровку

полей можно найти в табл. 20.2. Все необработанные исключения, полученные в управляемых приложениях, помещаются в контейнер CLR20r3.



Рис. 20.2. Журнал View reliability history показывает, какие приложения были закрыты из-за необработанных исключений



Рис. 20.3. В журнале View reliability history можно найти дополнительную информацию о поврежденных приложениях

Таблица 20.2. Описание полей журнала View reliability history

Номер поля	Описание ¹
01	Имя исполняемого файла (ограничение 32 знака)
02	Версия сборки исполняемого файла
03	Временная отметка исполняемого файла
04	Полное имя сборки исполняемого файла (ограничение 64 знака)
05	Версия аварийной сборки
06	Временная отметка аварийной сборки
07	Тип и метод аварийной сборки. Это метка метаданных MethodDef (после усечения верхнего байта 0x06), указывающая метод, ставший причиной исключения. Имея это значение, при помощи ILDasm.exe можно найти проблемные тип и метод
08	IL-код некорректного метода. Взяв величину смещения внутри этого кода, при помощи ILDasm.exe можно найти некорректный код
09	Тип вброшенного исключения (ограничение 32 знака)

После фиксации информации о некорректном приложении Windows выводит диалоговое окно, позволяющее пользователю отправить информацию об ошибке в Microsoft². Данный механизм информирования об ошибках называется Windows Error Reporting. Дополнительную информацию о его работе вы можете получить на сайте <http://WinQual.Microsoft.com>.

При желании компании могут зарегистрироваться в Microsoft и получать информацию об ошибках собственных приложений и компонентов. Подписка бесплатна, но только при условии, что сборки удостоверены подписью VeriSign ID (другое название — подпись издателя ПО для Authenticode).

Впрочем, вы вправе разработать собственную систему получения информации о необработанных исключениях, необходимую для устранения недостатков программы. При инициализации приложения можно проинформировать CLR, что есть метод, который нужно вызывать каждый раз, когда в каком-либо потоке возникает необработанное исключение.

Можно использовать следующие члены FCL (подробнее см. документацию):

- ❑ Для всех приложений — событие `UnhandledException` класса `System.AppDomain`. Работа приложений Silverlight недостаточно безопасна для регистрации с этим событием.
- ❑ Для приложений Windows Forms — виртуальный метод `OnThreadException` класса `System.Windows.Forms.NativeWindow`, одноименный виртуальный метод

¹ Строки, выходящие за допустимые границы, подвергаются саморегулирующейся обрезке, например удалению символов `Exception` из типа исключения или расширения `.dll` из имени файла. Если в результате все равно остается слишком длинная строка, CLR создает ее вариант путем хэширования или кодирования строки в base-64.

² Отключить это окно можно, вызвав при помощи механизма `P/Invoke` функцию `SetErrorMode` и передав ей параметр `SEM_NOGPFAULTERRORBOX`.

класса `System.Windows.Forms.Application` и событие `ThreadException` класса `System.Windows.Forms.Application`.

- ❑ Для приложений **Windows Presentation Foundation (WPF)** — событие `DispatcherUnhandledException` класса `System.Windows.Application`, а также события `UnhandledException` и `UnhandledExceptionFilter` класса `System.Windows.Threading.Dispatcher`.
- ❑ Для приложений **Silverlight** — событие `UnhandledException` класса `System.Windows.Application`.
- ❑ Для приложений **ASP.NET Web Form** — событие `Error` класса `System.Web.UI.TemplateControl`. Класс `TemplateControl` — базовый для `System.Web.UI.Page` и `System.Web.UI.UserControl`. Кроме того, можно задействовать событие `Error` класса `System.Web.HttpApplication`.
- ❑ Для приложений **Windows Communication Foundation** — свойство `ErrorHandlers` класса `System.ServiceModel.Dispatcher.ChannelDispatcher`.

В завершение темы хотелось бы сказать несколько слов о необработанных исключениях, которые могут произойти в распределенных приложениях, таких как веб-сайты и веб-службы. В идеальном мире серверное приложение, в котором случилось необработанное исключение, регистрирует сведения об исключении в журнале, уведомит клиента о невозможности выполнения запрошенной операции и завершит свою работу. Но мы живем в реальном мире, в котором может оказаться невозможным отправить уведомление клиенту. На некоторых серверах, поддерживающих данные состояния (таких как, например, **Microsoft SQL Server**), непрактично останавливать сервер и запускать его заново.

В серверном приложении информацию о необработанном исключении нельзя возвращать клиенту, так как ему от этих сведений мало пользы, особенно если клиент создан другой компанией. Более того, сервер должен предоставлять клиентам как можно меньше информации о себе самом, так как это снижает вероятность успешной хакерской атаки.

ПРИМЕЧАНИЕ

В CLR некоторые исключения, вбрасываемые машинным кодом, рассматриваются как исключения поврежденного состояния (**Corrupted State Exceptions, CSE**). Дело в том, что обычно они являются следствием проблем с CLR или с машинным кодом, которые разработчик просто не в состоянии контролировать. По умолчанию CLR не позволяет управляемому коду перехватывать такие исключения и блок `finally` не выполняется. Вот список CSE-исключений в Win32.

EXCEPTION_ACCESS_VIOLATION	EXCEPTION_STACK_OVERFLOW
EXCEPTION_ILLEGAL_INSTRUCTION	EXCEPTION_IN_PAGE_ERROR
EXCEPTION_INVALID_DISPOSITION	EXCEPTION_NONCONTINUABLE_EXCEPTION
EXCEPTION_PRIV_INSTRUCTION	STATUS_UNWIND_CONSOLIDATE

Отдельные управляемые методы могут перегружать методы, предлагаемые по умолчанию, и перехватывать эти исключения, применяя к методу атрибут `System.Runtime.ExceptionServices.HandleProcessCorruptedStateExceptionsAttribute`. Кроме того, к методу должен быть применен атрибут `System.Security.SecurityCriticalAttribute`. Можно перегрузить методы, предлагаемые по умолчанию для всего процесса, присвоив элементу `LegacyCorruptedStateExceptionPolicy` в конфигурационном XML-файле приложения значение `true`. CLR преобразует большинство этих исключений в объект `System.Runtime.InteropServices.SEHException`. Только исключение `EXCEPTION_ACCESS_VIOLATION` преобразуется в объект `System.AccessViolationException`, а исключение `EXCEPTION_STACK_OVERFLOW` — в объект `System.StackOverflowException`.

ПРИМЕЧАНИЕ

Перед вызовом метода можно воспользоваться методом `EnsureSufficientExecutionStack` класса `RuntimeHelper` для проверки количества свободного места в стеке. Если места в стеке недостаточно, метод вбрасывает исключение `InsufficientExecutionStackException`, которое вы можете перехватить. Метод `EnsureSufficientExecutionStack` не принимает аргументов и возвращает значение типа `void`. Обычно он применяется с рекурсивными методами.

Отладка исключений

В отладчике из Microsoft Visual Studio есть специальная поддержка исключений: выберите команду **Exceptions** в меню **Debug** — появится диалоговое окно, показанное на рис 20.4.

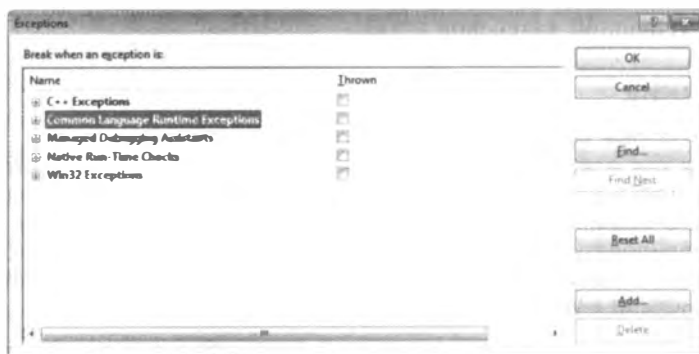


Рис. 20.4. Диалоговое окно Exceptions из Visual Studio с различными исключениями

Здесь показаны типы исключений, поддерживаемые Visual Studio. Раскрыв ветвь **Common Language Runtime Exceptions**, как показано на рис. 20.5, вы увидите пространства имен, поддерживаемые отладчиком из Visual Studio.



Рис. 20.5. CLR-исключения, систематизированные по пространствам имен, в диалоговом окне Exceptions в Visual Studio

Раскрыв пространство имен, как показано на рис. 20.6, вы увидите все определенные в нем типы, производные от `System.Exception`.

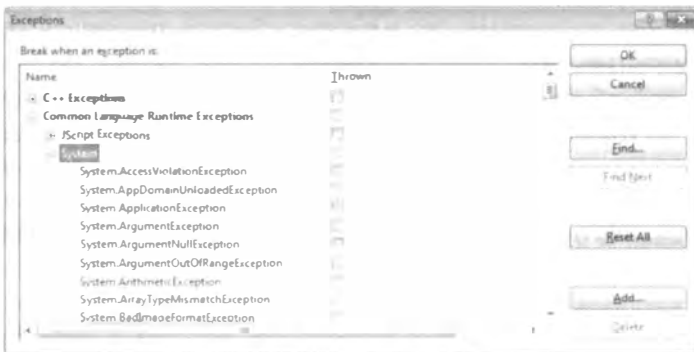


Рис. 20.6. Диалоговое окно Exceptions в Visual Studio с CLR-исключениями, определенными в пространстве имен System

Если для какого-либо исключения установить флажок **Thrown**, при вбрасывании этого исключения отладчик остановится. В момент останова среда CLR еще не приступила к поиску подходящего блока `catch`. Это может быть полезно для отладки кода, ответственного за перехват и обработку соответствующего исключения. Также это может пригодиться в ситуации, когда вы подозреваете, что компонент или библиотека проглатывает или повторно вбрасывает исключение, и не знаете, где поставить точку останова, чтобы «поймать компонент за руку».

Если для исключения флажок `Thrown` не установлен, отладчик остановится, только если после вбрасывания соответствующего исключения оно останется необработанным. Это наиболее популярный вариант, так как обработанное исключение означает, что приложение предвидит возникновение подобных исключений и знает, как с ними справляться.

Вы можете определять собственные типы исключений и добавлять их в окно, щелкнув на кнопке `Add`. В результате появится показанное на рис. 20.7 диалоговое окно.



Рис. 20.7. Передача Visual Studio сведений о собственном типе исключений

В этом окне сначала выбирают тип `Common Language Runtime Exceptions`, а затем вводят полное имя собственного типа исключений. Вводимый тип не обязательно должен быть потомком `System.Exception`, так как типы, не совместимые с CLS, поддерживаются в полном объеме. Если у вас два или больше типов с одинаковыми именами, но в разных сборках, различить эти типы невозможно. К счастью, такое случается редко.

Если в вашей сборке определены несколько типов исключений, следует добавлять их по очереди. Я бы хотел, чтобы в следующей версии это диалоговое окно позволяло находить сборку и автоматически импортировать из нее в отладчик Visual Studio все типы, производные от `Exception`. А возможность дополнительно идентифицировать каждый тип именем сборки решила бы проблему одноименных типов из разных сборок.

Скорость обработки исключений

В сообществе программистов вопросы быстродействия, связанные с обработкой исключений, обсуждаются очень часто и активно. Некоторые пользователи считают эту процедуру настолько медленной, что отказываются к ней прибегать. Но я утверждаю, что для объектно-ориентированной платформы обработка исключений обязательна. Кроме того, у этой процедуры отсутствует альтернатива. Неужели вы предпочтете, чтобы методы возвращали значение `true` или `false`, чтобы сообщать об успехе или неудачи своей работы? Или воспользуетесь кодом ошибок типа `enum`? В этом случае вас ждет разочарование. CLR и код библиотек классов будут вбрасывать исключения, и ваш код начнет возвращать код ошибок. В итоге вам все равно придется вернуться к необходимости обработки исключений.

Трудно сравнивать быстродействие механизма обработки исключений и более привычных средств уведомления об исключениях (возвращения значения `HRESULT`, специальных кодов и т. п.). Если вы напишете код, который сам будет проверять значение, возвращаемое каждым вызванным методом, фильтровать и передавать его коду, вызвавшему метод, то быстродействие приложения серьезно снизится. Даже если оставить быстродействие в стороне, объем дополнительного кодирования и потенциальная возможность ошибок окажутся неподъемными. В такой обстановке обработка исключений выглядит намного лучшей альтернативой.

Неуправляемым компиляторам C++ приходится генерировать код, отслеживающий успешно созданные объекты. Компилятор также должен генерировать код, который при перехвате исключения вызывает деструктор для каждого из успешно созданных объектов. Конечно, здорово, что компилятор принимает эту рутину на себя, однако он генерирует в приложении слишком много кода для ведения внутренней «бухгалтерии» объектов, что негативно влияет как на объем кода, так и на время исполнения.

В то же время управляемым компиляторам намного легче вести учет объектов, поскольку память для управляемых объектов выделяется из управляемой кучи, за которой следит сборщик мусора. Если объект был успешно создан, а затем возникло исключение, сборщик мусора, в конечном счете, освободит память, занятую объектом. Компилятору не приходится генерировать код для внутреннего учета успешно созданных объектов и последующего вызова деструктора. Это значит, что в сравнении с неуправляемым кодом на C++ компилятор генерирует меньше кода, меньше кода обрабатывается и во время выполнения, в результате быстродействие приложения растёт.

Мне приходилось пользоваться обработкой исключений на многих языках, в различных ОС и в системах с разными архитектурами процессора. В каждом случае обработка исключений была реализована по-своему. В некоторых случаях конструкции, обрабатывающие исключения, компилируются прямо в метод, а в других данные, связанные с обработкой исключений, хранятся в связанной с методом таблице, к которой обращаются только при возникновении исключений. Одни компиляторы не способны выполнять подстановку кода методов, содержащих обработчики исключений, другие не регистрируют переменные, если в методе есть обработчик исключений.

Суть в том, что нельзя оценить величину дополнительных издержек, которые влечет за собой обработка исключений в приложении. В управляемом мире сделать такую оценку еще труднее, так как код сборки может работать на любой платформе, поддерживающей .NET Framework. Так, код, сгенерированный JIT-компилятором для обработки исключений на машине x86, будет сильно отличаться от кода, сгенерированного JIT-компилятором в системах с процессором IA64 или JIT-компилятором из .NET Compact Framework.

Мне все же удалось протестировать некоторые мои программы с разными JIT-компиляторами производства Microsoft, предназначенными для внутреннего использования. Я неожиданно обнаружил огромную разницу в быстродей-

ствии. Отсюда следует, что нужно тестировать свой код на всех платформах, на которых предполагается его применять, и вносить соответствующие изменения. И в этом случае я бы не беспокоился о снижении быстродействия из-за обработки исключений. Как я уже отмечал, польза от обработки исключений намного перевешивает это снижение.

Если вам интересно, насколько обработка исключений снижает производительность вашего кода, можно задействовать встроенный монитор производительности Windows. Показанный на рис. 20.8 снимок экрана демонстрирует счетчики, связанные с обработкой исключений, которые устанавливаются при установке .NET Framework.

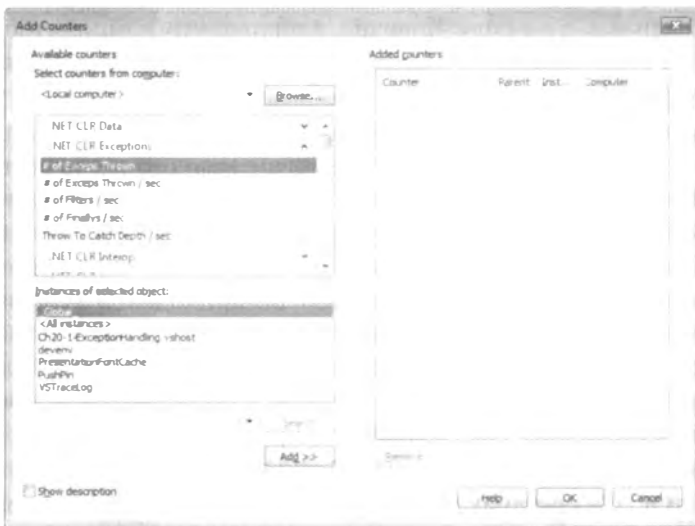


Рис. 20.8. Счетчики исключений .NET CLR в окне монитора производительности

Когда-нибудь вы столкнетесь с часто вызываемым методом, который активно вбрасывает исключения. В такой ситуации снижение производительности из-за обработки слишком частых исключений оказывается очень значительным. В частности, в Microsoft слышали от нескольких клиентов жалобы, что при вызове метода `Parse` класса `Int32` и передаче данных, введенных конечными пользователями, возникал сбой. Так как метод `Parse` вызывался часто, вбрасывание и перехват исключений серьезно снижали общую производительность приложения.

Для решения заявленной клиентами проблемы и в соответствии с принципами, описанными в этой главе, специалисты Microsoft добавили в класс `Int32` метод `TryParse`, имеющий две перегруженные версии:

```
public static Boolean TryParse(String s, out Int32 result);
public static Boolean TryParse(String s, NumberStyles styles,
    IFormatProvider, provider, out Int32 result);
```

Как видите, эти методы возвращают значение типа `Boolean`, которое указывает, можно ли обработать все символы переданной в `Int32` строки. Эти методы также возвращают выходной параметр `result`. Если метод возвращает значение `true`, параметр `result` содержит результат преобразования строки в 32-разрядное целое. В противном случае этот параметр оказывается равен 0, но это значение вряд ли может использоваться в коде.

Хотел бы прояснить одну вещь. Возвращенное методом `TryXxx` значение `false` указывает на один и только один тип сбоя. Для других сбоев метод может вбрасывать исключения. Например, метод `TryParse` класса `Int32` в случае неверного параметра вбрасывает исключение `ArgumentException` и, конечно же, может вбросить исключение `OutOfMemoryException`, если при вызове `TryParse` происходит ошибка выделения памяти.

Также хотелось бы подчеркнуть, что объектно-ориентированное программирование позволяет повысить производительность труда программиста. И не в последнюю очередь за счет запрета на передачу кодов ошибок через члены типа. Иначе говоря, конструкторы, методы, свойства и пр. создаются с тем расчетом, что в их работе сбоев не будет. И при условии правильности определения в большинстве случаев при использовании члена сбоев в нем не будет, а значит, не будет снижения производительности, потому что не будет исключений.

Определять типы и их члены надо с тем расчетом, чтобы свести к минимуму вероятность их сбоев при стандартных сценариях их использования. Если вы позже услышите от своих клиентов, что из-за вбрасывания множества исключений производительность неудовлетворительна, тогда и только тогда имеет смысл подумать о добавлении в тип методов `TryXxx`. Иначе говоря, сначала надо создать оптимальную объектную модель, а затем, если пользователи окажутся недовольными, добавить в тип несколько методов `TryXxx`, которые облегчат им жизнь. Тем же, кто не испытывает проблем с производительностью, лучше продолжить работать с версией типа без этих методов.

Области ограниченного выполнения

Есть приложения, которым не нужны высокая надежность и способность к восстановлению после любых сбоев, например `Notepad.exe` и `Calc.exe`. Многие из нас сталкивались с ситуацией, когда приложения из пакета `Microsoft Office` — `WinWord.exe`, `Excel.exe` и `Outlook.exe` — просто завершают свою работу из-за необработанных исключений. Также многие серверные приложения, например веб-сервер, имеют неизменное состояние и в случае необработанных исключений автоматически перезагружаются. Конечно, существуют серверы, например `SQL Server`, для которых потеря данных в подобных случаях намного более критична.

В CLR информацию о состоянии содержит класс `AppDomain` (подробно он рассмотрен в главе 22). Его выгрузка сопровождается выгрузкой всех состояний. Соответственно, если поток в домене приложений сталкивается с необработанным исключением, можно выгрузить домен (удалив все состояния), не завершая работы приложения¹.

Областью ограниченного выполнения (Constrained Execution Region, CER) называется устойчивый к сбоям фрагмент кода. Так как домены приложений допускают выгрузку с уничтожением своего состояния, области ограниченного выполнения обычно служат для управления состоянием, общим для нескольких доменов или процессов. Особенно они полезны, если вы собираетесь работать с состоянием, в котором возможны неожиданные исключения. Такие исключения иногда называют *асинхронными* (asynchronous exceptions). Например, для вызова метода среда CLR должна загрузить сборку, создать тип объекта в куче загрузчика домена приложений, вызвать статический конструктор типа, скомпилировать IL-код в машинный код и т. п. Сбой может произойти в ходе любой из этих операций, и CLR оповестит об этом при помощи исключения.

Если любой такой сбой произойдет в блоке `catch` или `finally`, код восстановления или очистки будет выполнен не полностью. Рассмотрим пример кода, в котором возможен сбой:

```
private static void Demo1() {
    try {
        Console.WriteLine("In try");
    }
    finally {
        // Неявный вызов статического конструктора Type1
        Type1.M();
    }
}

private sealed class Type1 {
    static Type1() {
        // В случае исключения М не вызывается
        Console.WriteLine("Type1's static ctor called");
    }
    public static void M() { }
}
```

Вот результат работы этого кода:

```
In try
Type1's static ctor called
```

¹ Это верно для случаев, когда поток не выходит за пределы одного класса `AppDomain` (подобно тому, как ASP.NET и управляемый SQL-сервер хранят сценарии процедур). Но для потоков, пересекающих границы домена приложений, может потребоваться завершение работы.

Нам нужно, чтобы код в блоке `try` выполнялся только при условии выполнения кода в связанных с ним блоках `catch` и `finally`. Для этого код следует переписать так:

```
private static void Demo2() {
    // Подготавливаем код в блоке finally
    RuntimeHelpers.PrepareConstrainedRegions(); // Пространство имен
                                                // System.Runtime.CompilerServices

    try {
        Console.WriteLine("In try");
    }
    finally {
        // Неявный вызов статического конструктора Type2
        Type2.M();
    }
}

public class Type2 {
    static Type2() {
        Console.WriteLine("Type2's static ctor called");
    }

    // Используем атрибут, определенный в пространстве имен
    // System.Runtime.ConstrainedExecution
    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
    public static void M() { }
}
```

После запуска этой версии кода получаем:

```
Type2's static ctor called
In try
```

Метод `PrepareConstrainedRegions` особенный. Обнаружив его перед блоком `try`, JIT-компилятор немедленно начинает работать с соответствующими блоками `catch` и `finally`. JIT-компилятор загружает любые сборки, создает любые типы, вызывает любые статические конструкторы и компилирует любые методы. Если хотя бы одна из этих операций дает сбой, исключение возникает до входа потока в блок `try`. В процессе подготовки методов JIT-компилятор просматривает весь граф вызовов. Однако обрабатывает он только методы, к которым был применен атрибут `ReliabilityContractAttribute` со значениями параметра `Consistency` равными `WillNotCorruptState` или `Consistency.MayCorruptInstance`, так как для методов, которые могут повредить домен приложений или состояние процесса, CLR не дает никаких гарантий. Нужно гарантировать, что внутри защищаемого методом `PrepareConstrainedRegions` блока `catch` или `finally` будут вызываться только методы с настроенным, как показано в предыдущем фрагменте кода, атрибутом `ReliabilityContractAttribute`.

Вот как выглядит этот атрибут:

```
public sealed class ReliabilityContractAttribute : Attribute {
    public ReliabilityContractAttribute(
        Consistency consistencyGuarantee, Cer cer);
    public Cer Cer { get; }
    public Consistency ConsistencyGuarantee { get; }
}
```

Он дает разработчику возможность указать степень надежности отдельного метода¹. Типы Cer и Consistency относятся к перечислениям и определяются следующим образом:

```
enum Consistency {
    MayCorruptProcess, MayCorruptAppDomain,
    MayCorruptInstance, WillNotCorruptState
}
enum Cer { None, MayFail, Success }
```

Если ваш метод гарантировано не может повредить состояние, используйте в качестве параметра Consistency.WillNotCorruptState. В противном случае выберите одно из трех других значений в зависимости от степени его надежности. Для метода с гарантированно успешным завершением используйте значение Cer.Success. В противном случае выбирайте параметр Cer.MayFail. Любой метод, для которого не определен атрибут ReliabilityContractAttribute, можно считать помеченным таким вот образом:

```
[ReliabilityContract(Consistency.MayCorruptProcess, Cer.None)]
```

Значение Cer.None указывает, что никаких CER-гарантий в данном случае не дается. Другими словами, метод может завершиться неудачно и даже не сообщить об этом. Помните, что большинство этих параметров дают методу возможность проинформировать вызывающую сторону, чего от него можно ожидать. CLR и JIT-компилятор эти сведения игнорируют.

Чтобы написать надежный метод, делайте его как можно меньше по размеру и ограничивайте сферу его действия. Убедитесь, что там не выделяется память под объекты (например, не выполняется упаковка), не вызывайте внутри ни виртуальных, ни интерфейсных методов, не пользуйтесь делегатами или отражениями, так как в этом случае JIT-компилятор не сможет определить, какой именно метод вызывается на самом деле. Можно даже вручную подготовить все методы при помощи одного из следующих методов класса RuntimeHelpers:

```
public static void PrepareMethod(RuntimeMethodHandle method)
public static void PrepareMethod(RuntimeMethodHandle method,
    RuntimeTypeHandle[] instantiation)
public static void PrepareDelegate(Delegate d);
public static void PrepareContractedDelegate(Delegate d);
```

¹ Атрибут можно применить также к интерфейсу, конструктору, структуре, классу или сборке.

Имейте в виду, что ни компилятор, ни CLR не проверяют гарантии, которые вы даете, снабжая свой метод атрибутом `ReliabilityContractAttribute`. Если вы что-то перепутали, состояние вполне может оказаться испорченным.

ПРИМЕЧАНИЕ

Даже хорошо подготовленный метод может стать источником исключения `StackOverflowException`. Если среда CLR не запущена, исключение `StackOverflowException` приводит к немедленному завершению процесса путем внутреннего вызова метода `Environment.FailFast`. Если же среда запущена, метод `PreparedConstrainedRegions` проверяет, осталось ли в стеке хотя бы 48 Кбайт свободного места. При ограниченном месте в стеке исключение `StackOverflowException` вбрасывается до начала блока `try`.

Не следует забывать и про метод `ExecuteCodeWithGuaranteedCleanup` класса `RuntimeHelper`, который предоставляет еще одну возможность гарантированной очистки:

```
public static void ExecuteCodeWithGuaranteedCleanup(  
    TryCode code, CleanupCode backoutCode, Object userData);
```

При работе с ним вы передаете тело блоков `try` и `finally` в качестве методов обратного вызова, свойства которых совпадают с этими двумя делегатами:

```
public delegate void TryCode(Object userData);  
public delegate void CleanupCode(Object userData, Boolean exceptionThrown);
```

Ну и наконец, упомяну еще один класс, помогающий гарантировать выполнение кода. Это класс `CriticalFinalizerObject`, который подробно рассмотрен в следующей главе.

Контракты кода

Контракты кода (code contracts) — это средство официально документировать спецификацию вашего кода внутри самого кода. Контракты бывают трех видов:

- ❑ *предусловия* (preconditions) используются для проверки аргументов;
- ❑ *постусловия* (postconditions) служат для проверки состояния завершения метода вне зависимости от того, нормально он завершился или с исключением;
- ❑ *инварианты* (object invariants) позволяют удостовериться, что данные объекта находятся в хорошем состоянии на всем протяжении жизни этого объекта.

Контракты облегчают использование кода, его понимание, разработку, тестирование¹, документирование и распознавание ошибок на ранних стадиях. Предусловия, постусловия и инварианты можно представить в виде части сигнатуры методов. При этом вы можете ослабить контракт для новой версии кода, но обратное невозможно — усиление контракта отрицательно скажется на совместимости версий.

Основой контрактов кода является статический класс `System.Diagnostics.Contracts.Contract`:

```
public static class Contract {
    // Методы с предусловиями: [Conditional("CONTRACTS_FULL")]
    public static void Requires(Boolean condition);
    public static void EndContractBlock();

    // Предусловия: Always
    public static void Requires<TException>(
        Boolean condition) where TException : Exception;

    // Методы с постусловиями: [Conditional("CONTRACTS_FULL")]
    public static void Ensures(Boolean condition);
    public static void EnsuresOnThrow<TException>(Boolean condition)
        where TException : Exception;

    // Специальные методы с постусловиями: Always
    public static T Result<T>();
    public static T OldValue<T>(T value);
    public static T ValueAtReturn<T>(out T value);

    // Инвариантные методы объекта: [Conditional("CONTRACTS_FULL")]
    public static void Invariant(Boolean condition);

    // Квантификаторные методы: Always
    public static Boolean Exists<T>(
        IEnumerable<T> collection, Predicate<T> predicate);
    public static Boolean Exists(
        Int32 fromInclusive, Int32 toExclusive, Predicate<Int32> predicate);
    public static Boolean ForAll<T>(
        IEnumerable<T> collection, Predicate<T> predicate);
    public static Boolean ForAll(
        Int32 fromInclusive, Int32 toExclusive,
        Predicate<Int32> predicate);

    // Вспомогательные методы:
    // [Conditional("CONTRACTS_FULL")] или [Conditional("DEBUG")]
```

¹ Для автоматического тестирования можно использовать инструмент Pex, созданный группой Microsoft Research: <http://research.microsoft.com/en-us/projects/pex/>.

```
public static void Assert(Boolean condition);  
public static void Assume(Boolean condition);  
  
// Инфраструктурное событие: обычно в коде это событие не используется  
public static event EventHandler<ContractFailedEventArgs> ContractFailed;  
}
```

Как показано в этом коде, многим из этих статических методов назначен атрибут `[Conditional("CONTRACTS_FULL")]`, а некоторым методам класса `Helper` — еще и атрибут `[Conditional("DEBUG")]`. Это означает, что при отсутствии специального символа, определенного в момент компиляции, компилятор проигнорирует любой написанный вами код вызова этих методов. Пометка `Always` означает, что компилятор всегда будет создавать код вызова этих методов. Кроме того, методы `Requires`, `Requires<TException>`, `Ensures`, `EnsuresOnThrow`, `Invariant`, `Assert` и `Assume` дополнительно имеют перегруженные версии (здесь они не показаны), позволяющие принимать аргумент типа `String`. В результате вы можете в явном виде задать сообщение, которое будет появляться при нарушении контракта.

По умолчанию контракты практически не используются и для их включения нужно вручную указать в свойствах проекта символ `CONTRACTS_FULL`. Также вам могут потребоваться дополнительные инструменты и панель свойств Visual Studio, которые можно загрузить с сайта <http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx>. В пакет Visual Studio эти инструменты пока не входят, так как являясь относительно новыми, они крайне быстро развиваются. И на сайте DevLabs получить новую версию можно быстрее, чем вместе с Visual Studio. После установки новых инструментов появится новая панель свойств (рис. 20.9).



Рис. 20.9. Панель Code Contracts для Visual Studio

Для включения контрактов кода установите флажок **Perform Runtime Contract Checking** и в расположенном справа от него раскрывающемся списке выберите вариант **Full**. Это задаст символ `CONTRACTS_FULL` в момент создания проекта и активирует необходимые инструменты (они кратко описаны далее). В результате нарушение контракта во время выполнения программы будут сопровождаться событием `ContractFailed` класса `Contract`. Обычно разработчики не регистрируют методов с этим событием, но если вы решите изменить этой традиции, все зарегистрированные вами методы получают объект `ContractFailedEventArgs`, который выглядит следующим образом:

```
public sealed class ContractFailedEventArgs : EventArgs {
    public ContractFailedEventArgs(ContractFailureKind failureKind,
        String message, String condition, Exception originalException);

    public ContractFailureKind FailureKind { get; }
    public String Message { get; }
    public String Condition { get; }
    public Exception OriginalException { get; }

    public Boolean Handled { get; } // Верно, если хоть один обработчик
                                    // вызвал SetHandled
    public void SetHandled();       // Присваивает Handled значение true,
                                    // позволяя игнорировать нарушение

    public Boolean Unwind { get; } // Верно, если хоть один обработчик
                                    // вызвал SetUnwind или threw
    public void SetUnwind();       // Присваивает Unwind значение true,
                                    // принудительно вбрасывая ContractException
}
```

С этим событием можно зарегистрировать множество методов его обработки. И каждый такой метод может обработать нарушение контракта указанным вами способом. Например, можно записать сведения о нарушении в журнал, проигнорировать нарушение (вызвав метод `SetHandled`) или завершить процесс. При вызове любым из методов метода `SetHandled` нарушение считается обработанным и после результата, возвращенного методом обработки, приложение может работать дальше, если конечно, обработчик не вызвал метод `SetUnwind`. Если же такой вызов произошел, то после завершения всех методов обработки вбрасывается исключение `System.Diagnostics.Contracts.ContractException`. Это внутреннее исключение библиотеки `mscorlib.dll`, значит, вы не сможете написать блок `catch` для его перехвата. Если же какой-нибудь из методов обработки становится источником необработанного исключения, сначала вызываются все остальные обработчики, а затем вбрасывается исключение `ContractException`.

Если обработчики событий отсутствуют или ни один из них не вызывает методы `SetHandled` и `SetUnwind` и не становится источником необработанного исключения, нарушение контракта сопровождается заданной по умолчанию

процедурой. Если среда CLR загружена, приложение оповещается о нарушении контракта. В случаях когда CLR запускает приложение в виде неинтерактивного оконного терминала (сюда относится, к примеру, Windows service application), вызывается метод `Environment.FailFast`, мгновенно завершающий процесс.

Если перед компиляцией был установлен флажок **Assert On Contract Failure**, появится диалоговое окно, в котором с приложением можно будет связать отладчик. При сброшенном флажке нарушение контракта сопровождается исключением `ContractException`.

Рассмотрим класс, использующий контракты кода.

```
public sealed class Item { /* ... */ }

public sealed class ShoppingCart {
    private List<Item> m_cart = new List<Item>();
    private Decimal m_totalCost = 0;

    public ShoppingCart() {
    }

    public void AddItem(Item item) {
        AddItemHelper(m_cart, item, ref m_totalCost);
    }

    private static void AddItemHelper(
        List<Item> m_cart, Item newItem, ref Decimal totalCost) {

        // Предусловия:
        Contract.Requires(newItem != null);
        Contract.Requires(Contract.ForAll(m_cart, s => s != newItem));

        // Постусловия:
        Contract.Ensures(Contract.Exists(m_cart, s => s == newItem));
        Contract.Ensures(totalCost >= Contract.OldValue(totalCost));
        Contract.EnsuresOnThrow<IOException>(
            totalCost == Contract.OldValue(totalCost));

        // Какие-то операции (способные вбросить IOException)
        m_cart.Add(newItem);
        totalCost += 1.00M;
    }

    // Инвариант
    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(m_totalCost >= 0);
    }
}
```


В методе `AddItemHelper` определяется множество контрактов кода. Предусловие указывает, что параметр `newItem` должен отличаться от `null`, а добавляемый в список элемент не может дублировать уже имеющиеся. Постусловие гласит, что новый элемент должен присутствовать в списке, а общая цена покупок после этой операции должна увеличиться. В постусловии также сказано, что если метод `AddItemHelper` по какой-то причине станет источником исключения `IOException`, параметр `totalCost` должен сохранить значение, которое он имел перед вызовом метода. Защищенный метод `ObjectInvariant` гарантирует, что поле `m_totalCost` объекта не будет содержать отрицательного значения.

ВНИМАНИЕ

Все члены, на которые присутствуют ссылки в предусловиях, постусловиях и инвариантах, должны быть свободны от сторонних эффектов. Такое требование связано с тем, что во время тестирования ни в коем случае не должно меняться состояние объекта. Кроме того, все методы, на которые есть ссылки в предусловии, должны быть доступны так же, как и метод, определяющий само предусловие. Иначе перед вызовом метода не будет возможности проверить соответствие условиям. Это ограничение не касается членов, на которые есть ссылки в постусловиях и инвариантах. Степень доступа к ним может быть любой, главное, чтобы код компилировался. Причина снятия ограничения состоит в том, что проверки в постусловии и инварианте не влияют на корректность вызова метода.

ВНИМАНИЕ

Что касается наследования, производный тип не может перегружать и менять предусловия в виртуальных членах, определенных в базовом типе. Аналогично, тип реализации члена интерфейса не может менять предусловия, определенные этим членом. Если для члена отсутствует определенный в явном виде контракт, значит, существует неявный контракт, который логично представить в таком виде:

```
Contract.Requires(true);
```

И так как при переходе к новой версии ужесточить контракт не получится (или придется пожертвовать совместимостью версий), при вводе новых виртуальных, абстрактных или интерфейсных членов следует очень аккуратно выбирать предусловия. Для постусловий и инвариантов вы можете добавлять и убирать контракты по желанию, главное, чтобы условия, поставленные в виртуальном/абстрактном/интерфейсном члене, можно было логически соединить с условиями в перегруженном члене.

Итак, вы научились определять контракты. Пришло время поговорить о том, как они функционируют во время работы программы. Объявлять предусловия и постусловия следует в верхней части методов, чтобы их легко можно было найти. Предусловия проверяются при вызове метода. При

этом хотелось бы, чтобы постусловия проверялись только после завершения метода. Для этого созданную компилятором C# сборку следует обработать инструментом Code Contract Rewriter (файл **CCRewrite.exe**, находится по адресу **C:\Program Files (x86)\Microsoft\Contracts\Bin**) и получить ее модифицированную версию. После установки флажка **Perform Runtime Contract Checking** Visual Studio начнет вызывать эту утилиту автоматически еще на стадии создания проекта. Утилита анализирует IL-код всех ваших методов и переписывает его таким образом, чтобы постусловия выполнялись только после завершения методов. Для методов, имеющих несколько точек выхода, утилита **CCRewrite.exe** редактирует IL-код, заставляя проверять условие перед завершением метода.

Утилита **CCRewrite.exe** проверяет тип метода, помеченного атрибутом `[ContractInvariantMethod]`. Имя такого метода может быть любым, но обычно его называют `ObjectInvariant` и добавляют модификатор `private` (как я и сделал). Этот метод не имеет аргументов и возвращает значение типа `void`. При виде его **CCRewrite.exe** вставляет код вызова метода `ObjectInvariant` после всех открытых экземплярных методов. В результате состояние объекта проверяется после возвращения значения каждым из методов, гарантируя, что ни один из них не нарушил условия контракта. Метод `Finalize` и метод `Dispose` класса `IDisposable` утилитой **CCRewrite.exe** не редактируются, потому что состояние объекта перед его уничтожением или отправкой в корзину не имеет никакого значения. Следует также заметить, что для одного типа можно определить множество методов с атрибутом `[ContractInvariantMethod]`; это полезно при работе с разделяемыми типами. Утилита **CCRewrite.exe** переписит IL-код таким образом, что все эти методы будут вызываться (в неопределенном порядке) в конце каждого открытого метода.

Методы `Assert` и `Assume` не похожи на остальные. Во-первых, они не являются частью сигнатуры метода и их нельзя поместить в начало. Во время выполнения они действуют идентично: проверяют переданное им условие и в случае его несоблюдения вбрасывают исключение. Впрочем, есть еще и такой инструмент, как **Code Contract Checker (CCCheck.exe)**, анализирующий производимый компилятором C# IL-код в попытке статистически удостовериться в отсутствии нарушений контракта. Эта утилита пытается удостовериться, что все условия, переданные методу `Assert`, выполнены. Если сделать это не получается, происходит переход к методу `Assume`.

Давайте рассмотрим пример. Предположим, что у нас есть следующее определение типа:

```
internal sealed class SomeType {  
    private static String s_name = "Jeffrey";  
  
    public static void ShowFirstLetter() {  
        Console.WriteLine(s_name[0]); // внимание: требования  
                                     // не подтверждены: index < this.Length  
    }  
}
```

При сборке этого кода с установленным флажком **Perform Static Contract Checking** утилита **CCCheck.exe** выводит предостерегающее сообщение (внимание: требования не подтверждены: `index < this.Length`):

```
warning: requires unproven: index < this.Length
```

Это сообщение гласит, что запрос первой буквы элемента `s_name` может закончиться неудачей и стать источником исключения, так как неизвестно, ссылается ли этот элемент на строку, состоящую хотя бы из одного символа.

Следовательно, в метод `ShowFirstLetter` нужно добавить утверждение:

```
public static void ShowFirstLetter() {  
    Contract.Assert(s_name.Length >= 1); // внимание: утверждение  
                                         // не подтверждено  
    Console.WriteLine(s_name[0]);  
}
```

К сожалению, анализируя этот код, утилита **CCCheck.exe** все равно не может проверить, ссылается ли элемент `s_name` на строку, содержащую хотя бы один символ. В итоге мы снова получаем предупреждение. Иногда утилита не может проверить утверждение из-за своих внутренних ограничений; ее будущие версии смогут осуществлять более полный анализ.

Чтобы обойти недостатки этой утилиты, перейдем от метода `Assert` к методу `Assume`. Зная наверняка, что никакой код не внесет изменений в элемент `s_name`, мы можем отредактировать метод `ShowFirstLetter` следующим образом:

```
public static void ShowFirstLetter() {  
    Contract.Assume(s_name.Length >= 1); // Предостережений нет!  
    Console.WriteLine(s_name[0]);  
}
```

В этой версии кода утилита **CCCheck.exe** верит нам на слово и заключает, что элемент `s_name` всегда ссылается на строку, содержащую хотя бы один символ. В результате метод `ShowFirstLetter` проходит статическую проверку контракта кода без предостерегающих сообщений.

Осталось рассмотреть инструмент **Code Contract Reference Assembly Generator (CCRefGen.exe)**. Эта утилита ускоряет поиск ошибок, но произведенный в процессе проверки контракта код увеличивает размер вашей сборки и отрицательно сказывается на производительности. Исправить этот недостаток можно при помощи утилиты **CCRefGen.exe**, создающей отдельную сборку со ссылкой на контракт. В **Visual Studio** она запускается автоматически, если выбрать в раскрывающемся списке **Contract Reference Assembly** вариант **Build**. Сборки с контрактами обычно носят имя *ИмяСборки.Contracts.dll* (например, **MSCorLib.Contracts.dll**) и содержат только метаданные и описывающий контракт IL-код. Опознать их можно также по примененному к таблице метаданных с определением сборки атрибуту `System.Diagnostics.Contracts.ContractRefere`

nceAssemblyAttribute. Утилиты **CCRewrite.exe** и **CCCheck.exe** могут использовать сборки со ссылками на контракты в качестве входных данных для анализа.

Ну и самый последний инструмент **Code Contract Document Generator (CCDocGen.exe)** добавляет информацию о контракте в XML-файл, создаваемый компилятором **C#** при установке переключателя `/doc:file`. Этот XML-файл, дополненный утилитой **CCDocGen.exe**, после обработки инструментом **Sandcastle** выдает документацию в стиле **MSDN** с информацией о контрактах.

Глава 21. Автоматическое управление памятью (сборка мусора)

Эта глава рассказывает о создании новых объектов управляемыми приложениями, о том, как управляемая куча распоряжается временем жизни этих объектов и как освобождается занятая ими память. Мы рассмотрим работу сборщика мусора общезыковой среды CLR и проблемы, связанные с его производительностью. В конце главы мы поговорим о приемах разработки приложений, наиболее эффективно использующих память.

Работа на платформе, поддерживающей сборку мусора

Любая программа использует ресурсы — файлы, буферы в памяти, пространство экрана, сетевые подключения, базы данных и т. п. В объектно-ориентированной среде каждый тип идентифицирует некий доступный этой программе ресурс. Чтобы им воспользоваться, должна быть выделена память для представления этого типа. Для доступа к ресурсу вам нужно:

1. Выделить память для типа, представляющего ресурс, вызвав команду `newobj` промежуточного языка, которая генерируется при использовании оператора `new` в C#.
2. Инициализировать выделенную память, установив начальное состояние ресурса и сделав его пригодным к использованию. За установку начального состояния типа отвечает его конструктор.
3. Использовать ресурс, обращаясь к членам его типа (при необходимости операция может повторяться).
4. В рамках процедуры очистки ликвидировать состояние ресурса (этому этапу посвящен раздел «Эталон освобождения ресурсов: принудительная очистка объекта» данной главы).
5. Освободить память. За этот этап отвечает исключительно сборщик мусора.

Эта, на первый взгляд, простая парадигма стала одним из основных источников ошибок программирования. Сколько раз программисты забывали освободить память, ставшую ненужной, или пытались использовать уже освобожденную память.

При неуправляемом программировании эти два вида ошибок в приложениях опаснее остальных, так как обычно нельзя предсказать ни их последствий, ни периодичность их появления. Прочие ошибки довольно просто исправить, заметив, что приложение функционирует неправильно. Но эти две ошибки ведут к утечке ресурсов (увеличивая потребление памяти) и повреждению объектов (дестабилизируя систему), что делает работу приложения непредсказуемой. Для облегчения поиска таких ошибок специально разработано множество инструментов, в Microsoft Windows это Task Manager, Process Explorer и Performance Monitor, кроме того, можно отметить утилиту Purify компании Rational.

Правильное управление ресурсами — сложная и затратная задача, отвлекающая разработчиков от решения реальных задач. Вот если бы у разработчиков был механизм, упрощающий управление памятью! И такой механизм имеется — это сборка мусора.

Сборка мусора (garbage collection) полностью освобождает разработчика от необходимости следить за использованием и своевременным освобождением памяти. Однако сборщик мусора ничего не знает о ресурсе, представленном типом в памяти, а значит, не может знать, как выполнить четвертый этап нашей пошаговой процедуры — корректно ликвидировать состояние ресурса. Для должной очистки ресурса разработчику нужно написать код, «умеющий» правильно выполнить очистку. Этот код следует поместить в метод финализации, а также в методы Dispose и Close, о которых мы поговорим позже. Однако, как показано далее, и здесь сборщик мусора может быть полезным, во многих случаях позволяя разработчикам пропускать этот этап.

Кроме того, многие типы, такие как String, Attribute, Delegate и Exception, представляют ресурсы, не требующие специальной очистки. Так, чтобы полностью очистить ресурс типа String, достаточно уничтожить массив символов в памяти его объекта.

В тоже время, когда нужно освободить память от объекта типа, представляющего (или являющегося оболочкой) для неуправляемого или машинного ресурса, например файла, подключения к базе данных, сокета, мьютекса, битовой карты, значка и пр., всегда требуется выполнить код очистки. В этой главе мы поговорим о том, как правильно определять типы, требующие явной очистки, и как использовать типы, поддерживающие явную очистку. Для начала же рассмотрим процесс выделения памяти и инициализации ресурсов.

Выделение ресурсов из управляемой кучи

CLR требует выделять память для всех ресурсов из так называемой *управляемой кучи* (managed heap). От кучи исполняющей среды языка C она отличается лишь тем, что разработчику из управляемой кучи удалять объекты не нужно. Став ненужными приложению, они удаляются автоматически. Естественно, сразу возникает вопрос: «Как управляемая куча узнает, что объект больше не нужен приложению?» Впрочем, об этом мы поговорим чуть позже.

В настоящее время существует несколько алгоритмов сборки мусора. Каждый из них оптимизирован под конкретную среду, обеспечивая максимальную производительность. В этой главе основное внимание уделяется алгоритму сборки мусора, реализованному в CLR Microsoft .NET Framework. Начнем с основных понятий.

При инициализации процесса CLR резервирует непрерывную область адресного пространства, которая изначально не соответствует никакой физической памяти. Это и есть управляемая куча. Она также поддерживает указатель, который я называю `NextObjPtr`. Он определяет, где в куче будет выделена память для следующего объекта, и изначально указывает на базовый адрес этой зарезервированной области адресного пространства¹.

IL-команда `newobj` создает объект. Многие языки (в том числе C#, C++/CLI и Microsoft Visual Basic) поддерживают оператор `new`, заставляющий компилятор поместить команду `newobj` в IL-код метода. После получения этой команды CLR:

- 1) подсчитывает количество байтов, необходимых для размещения полей типа (и всех полей его базового типа);
- 2) прибавляет к полученному значению количество байтов, необходимых для размещения системных полей объекта. У каждого объекта есть пара таких полей: указатель на тип объекта и индекс синхронизации блока. В 32-разрядных приложениях для каждого из этих полей требуется 32 бита, что увеличивает размер каждого объекта на 8 байт, а в 64-разрядных приложениях каждое поле занимает 64 бита, добавляя к каждому объекту 16 байт;
- 3) проверяет, хватает ли в зарезервированной области байтов на выделение памяти для объекта (при необходимости передает память). Если в управляемой куче достаточно места для объекта, ему выделяется память, начиная с адреса, на который ссылается указатель `NextObjPtr`, а занимаемые им байты обнуляются. Затем вызывается конструктор типа (передающий `NextObjPtr` в качестве параметра `this`), и IL-команда `newobj` (или оператор `new` в C#) возвращает адрес объекта. Перед возвратом этого адреса `NextObjPtr` переходит на первый адрес после объекта, указывая на адрес, по которому в куче будет помещен следующий объект.

Рисунок 21.1 демонстрирует управляемую кучу с тремя объектами: *A*, *B* и *C*. Новый объект размещается по адресу, заданному указателем `NextObjPtr` (сразу после объекта *C*).

¹ При инициализации CLR резервирует два сегмента виртуального адресного пространства: один в куче для обычных объектов, а второй — в куче для больших. Для клиентских приложений размер сегмента составляет примерно 16 Мбайт, для серверных — примерно 64 Мбайт. Он зависит также от того, 32- или 64-разрядную операционную систему вы используете, от количества процессоров (чем больше процессоров, тем меньше будет выделенный сегмент). Дополнительные сегменты выделяются по мере заполнения объектами уже выделенных, вплоть до завершения адресного пространства процесса. Так что память, выделенная для вашего приложения, ограничена виртуальным адресным пространством процесса. Для 64-разрядного процесса выделяется намного больше памяти, чем для 32-разрядного.

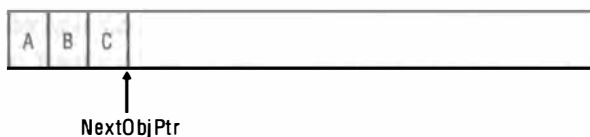


Рис. 21.1. Только что инициализированная управляемая куча с тремя объектами

Для сравнения посмотрим, как выделяется память в куче исполняющей среды C. Чтобы выделить в куче память для объекта, исполняющая среда C должна перебрать связный список структур данных. Обнаружив свободный блок достаточного размера, среда разбивает его, модифицируя указатели в узлах связного списка, чтобы сохранить его целостность. Для сравнения: в случае управляемой кучи выделение памяти для объекта означает просто прибавление некоторого значения к указателю, что намного быстрее. По сути, объект в управляемой куче выделяется почти так же быстро, как память в стеке потока! Кроме того, в большинстве куч (таких как куча исполняющей среды C) память для объектов выделяется в любой свободной области. Поэтому вполне вероятно, что несколько последовательно созданных объектов окажутся разделенными мегабайтами адресного пространства. В то же время в управляемой куче последовательно созданные объекты гарантированно будут расположены друг за другом.

Во многих приложениях объекты, выделяемые примерно в одно время, обычно связаны теснее, к тому же часто к ним обращаются примерно в одно время. Так, обычно сразу после объекта `FileStream` создается объект `BinaryWriter`. Затем приложение обращается к объекту `BinaryWriter`, внутренний код которого использует `FileStream`. В среде, поддерживающей сборку мусора, новые объекты располагаются в памяти непрерывно, что повышает производительность за счет близкого расположения ссылок. В частности, это значит, что рабочий набор процесса будет меньше, чем у подобного приложения, работающего в неуправляемой среде. Также, скорее всего, все объекты, используемые в программе, уместятся в кэше процессора. Приложение сможет получать доступ к этим объектам с феноменальной скоростью, так как процессор будет выполнять большинство своих операций без кэш-промахов, замедляющих доступ к оперативной памяти.

Итак, пока складывается впечатление, что управляемая куча намного превосходит кучу исполняющей среды C по простоте реализации и быстродействию. И все же есть одно «но», которое немного охладит ваш восторг. Все эти преимущества управляемой кучи являются следствием очень существенного допущения о бесконечности адресного пространства и ресурсов памяти. Конечно же, это не так, поэтому у управляемой кучи должен быть механизм, который делает такое допущение возможным. Это — сборщик мусора. Посмотрим, как он работает.

При вызове оператора `new` в области, выделяемой под объект, может не хватать свободного адресного пространства. Куча выясняет объем недостающей памяти и добавляет байты, необходимые для объекта, к адресу, заданному указателем `NextObjPtr`. Если результирующее значение выходит за пределы адресного пространства, значит, куча заполнена и следует выполнить сборку мусора.

ВНИМАНИЕ

Это весьма упрощенное описание работы сборщика мусора. На самом деле, сборка мусора начинается при заполнении поколения 0. Некоторые сборщики используют механизм поколений (*generations*), единственное назначение которого — повышение производительности. Идея такова: недавно созданные объекты составляют новое поколение, а созданные в самом начале жизненного цикла приложения — старое. Объекты поколения 0 созданы недавно и еще не проходили проверку алгоритмом сборки мусора. Объекты, оставшиеся после каждой сборки мусора, переходят в следующее поколение (например, в поколение 1). Деление объектов на поколения позволяет сборщику мусора ограничиться обработкой нескольких поколений вместо всей управляемой кучи. Подробнее о поколениях мы поговорим чуть позже, а пока для простоты будем считать, что сборка мусора происходит при заполнении кучи.

Алгоритм сборки мусора

Сборщик мусора проверяет наличие в куче больше не используемых приложением объектов, чтобы освободить занятую ими память (если даже после сборки мусора в куче не оказывается свободной памяти, оператор `new` вбрасывает исключение `OutOfMemoryException`). Откуда сборщик знает, используется объект приложением или нет? Это не такой-то простой вопрос.

У каждого приложения есть набор *корней* (*roots*). Корнем называется адрес указателя на объект ссылочного типа. Этот указатель содержит ссылку на объект в управляемой куче или равен `null`. Например, статическое поле (определенное в типе) считается корнем, как и любой параметр метода или локальная переменная. Корнями могут быть переменные только ссылочного, а не значимого типа. Рассмотрим конкретный пример и начнем с определения класса:

```
internal sealed class SomeType {  
    private TextWriter m_textWriter;  
  
    public SomeType(TextWriter tw) {  
        m_textWriter = tw;  
    }  
}
```

```

public void WriteBytes(Byte[] bytes) {
    for (Int32 x = 0; x < bytes.Length; x++) {
        m_textWriter.Write(bytes[x]);
    }
}
}

```

При первом вызове метода `WriteBytes` JIT-компилятор преобразует код метода на промежуточном языке в машинные команды процессора. Допустим, CLR работает на базе процессора x86, а метод `WriteBytes` компилируется в команды процессора, показанные на рис. 21.2 (комментарии справа поясняют, как машинный код соотносится с исходным).

00000000	push	edi	// Пролог
00000001	push	esi	
00000002	push	ebx	
00000003	mov	ebx, ecx	// ebx = this (экземпляр)
00000005	mov	esi, edx	// esi = байтовый массив
00000007	xor	edi, edi	// edi = 0
00000009	cmp	dword ptr [esi+4], 0	// сравнение bytes.Length с 0
0000000d	jle	0000002A	// если bytes.Length <= 0, переход к 2a
0000000f	mov	ecx, dword ptr [ebx+4]	// ecx = m_textWriter
00000012	cmp	edi, dword ptr [esi+4]	// сравнение x с bytes.Length
00000015	jae	0000002E	// если x >= bytes.Length, переход к 2e
00000017	movzx	edx, byte ptr [esi+edi+8]	// edx = bytes[x]
0000001c	mov	eax, dword ptr [ecx]	// eax = объект типа m_textWriter
0000001e	call	dword ptr [eax+00000008Ch]	// Вызов метода Write объекта m_textWriter
00000024	inc	edi	// x++
00000025	cmp	dword ptr [esi+4], edi	// сравнение bytes.Length с x
00000028	jg	0000000F	// если bytes.Length > x, переход к f
0000002a	pop	ebx	// Эпилог
0000002b	pop	esi	
0000002c	pop	edi	
0000002d	ret		// Возврат управления вызвавшему коду
0000002e	call	76B6E337	// Генерация исключения IndexOutOfRangeException
00000033	int	3	// Приостановка отладчика

Рис. 21.2. Созданный JIT-компилятором машинный код с иерархией корней

Генерация машинного кода JIT-компилятором сопровождается созданием внутренней таблицы. Логически каждая ее строка указывает диапазон смещений байтов машинных кодов процессора для этого метода, а также для каждого диапазона — набор адресов памяти и регистры процессора, содержащие корни. Для метода `WriteBytes` в этой таблице видно, что регистр `EBX` сначала является корнем со смещением `0x00000003`, регистр `ESI` — корнем со смещением `0x00000005`, а регистр `ECX` — корнем со смещением `0x0000000f`. Все эти регистры перестают быть корнями в конце цикла (смещение `0x00000028`). Также обратите внимание, что регистр `EAX` является корнем между `0x0000001c` и `0x0000001e`. Регистр `EDI` служит для хранения значения типа `Int32`, представленной переменной `x`

в исходном коде. Так как `Int32` — значимый тип, JIT-компилятор не считает регистр `EDI` корнем.

Метод `WriteBytes` довольно прост, и все используемые в нем переменные могут быть зарегистрированы. Более сложный метод может задействовать все имеющиеся регистры процессора, а некоторые корни будут располагаться в памяти относительно стекового фрейма метода. Также учтите, что в архитектуре x86 CLR передает первые два аргумента методу через регистры `ECX` и `EDX`. Для экземплярных методов в качестве первого аргумента выступает указатель `this`, всегда передаваемый в регистре `ECX`. Именно поэтому я знаю, что в случае метода `WriteBytes` указатель `this` передается в регистре `ECX` и сохраняется в регистре `EBX` сразу после пролога метода, а также то, что аргумент `bytes` передается в регистре `EDX` и сохраняется в регистре `ESI` после пролога.

Если бы сборка мусора началась во время исполнения кода со смещением `0x00000017` в методе `WriteBytes`, сборщик мусора знал бы, что объекты, на которые ссылаются регистры `EBX` (аргумент `this`), `ESI` (аргумент `bytes`) и `ECX` (поле `m_textWriter`), являются корнями, а объекты, соответствующие им в куче, нельзя считать мусором. Кроме того, сборщик может пройти по стеку вызовов потока и определить корни всех вызывающих методов, изучив внутреннюю таблицу каждого из них. Для получения набора корней, хранимых в статических полях, сборщик мусора просматривает все типы.

Начиная работу, сборщик предполагает, что все объекты в куче — мусор. Иначе говоря, он считает, что в стеке потока нет переменных, ссылающихся на объекты в куче, а также то, что на объекты в куче не ссылаются регистры процессора и статические поля. Затем сборщик переходит к этапу сборки мусора, называемому *маркировкой* (*marking*). Он проходит по стеку потока и проверяет все корни. Если оказывается, что корень ссылается на объект, в поле `SyncBlockIndex` этого объекта включается бит — это и есть признак маркировки объекта. Например, сборщик мусора может найти локальную переменную, указывающую на объект в куче. На рис. 21.3 показана куча с несколькими объектами, в которой корни приложения напрямую ссылаются на объекты `A`,

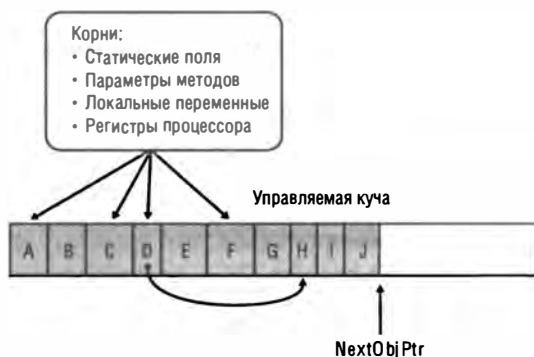


Рис. 21.3. Управляемая куча перед сборкой мусора

C, *D* и *F*. Все эти объекты маркируются. При маркировке объекта *D* сборщик мусора обнаруживает, что в этом объекте есть поле, ссылающееся на объект *H*. Поэтому объект *H* также помечается. Затем сборщик продолжает рекурсивный просмотр всех достижимых объектов.

После маркировки корня и объекта, на который ссылается его поле, сборщик мусора проверяет следующий корень и продолжает маркировать объекты. Встретив уже маркированный объект, сборщик мусора останавливается. Это нужно по двум причинам. Во-первых, заметно повышается быстродействие, так как сборщик проходит набор объектов не больше одного раза, во-вторых, исключается возможность бесконечных циклов в случае замкнутых связных списков объектов.

После проверки всех корней куча содержит набор маркированных и немаркированных объектов. Маркированные объекты, в отличие от немаркированных, достижимы из кода приложения. Недостижимые объекты считаются мусором, а занимаемая ими память становится доступной для освобождения. Затем сборщик переходит к следующему этапу сборки мусора, называемому *сжатием* (compact phase). Теперь он проходит кучу линейно в поисках непрерывных блоков немаркированных объектов, то есть мусора.

Небольшие блоки сборщик не трогает, а в больших непрерывных блоках он перемещает вниз все «немусорные» объекты, сжимая при этом кучу.

Естественно, перемещение объектов в памяти делает недействительными все переменные и регистры процессора, содержащие указатели на объекты. Поэтому сборщик мусора должен вновь проверить и обновить все корни приложения, чтобы все значения корней указывали на новые адреса объектов в памяти. Кроме того, если объект содержит поле, указывающее на другой перемещенный объект, сборщик должен исправить и эти поля. После сжатия памяти кучи в указатель `NextObjPtr` управляемой кучи заносится первый адрес за последним объектом, не являющимся мусором. На рис. 21.4 показана управляемая куча после сборки мусора.

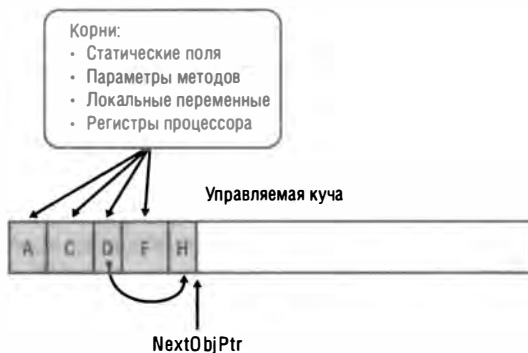


Рис. 21.4. Управляемая куча после сборки мусора

Как видите, сборка мусора сильно бьет по производительности — это основной недостаток управляемой кучи. Однако не следует забывать, что сборка мусора начинается только после заполнения поколения 0, а до этого управляемая куча работает намного быстрее, чем куча исполняющей среды С. Наконец, ряд оптимизаций сборщика мусора в CLR существенно повысили производительность сборки мусора.

Программист должен извлечь для себя несколько важных уроков из этого обсуждения. Начнем с того, что вам больше не требуется код, управляющий временем жизни объектов, которые используются приложением. В результате исключается возможность появления двух видов ошибок, описанных в начале этой главы. Во-первых, исключается утечка объектов, так как все объекты, недоступные от корней приложения, рано или поздно уничтожает сборщик мусора. Во-вторых, благодаря сборке мусора невозможно получить доступ к освобожденному объекту. Ведь доступные объекты не освобождаются, а если объект недостижим, приложение не получает к нему доступа. Кроме того, поскольку при сборке мусора происходит сжатие памяти, управляемые объекты не фрагментируют виртуальное адресное пространство процесса. Иногда это становилось серьезным препятствием при работе с неуправляемой кучей, а в управляемой куче это уже не проблема. Есть одно исключение: при хранении больших объектов все-таки возможна фрагментация кучи, но об этом мы поговорим позже.

ВНИМАНИЕ

Статическое поле типа поддерживает любой объект, на который ссылается, бессрочно или до выгрузки домена приложений с загруженными типами. Чаще всего утечка памяти возникает при ссылке статического поля на коллекцию, в которую добавляются элементы. Статическое поле сохраняет коллекцию, которая, в свою очередь, сохраняет все свои элементы. Поэтому старайтесь по возможности избегать статических полей.

Сборка мусора и отладка

Как показано на рис. 21.2, на аргумент `bytes` метода (хранящийся в регистре `ESI`) нет ссылок после команды процессора со смещением `0x00000028`. Это значит, что массив `Byte`, на который ссылается аргумент `bytes`, может быть уничтожен сборщиком мусора в любой момент после выполнения команды со смещением `0x00000028` (при условии, что в приложении нет других корней, ссылающихся на этот массив). Иначе говоря, став недостижимым, объект превращается в кандидата на удаление — объекты далеко не всегда «доживают»

до завершения работы метода. Для приложения эта особенность может иметь интересные последствия. Например, рассмотрим следующий код:

```
using System;
using System.Threading;

public static class Program {
    public static void Main() {
        // Создание объекта Timer, вызывающего метод TimerCallback
        // каждые 2000 миллисекунд
        Timer t = new Timer(TimerCallback, null, 0, 2000);

        // Ждем, когда пользователь нажмет Enter
        Console.ReadLine();
    }

    private static void TimerCallback(Object o) {
        // Вывод даты/времени вызова этого метода
        Console.WriteLine("In TimerCallback: " + DateTime.Now);
        // Принудительный вызов сборщика мусора в этой программе
        GC.Collect();
    }
}
```

Скомпилируйте этот код из командной строки, не используя никаких специальных параметров компилятора. Затем, запустив полученный исполняемый файл, вы увидите, что метод `TimerCallback` вызывается всего один раз!

Изучив приведенный код, можно подумать, что метод `TimerCallback` будет вызываться каждые 2000 миллисекунд. В итоге создается объект `Timer`, на который ссылается переменная `t`. Поскольку таймер существует, он должен срабатывать. Обратите внимание, что в методе `TimerCallback` процедура сборки мусора вызывается принудительно методом `GC.Collect()`.

После запуска сборщик мусора предполагает, что все объекты в куче недостижимы (то есть являются мусором), в том числе объект `Timer`. Затем сборщик проверяет корни приложения и видит, что метод `Main` не использует переменную `t` после присвоения ей значения. Поэтому в приложении нет переменной, ссылающейся на объект `Timer`, и сборщик мусора освобождает занятую им память. В итоге таймер останавливается, а метод `TimerCallback` вызывается всего один раз.

Допустим, вы используете отладчик для метода `Main`, а сборка мусора происходит сразу после присвоения переменной `t` адреса нового объекта `Timer`. Что случится, если затем вы попытаетесь просмотреть объект, на который ссылается `t`, в окне **Quick Watch** отладчика? Отладчик не сможет показать объект, потому что тот был удален сборщиком мусора. Для многих разработчиков такой вариант развития событий стал бы очень неприятным сюрпризом, поэтому специалисты Microsoft предложили другое решение.

Когда JIT-компилятор преобразует IL-код метода в машинный код, он проверяет, была ли сборка, в которой определен метод, скомпилирована без оптимизации и выполняется ли процесс с отладчиком. Если то и другое верно, JIT-компилятор генерирует внутреннюю таблицу корней метода таким образом, чтобы искусственно продлить время жизни всех переменных до завершения метода. Иначе говоря, JIT-компилятор занимается самообманом, убеждая себя в том, что переменная `t` метода `Main` должна жить до завершения метода. Поэтому при сборке мусора сборщик считает эту переменную корнем, а объект `Timer`, на который она ссылается, доступным. Объект не будет удален, а значит, продолжится вызов метода `TimerCallback`, пока метод `Console.ReadLine` не вернет управление и пока существует метод `Main`. Чтобы убедиться в этом, запустите тот же исполняемый файл в режиме отладки — метод `TimerCallback` будет вызываться регулярно.

Теперь перекомпилируйте программу из командной строки, установив переключатель `/debug+` компилятора C#. Запустив полученный исполняемый файл, вы увидите, что метод `TimerCallback` вызывается многократно, даже когда программа запущена без отладчика! Что же происходит?

При компиляции метода JIT-компилятор смотрит, чтобы сборка, определяющая метод, содержала атрибут `System.Diagnostics.DebuggableAttribute`, а аргумент `isJITOptimizerDisabled` его конструктора был равен `true`. Обнаружив, что этот атрибут задан, он также скомпилирует метод, искусственно продлевая время жизни всех переменных до завершения метода. Если установлен переключатель `/debug+`, компилятор C# добавляет этот атрибут в готовую сборку. Учтите, что параметр `/optimize+` компилятора C# может вновь включить режим оптимизации, поэтому при подобном эксперименте этот параметр компилятора указывать не следует.

Таким образом, JIT-компилятор помогает своевременно выполнить отладку. Теперь можно запустить приложение в обычном режиме (без отладчика), и если метод будет вызван, JIT-компилятор искусственно увеличит время жизни переменных до его окончания. Затем, если к процессу будет добавлен отладчик, можно вставить точку останова в ранее скомпилированный метод и изучить переменные.

Теперь вы знаете, как создать программу, которая работает на этапе отладки, но не работает должным образом в готовой версии. Но программа, корректно работающая только в режиме отладки, бесполезна. Поэтому необходимо средство, обеспечивающее работу программы независимо от типа ее сборки.

Можно попробовать изменить метод `Main` следующим образом:

```
public static void Main() {  
    // Создание объекта Timer, вызывающего метод TimerCallback каждые 2000 мс  
    Timer t = new Timer(TimerCallback, null, 0, 2000);  
  
    // Ждем, когда пользователь нажмет Enter  
    Console.ReadLine();  
}
```

```
// Создаем ссылку на t после ReadLine
// (в ходе оптимизации эта строка удаляется)
t = null;
}
```

Все равно после компиляции этого кода (без параметра `/debug+`) и запуска полученного исполняемого файла (без отладчика) выяснится, что метод `Timer_Callback` вызывается всего раз. Дело здесь в том, что JIT-компилятор является оптимизирующим, а приравнивание локальной переменной или переменной-параметра к `null` равнозначно отсутствию ссылки на эту переменную. Иначе говоря, JIT-компилятор в ходе оптимизации полностью убирает строку `t = null;` из программы, из-за этого она работает не так, как хотелось бы. Вот как правильно следовало изменить метод `Main`:

```
public static void Main() {
    // Создание объекта Timer, вызывающего метод TimerCallback каждые 2000 мс
    Timer t = new Timer(TimerCallback, null, 0, 2000);

    // Ждем, когда пользователь нажмет Enter
    Console.ReadLine();

    // Создаем ссылку на переменную t после ReadLine
    // (t не удаляется сборщиком мусора
    // до возвращения управления методом Dispose)
    t.Dispose();
}
```

Теперь, скомпилировав этот код (без параметра `/debug+`) и запустив полученный исполняемый файл (без отладчика), вы увидите, что метод `TimerCallback` вызывается несколько раз, и программа работает корректно. Это объясняется тем, что объект, на который ссылается переменная `t`, не должен удаляться, чтобы можно было вызвать метод экземпляра `Dispose` (значение `t` нужно передать методу `Dispose` как аргумент `this`).

ПРИМЕЧАНИЕ

Завершив это обсуждение, не стоит преждевременно беспокоиться о сборке мусора среди ваших собственных объектов. Класс `Timer` использовался в обсуждении только из-за своего специфического отсутствующего у других классов поведения. Дело в том, что присутствие в куче объекта `Timer` приводит к объединению потоков и периодическому вызову метода. Другие типы не в состоянии так себя вести. К примеру, наличие в памяти объекта `String` не имеет никаких последствий. Строка просто находится в куче. Именно поэтому, чтобы продемонстрировать, как работают корни и как время жизни объекта связано с отладчиком, я использовал объект `Timer`. Но при этом основной вопрос состоял не в том, как растянуть время жизни объекта. Время жизни остальных объектов определяется приложением автоматически.

Освобождение ресурсов при помощи механизма финализации

Итак, мы познакомились с азами сборки мусора и управляемой кучей, а также тем, как сборщик мусора освобождает память объекта. На наше счастье, большинству типов для работы требуется только память. Так, типы `String`, `Attribute`, `Delegate` и `Exception` всего лишь манипулируют байтами в памяти. Но есть и типы, которым помимо памяти необходимы машинные ресурсы.

Например, типу `System.IO.FileStream` нужно открыть файл (машинный ресурс) и сохранить его дескриптор. Затем при помощи этого дескриптора методы `Read` и `Write` данного типа работают с файлом. Аналогично, тип `System.Threading.Mutex` открывает мьютекс, являющийся объектом ядра Windows (машинный ресурс), сохраняет его описатель, который использует при вызове методов объекта `Mutex`.

Финализацией (finalization) называется поддерживаемый CLR механизм, позволяющий объекту выполнить корректную очистку, прежде чем сборщик мусора освободит занятую им память. Любой тип, выполняющий функцию оболочки машинного ресурса, например файла, сетевого соединения, сокета, мьютекса и др., должен поддерживать финализацию. Для этого в типе реализуют метод финализации. Определив, что объект стал мусором, сборщик вызывает метод финализации объекта (если он есть). Иначе говоря, реализация в типе метода финализации означает, что все его объекты имеют право на исполнение «последнего желания перед экзекуцией».

Группа разработчиков C# из Microsoft посчитала, что метод финализации отличается от остальных и требует специального синтаксиса в языке программирования (точно так же, как в C# специальный синтаксис используется для определения конструктора). Поэтому для определения метода финализации в C# перед именем класса нужно добавить знак тильды (~):

```
internal sealed class SomeType {  
    // Метод финализации  
    ~SomeType() {  
        // Код метода финализации  
    }  
}
```

Скомпилировав этот код и проверив полученную сборку с помощью утилиты `ILDasm.exe`, вы увидите, что компилятор C# внес метод с именем `Finalize` в метаданные этого модуля. При изучении IL-кода метода `Finalize` также становится ясно, что код в теле метода генерируется в блок `try`, а вызов метода `base.Finalize` — в блок `finally`.

ВНИМАНИЕ

Разработчики, хорошо знакомые с C++, заметят, что специальный синтаксис, используемый в C# для определения метода финализации, напоми-

нает синтаксис деструктора C++. Действительно, в предыдущих версиях спецификации C# этот метод назывался деструктором (destructor). Однако метод финализации работает совсем не так, как неуправляемый деструктор C++, что приводит в замешательство многих разработчиков, переходящих с одного языка на другой.

Беда в том, что разработчики ошибочно полагают, что использование синтаксиса деструктора означает в C# детерминированное уничтожение объектов типа, как это происходит в C++. Но CLR не поддерживает детерминированное уничтожение, поэтому C# не может предоставить этот механизм.

Метод финализации обычно вызывает Win32-функцию `CloseHandle`, передавая ей дескриптор машинного ресурса. В типе `FileStream` определено поле дескриптора файла, указывающее на этот машинный ресурс. В типе `FileStream` также определен метод финализации, внутренний код которого вызывает метод `CloseHandle`, передавая последнему поле дескриптора файла. Это гарантирует, что собственный дескриптор файла будет закрыт, когда управляемый объект `FileStream` станет мусором. Если у типа, служащего оболочкой для машинного ресурса, нет метода финализации, машинный ресурс не будет закрыт, и возникнет утечка ресурса, продолжающая до завершения процесса, в ходе которого ОС освобождает машинные ресурсы.

Гарантированная финализация с использованием типов `CriticalFinalizerObject`

Для удобства разработчиков в пространстве имен `System.Runtime.Constrained-Execution` был определен класс `CriticalFinalizerObject` следующего вида:

```
public abstract class CriticalFinalizerObject {  
    protected CriticalFinalizerObject() { /* здесь нет никакого кода */ }  
    // Далее следует метод финализации  
    ~CriticalFinalizerObject() { /* здесь нет никакого кода */ }  
}
```

Уверен, вам покажется, что ничего особенного в этом классе нет, но CLR работает с ним и с его производными не так, как с другими классами, а наделяет этот класс тремя замечательными возможностями.

- ❑ При первом создании любого объекта, производного от типа `CriticalFinalizerObject`, CLR автоматически запускает JIT-компилятор, компилирующий все методы финализации в иерархии наследования. Компиляция этих методов после создания объекта гарантирует, что машинные ресурсы освободятся, как только объект станет мусором. Без компиляции метода финализации возможно лишь выделение и использование ресурсов, но не их освобождение. При недостатке памяти CLR не сможет найти достаточно памяти для компиляции метода финализации. В этом случае метод не

будет исполнен, что приведет к утечке машинных ресурсов. Также ресурсы не освобождаются, если код в методе финализации содержит ссылку на тип в другой сборке, которая не была обнаружена CLR.

- ❑ CLR вызывает метод финализации для типов, производных от `CriticalFinalizerObject`, после вызова методов финализации для типов, производных от `CriticalFinalizerObject`. Благодаря этому классы управляемых ресурсов, имеющие метод финализации, могут успешно обращаться к объектам, производным от `CriticalFinalizerObject`, в их методах финализации. Так, метод финализации класса `FileStream` может сбросить данные из буфера памяти на диск в полной уверенности, что дисковый файл все еще открыт.
- ❑ CLR вызывает метод финализации для типов, производных от `CriticalFinalizerObject`, если домен приложения был аварийно завершен хост-приложением (например, `Microsoft SQL Server` или `Microsoft ASP.NET`). Это гарантирует освобождение машинных ресурсов даже в том случае, когда хост-приложение больше не доверяет работающему внутри него управляемому коду.

Тип `SafeHandle` и его потомки

В Microsoft понимают, что из всех машинных ресурсов чаще всего используются ресурсы Windows, а также то, что работа с большинством ресурсов Windows выполняется через дескрипторы (32-разрядные значения в 32-разрядной системе и 64-разрядные — в 64-разрядной). Чтобы облегчить жизнь разработчикам, в пространство имен `System.Runtime.InteropServices` был добавлен класс `SafeHandle` следующего вида (комментарии мои):

```
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable {  
    // Это дескриптор машинного ресурса  
    protected IntPtr handle;  
  
    protected SafeHandle(IntPtr invalidHandleValue, Boolean ownsHandle) {  
        this.handle = invalidHandleValue;  
        // Если значение ownsHandle равно true, машинный ресурс закрывается,  
        // когда этот производный от SafeHandle объект, уничтожается  
        // сборщиком мусора  
    }  
  
    protected void SetHandle(IntPtr handle) {  
        this.handle = handle;  
    }  
  
    // Явно освободить ресурс можно, вызвав метод Dispose или Close  
    public void Dispose() { Dispose(true); }  
    public void Close() { Dispose(true); }
```

```
// Здесь подойдет стандартная реализация метода Dispose
// Настоятельно не рекомендуется переопределять этот метод!
protected virtual void Dispose(Boolean disposing) {
    // В стандартной реализации аргумент, вызывающий метод
    // Dispose, игнорируется
    // Если ресурс уже освобожден, управление возвращается коду
    // Если значение ownsHandle равно false, управление возвращается
    // Установка флага, означающего, что этот ресурс был освобожден
    // Вызов виртуального метода ReleaseHandle
    // Вызов GC.SuppressFinalize(this), отменяющий вызов метода финализации
    // Если значение ReleaseHandle равно true, управление возвращается коду
    // Запуск ReleaseHandleFailed Managed Debugging Assistant (MDA)
}

// Здесь подходит стандартная реализация метода финализации
// Настоятельно не рекомендуется переопределять этот метод!
~SafeHandle() { Dispose(false); }

// Производный класс переопределяет этот метод,
// чтобы реализовать код, освобождающий ресурс
protected abstract Boolean ReleaseHandle();

public void SetHandleAsInvalid() {
    // Установка флага, означающего, что этот ресурс был освобожден
    // Вызов GC.SuppressFinalize(this), отменяющий вызов метода финализации
}

public Boolean IsClosed {
    get {
        // Возвращение флага, показывающего, был ли ресурс освобожден
    }
}

public abstract Boolean IsInvalid {
    get {
        // Производный класс переопределяет это свойство
        // Реализация должна вернуть значение true, если значение
        // дескриптора не представляет ресурс (обычно это значит,
        // что дескриптор равен 0 или @1)
    }
}

// Эти три метода имеют отношение к безопасности и подсчету ссылок
// Подробнее о них рассказывается в конце этого раздела
public void DangerousAddRef(ref Boolean success) {...}
public IntPtr DangerousGetHandle() {...}
public void DangerousRelease() {...}
}
```

Рассматривая класс `SafeHandle`, прежде всего нужно отметить, что он наследует от класса `CriticalFinalizerObject`. Это гарантирует, что CLR будет обращаться с ним не так, как с другими классами. Кроме того, это абстрактный класс: предполагается, что будет создан еще один производный от `SafeHandle` класс, который переопределит защищенный конструктор, абстрактный метод `ReleaseHandle` и абстрактное свойство `IsValid` метода доступа `get`.

В Windows большинство дескрипторов считаются недействительными при равенстве их значения 0 или -1. Пространство имен `Microsoft.Win32.SafeHandle` содержит еще один вспомогательный класс `SafeHandleZeroOrMinusOneIsInvalid` следующего вида:

```
public abstract class SafeHandleZeroOrMinusOneIsInvalid : SafeHandle {
    protected SafeHandleZeroOrMinusOneIsInvalid(Boolean ownsHandle)
        : base(IntPtr.Zero, ownsHandle) {
    }

    public override Boolean IsValid {
        get {
            if (base.handle == IntPtr.Zero) return true;
            if (base.handle == (IntPtr) (@1)) return true;
            return false;
        }
    }
}
```

Обратите внимание, что класс `SafeHandleZeroOrMinusOneIsInvalid` является абстрактным, поэтому надо создать дочерний класс, который переопределит защищенный конструктор и абстрактный метод `ReleaseHandle`. На платформе `Microsoft .NET Framework` есть два открытых класса, производных от `SafeHandleZeroOrMinusOneIsInvalid`, — `SafeFileHandle` и `SafeWaitHandle`. Они также входят в пространство имен `Microsoft.Win32.SafeHandles`. А так выглядит класс `SafeFileHandle`:

```
public sealed class SafeFileHandle : SafeHandleZeroOrMinusOneIsInvalid {
    public SafeFileHandle(IntPtr preexistingHandle, Boolean ownsHandle)
        : base(ownsHandle) {
        base.SetHandle(preexistingHandle);
    }

    protected override Boolean ReleaseHandle() {
        // Сообщить Windows, что машинный ресурс нужно закрыть
        return Win32Native.CloseHandle(base.handle);
    }
}
```

Класс `SafeWaitHandle` реализован сходным образом. Единственной причиной наличия у разных классов похожих реализаций является обеспечение безопасности типов: компилятор не позволит использовать файловый дескриптор

в качестве аргумента метода, принимающего дескриптор блокировки, и наоборот. Метод `ReleaseHandle` класса `SafeRegistryHandle` вызывает Win32-функцию `RegCloseKey`.

Жаль, что на платформе .NET Framework отсутствуют дополнительные классы, служащие оболочкой различных машинных ресурсов, например таких, как `SafeProcessHandle`, `SafeThreadHandle`, `SafeTokenHandle`, `SafeFileMappingHandle`, `SafeViewOfFileHandle` (его метод `ReleaseHandle` вызывал бы Win32-функцию `UnmapViewOfFile`), `SafeRegistryHandle` (его метод `ReleaseHandle` вызывал бы Win32-функцию `RegCloseKey`), `SafeLibraryHandle` (его метод `ReleaseHandle` вызывал бы Win32-функцию `FreeLibrary`), `SafeLocalAllocHandle` (его метод `ReleaseHandle` вызывал бы Win32-функцию `LocalFree`) и т. п.

Все эти классы (а также некоторые другие) есть в библиотеке FCL. Но широкой аудитории они не известны, являясь внутренними классами `MSCorLib.dll` или `System.dll`. Microsoft не афиширует эти классы, чтобы не выполнять их полное тестирование и не тратить время на их документирование. Если же вам придется с ними столкнуться, рекомендую воспользоваться утилитой `ILDasm.exe` или другим IL-декомпилятором, чтобы извлечь код этих классов и интегрировать его в исходный текст программы. Все эти классы просты в реализации и их несложно написать самостоятельно.

Взаимодействие с неуправляемым кодом с использованием типов `SafeHandle`

Вы уже имели шанс убедиться в полезности классов, производных от `SafeHandle`. Ведь они гарантируют освобождение машинного ресурса в ходе сборки мусора. Стоит добавить, что у типа `SafeHandle` есть еще две функциональные особенности. Во-первых, когда производные от него типы используются в сценариях взаимодействия с неуправляемым кодом, им гарантирован особый подход со стороны CLR. Вот пример:

```
using System;
using System.Runtime.InteropServices;
using Microsoft.Win32.SafeHandles;

internal static class SomeType {
    [DllImport("Kernel32", CharSet=CharSet.Unicode, EntryPoint="CreateEvent")]

    // Этот прототип неустойчив к сбоям
    private static extern IntPtr CreateEventBad(
        IntPtr pSecurityAttributes, Boolean manualReset,
        Boolean initialState, String name);
}
```

продолжение ⇨

```
// Этот прототип устойчив к сбоям
[DllImport("Kernel32", CharSet=CharSet.Unicode, EntryPoint="CreateEvent")]
private static extern SafeWaitHandle CreateEventGood(
    IntPtr pSecurityAttributes, Boolean manualReset,
    Boolean initialState, String name);

public static void SomeMethod() {
    IntPtr handle = CreateEventBad(IntPtr.Zero, false, false, null);
    SafeWaitHandle swh = CreateEventGood(IntPtr.Zero, false, false, null);
}
```

Обратите внимание, что прототип метода `CreateEventBad` возвращает `IntPtr`. В версиях .NET Framework, предшествующих версии 2.0, класса `SafeHandle` не было, и для представления обработчиков приходилось использовать тип `IntPtr`. Группа разработчиков CLR из Microsoft обнаружила неустойчивость этого кода к сбоям. После вызова метода `CreateEventBad` (создающего ресурс машинного события) возможна ситуация, когда исключение `ThreadAbortException` вбрасывается до присвоения дескриптора переменной `handle`. В таких редких случаях в управляемом коде образуется утечка машинного ресурса. И событие можно закрыть только одним способом — завершив процесс.

С выходом версии 2.0 .NET Framework для устранения этой утечки ресурсов стал применяться класс `SafeHandle`. Обратите внимание, что прототип метода `CreateEventGood` возвращает `SafeWaitHandle`, а не `IntPtr`. При вызове метода `CreateEventGood` CLR вызывает Win32-функцию `CreateEvent`. Когда эта функция возвращает управление управляемому коду, CLR «знает», что `SafeWaitHandle` является производным от `SafeHandle`. Поэтому CLR автоматически создает экземпляр класса `SafeWaitHandle`, передавая ему полученное от метода `CreateEvent` значение дескриптора. Обновление объекта `SafeWaitHandle` и присвоение дескриптора происходят в неуправляемом коде, который не может быть прерван исключением `ThreadAbortException`. В результате в управляемом коде не может возникнуть утечка этого машинного ресурса. А в итоге объект `SafeWaitHandle` удаляется сборщиком мусора и вызывается его метод финализации, обеспечивающий освобождение памяти.

И наконец, классы, производные от `SafeHandle`, гарантируют, что никто не сможет воспользоваться возможными брешами в системе безопасности. Проблема в том, что один из потоков может попытаться использовать машинный ресурс, освобождаемый другим потоком. Это называется атакой с повторным использованием дескрипторов. Класс `SafeHandle` предотвращает это нарушение безопасности благодаря подсчету ссылок. В нем определено закрытое поле, исполняющее роль счетчика. Когда производному от `SafeHandle` объекту присваивается корректный дескриптор, счетчик приравнивается к 1. Всякий раз, когда производный от `SafeHandle` объект передается как аргумент неуправляемому методу, CLR автоматически увеличивает значение счетчика на единицу. Когда неуправляемый метод возвращает управление управляемому

коду, CLR уменьшает значение счетчика на ту же величину. Например, вот как выглядит прототип Win32-функции SetEvent:

```
[DllImport("Kernel32", ExactSpelling=true)]  
private static extern Boolean SetEvent(SafeWaitHandle swh);
```

При вызове этого метода и передаче ему ссылки на объект SafeWaitHandle CLR увеличивает значение счетчика перед вызовом и уменьшает значение счетчика сразу после вызова. Разумеется, счетчик работает в безопасном режиме. Как это повышает безопасность? Если другой поток попытается освободить машинный ресурс, оболочкой которого является объект SafeHandle, CLR узнает, что это ему не разрешено, потому что данный ресурс используется неуправляемой функцией. Когда функция вернет управление программе, значение счетчика будет приравнено к 0 и ресурс освободится.

При написании или вызове кода, работающего с дескриптором, например IntPtr, к нему можно обратиться из объекта SafeHandle, но подсчет ссылок придется выполнять явно с помощью методов DangerousAddRef и DangerousRelease объекта SafeHandle. Обращение к исходному дескриптору происходит через метод DangerousGetHandle.

И конечно, нельзя не упомянуть о классе CriticalHandle, также определенном в пространстве имен System.Runtime.InteropServices. Он работает точно так же, как и SafeHandle, но не поддерживает подсчет ссылок. В CriticalHandle и производных от него классах безопасность принесена в жертву повышению производительности (за счет отказа от счетчиков). Как и у SafeHandle, у CriticalHandle есть два производных типа — CriticalHandleMinusOneIsInvalid и CriticalHandleZeroOrMinusOneIsInvalid. Поскольку Microsoft отдает предпочтение безопасности, а не производительности системы, в библиотеке классов нет типов, производных от этих двух классов. Я рекомендую использовать типы, производные от CriticalHandle, только в случаях, когда высокая производительность крайне необходима и оправдывает некоторое ослабление защиты.

Финализация управляемых ресурсов

ВНИМАНИЕ

Некоторые полагают, что к управляемым ресурсам не следует применять механизм финализации. В принципе, я с ними согласен. Поэтому вы можете пропустить этот раздел. Финализация управляемых ресурсов — это высший пилотаж в разработке кода, прибегать к которому допустимо лишь в крайних случаях. Вы должны досконально знать код, вызываемый из метода финализации. Более того, необходима уверенность в устойчивости кода даже после выхода новых версий, а также в том, что код, вызываемый из метода финализации, не использует другой уже финализированный объект.

Финализация обычно требуется исключительно для освобождения машинного ресурса, но иногда бывает полезной и для освобождения управляемых ресурсов. Вот класс, который заставляет компьютер давать звуковой сигнал каждый раз, когда сборщик мусора начинает свою работу:

```
internal sealed class GCBeep {
    // Это метод финализации
    ~GCBeep() {
        // Идет финализация – дать сигнал
        Console.Beep();

        // Если домен приложения не выгружается, а процесс не завершается,
        // создать новый объект, финализация которого произойдет
        // в ходе следующей сборки мусора
        if (!AppDomain.CurrentDomain.IsFinalizingForUnload()
            &&!Environment.HasShutdownStarted)
            new GCBeep();
    }
}
```

Для работы с этим классом достаточно создать один его экземпляр. Затем при каждой сборке мусора будет вызываться метод финализации объекта, вызывающий метод Beep и создающий новый объект GCBeep. Метод финализации этого объекта GCBeep вызывается при следующей сборке мусора. Вот пример программы, демонстрирующей применение класса GCBeep:

```
public static class Program {
    public static void Main() {
        // После создания объекта GCBeep каждый раз, когда
        // начинается сборка мусора, подается звуковой сигнал
        new GCBeep();

        // Создать много 100-байтовых объектов
        for (Int32 x = 0; x < 10000; x++) {
            Console.WriteLine(x);
            Byte[] b = new Byte[100];
        }
    }
}
```

ПРИМЕЧАНИЕ

Класс GCBeep, конечно, полезен, но я считаю намного более полезным класс GCNotification, дающий возможность анализировать приложения, а значит, получать информацию о том, каким именно образом они используют память. Подробно этот класс рассмотрен далее.

Также учтите, что метод финализации будет вызван, даже если конструктор экземпляра этого типа вбросит исключение. То есть ваш метод финализации

не должен предполагать, что объект находится в корректном согласованном состоянии. Это демонстрирует следующий код:

```
internal sealed class TempFile {
    private String m_filename = null;
    private FileStream m_fs;

    public TempFile(String filename) {
        // Следующая строка может вбросить исключение
        m_fs = new FileStream(filename, FileMode.Create);

        // Сохраняем имя этого файла
        m_filename = filename;
    }

    ~TempFile() {          // Это метод финализации
        // Здесь надо бы проверить, не пустое ли имя файла,
        // поскольку нельзя быть уверенным, что оно было
        // инициализировано в конструкторе
        if (m_filename != null)
            File.Delete(m_filename);
    }
}
```

Можно написать этот код и так:

```
internal sealed class TempFile {
    private String m_filename;
    private FileStream m_fs;

    public TempFile(String filename) {
        try {
            // Следующая строка может вбросить исключение
            m_fs = new FileStream(filename, FileMode.Create);

            // Сохранить имя этого файла
            m_filename = filename;
        }
        catch {
            // Если что-то пойдет не так, запретить
            // сборщику мусора вызывать метод финализации
            // 0 SuppressFinalize см. далее в этой главе
            GC.SuppressFinalize(this);

            // Уведомить вызывающий код об ошибке
            throw;
        }
    }
}
```

продолжение ➤

```
~TempFile() {           // Метод финализации
    // Условная инструкция теперь не нужна, поскольку этот код
    // выполняется только после успешного исполнения конструктора
    File.Delete(m_filename);
}
}
```

При конструировании типа метода финализации лучше избегать по ряду причин.

- ❑ Выделение памяти для объектов, поддерживающих финализацию, занимает больше времени, так как указатели на них должны размещаться в списке финализации.
- ❑ Объекты, поддерживающие финализацию, переходят в старшие поколения, что повышает нагрузку на память и не позволяет освободить память объекта в тот момент, когда сборщик мусора считает его мусором. Кроме того, все объекты, на которые прямо или косвенно ссылается этот объект, тоже переходят в старшие поколения (о поколениях и переходах между ними см. далее).
- ❑ Объекты, поддерживающие финализацию, замедляют работу приложения, потому что каждое удаление объекта при сборке мусора требует дополнительного расхода ресурсов.

Следует также учесть невозможность контролировать момент исполнения метода финализации. Этот метод вызывается при сборке мусора, которая начинается тогда, когда приложению требуется дополнительная память. Кроме того, CLR не гарантирует определенного порядка вызова методов финализации. Поэтому лучше не создавать метод финализации, вызывающий другие объекты, в типе которых определен метод финализации, поскольку эти объекты могут оказаться уже финализированными. Обращаться к экземплярам значимого или ссылочного типа, в которых не определен метод финализации, вполне допустимо. Также следует быть осторожным при вызове статических методов, поскольку их внутренний код может обращаться к финализированным объектам, что сделает их работу непредсказуемой.

Мотивы вызова методов финализации

Методы финализации вызываются по окончании сборки мусора, которая происходит в результате одного из пяти следующих событий:

- ❑ **Заполнение поколения 0 приводит к запуску сборщика мусора.** Это событие намного чаще остальных становится причиной вызова метода финализации, так как является естественным следствием создания новых объектов в ходе выполнения приложения.

- ❑ **Явный вызов статического метода `Collect` объекта `System.GC`.** Код может явно потребовать у CLR сборки мусора. Хотя Microsoft настоятельно не рекомендует так поступать, порой принудительная сборка мусора имеет смысл.
- ❑ **Windows сообщает о нехватке памяти.** Для общего мониторинга системной памяти CLR использует Win32-функции `CreateMemoryResourceNotification` и `QueryMemoryResourceNotification`. При сообщении Windows о нехватке памяти CLR инициирует сборку мусора, чтобы освободить нерабочие объекты и уменьшить рабочий набор процесса.
- ❑ **Выгрузка среды CLR домена приложения.** Выгружая домен приложения, CLR считает, что ни один объект в нем не является корнем, и выполняет сборку мусора всех поколений (о доменах приложения рассказывается в главе 22).
- ❑ **Закрытие CLR.** CLR заканчивает работу после нормального окончания процесса (в отличие от принудительного окончания, например, через диспетчер задач). При этом CLR считает, что в процессе нет корней, и вызывает метод финализации для всех объектов в управляемой куче. Учтите, что CLR не пытается сжать или освободить память, поскольку процесс заканчивается, и Windows освобождает всю занятую им память.

CLR использует особый выделенный поток для вызова методов финализации. В случае первых четырех событий, если метод финализации заикливается, выделенный поток блокируется, и вызов методов финализации прекращается. Это очень плохо, потому что приложение не может освободить память, занятую объектами, у которых есть метод финализации, и в итоге начинает испытывать нехватку памяти.

В случае пятого события каждому методу финализации дается примерно две секунды на то, чтобы вернуть управление программе. Если он не успевает, CLR просто прерывает процесс, и методы финализации больше не вызываются. А если для вызова методов финализации всех объектов требуется более 40 секунд, CLR также просто прерывает процесс.

ПРИМЕЧАНИЕ

Указанные значения тайм-аута были верны на момент написания этой книги, но ничто не мешает Microsoft изменить их в будущем. Код метода финализации может создавать новые объекты. Если это происходит при закрытии среды CLR, она продолжает уничтожать объекты и вызывать их методы финализации, пока не кончатся объекты или не пройдет 40 секунд.

Вспомните показанный ранее тип `GCBeep`. Если объект `GCBeep` завершится по первой, второй или третьей причине, будет создан новый объект `GCBeep`. Это нормально, поскольку приложение продолжает работать, предполагая, что в дальнейшем еще произойдет сборка мусора. Однако если объект `GCBeep` завершится по четвертой или пятой причине, новый объект `GCBeep` не должен созда-

ваться, так как это произойдет при выгрузке домена приложения или закрытии CLR. Если такие новые объекты все же будут созданы, CLR начнет выполнять бесполезную работу, продолжая вызывать методы финализации.

Чтобы предотвратить создание новых объектов GCBeep, метод финализации объекта GCBeep вызывает метод `IsFinalizingForUnload` объекта `AppDomain` и запрашивает свойство `HasShutdownStarted` объекта `System.Environment`. Метод `IsFinalizingForUnload` возвращает значение `true`, если метод финализации объекта вызван в процессе выгрузки домена приложения. Свойство `HasShutdownStarted` возвращает значение `true`, если метод финализации объекта вызван из-за закрытия процесса.

Детали механизма финализации

На первый взгляд, в финализации нет ничего особенного. Вы создаете объект, а когда его подбирает сборщик мусора, вызывается метод финализации этого объекта. Но на самом деле все гораздо сложнее.

Когда приложение создает новый объект, оператор `new` выделяет для него память из кучи. Если в типе объекта определен метод финализации, непосредственно перед вызовом конструктора экземпляра типа указатель на объект помещается в *список финализации* (finalization list) — внутреннюю структуру данных, управляемую сборщиком мусора. Каждая запись этого списка указывает на объект, для которого нужно вызвать метод финализации, прежде чем освободить занятую им память.

На рис. 21.5 показана куча с несколькими объектами. Одни достижимы из корней приложения, другие — нет. При создании объектов *C*, *E*, *F*, *I* и *J* система,

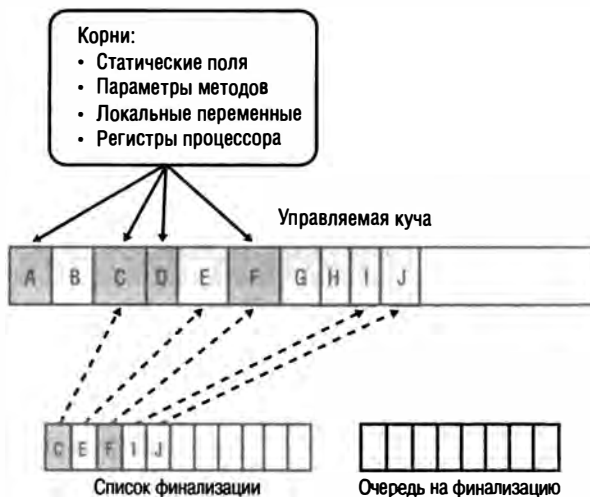


Рис. 21.5. Управляемая куча с указателями в списке финализации

обнаружив в их типах методы финализации и добавила указатели на эти объекты в список финализации.

ПРИМЕЧАНИЕ

Хотя в `System.Object` определен метод финализации, CLR его игнорирует. То есть если при создании экземпляра типа метод финализации этого типа унаследован от `System.Object`, созданный объект не считается подлежащим финализации. Метод финализации объекта `Object` должен переопределяться в одном из производных типов.

Сначала сборщик мусора определяет, что объекты *B*, *E*, *G*, *G*, *I* и *J* — это мусор. Сборщик сканирует список финализации в поисках указателей на эти объекты. Обнаружив указатель, он извлекает его из списка финализации и добавляет в конец *очереди на финализацию* (*freachable queue*) — еще одной внутренней структуры данных сборщика мусора. Каждый указатель в этой очереди идентифицирует объект, готовый к вызову своего метода финализации. Вид управляемой кучи после сборки мусора показан на рис. 21.6.

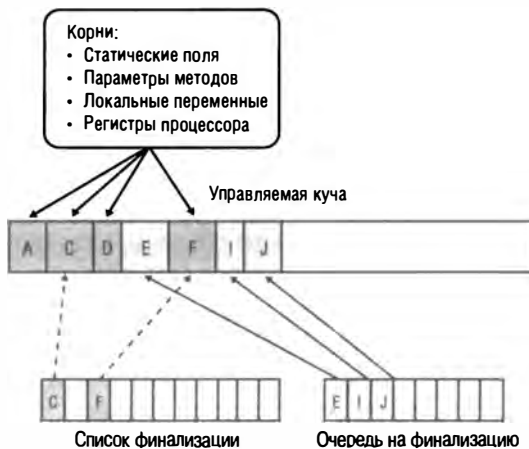


Рис. 21.6. Управляемая куча с указателями, перемещенными из списка финализации в очередь на финализацию

На рисунке видно, что занятая объектами *B*, *G* и *H* память была освобождена, поскольку у них нет метода финализации. Однако память, занятую объектами *E*, *I* и *J*, освободить нельзя, так как их методы финализации еще не вызывались.

В CLR есть особый высокоприоритетный поток, выделенный для вызова методов финализации. Он нужен для предотвращения возможных проблем синхронизации, которые могли бы возникнуть при использовании вместо него одного из потоков приложения с обычным приоритетом. При пустой очереди на финализацию (это ее обычное состояние) данный поток бездействует.

Но как только в ней появляются элементы, он активизируется и последовательно удаляет элементы из очереди, вызывая соответствующие методы финализации. Особенности работы данного потока запрещают исполнять в методе финализации любой код, имеющий какие-либо допущения о потоке, исполняющем код. Например, в методе финализации следует избегать обращения к локальной памяти потока.

Возможно, в будущем, CLR будет поддерживать несколько потоков финализации. Поэтому следует избегать создания кода, в котором методы финализации вызываются последовательно. Иначе говоря, если код в методе финализации затрагивает общее состояние, нужно прибегнуть к запикиванию в рамках синхронизации потоков. При наличии всего лишь одного потока финализации могут возникнуть проблемы производительности и масштабируемости в ситуации, когда финализируемые объекты распределяются между несколькими процессорами, но лишь один поток исполняет методы финализации — он может просто не успеть.

Взаимодействие списка финализации и очереди на финализацию само по себе замечательно, но сначала я расскажу, почему эта очередь получила свое оригинальное название. Очевидно, буква «f» означает «finalization», то есть «финализация»: каждая запись в очереди — это ссылка на объект в управляемой куче, для которого должен быть вызван метод финализации. Вторая часть оригинального имени, «reachable», означает, что эти объекты доступны. То есть ее можно было бы назвать очередью ссылок на объекты, доступные для финализации, но для краткости мы будем называть ее очередью на финализацию. Эту очередь можно рассматривать и просто как корень, подобно статическим полям, которые являются корнями. Таким образом, находящийся в очереди на финализацию объект доступен и *не* является мусором.

Короче говоря, если объект недоступен, сборщик считает его мусором. Далее, когда сборщик перемещает ссылку на объект из списка финализации в очередь на финализацию, объект перестает считаться мусором, а это означает, что занятую им память освобождать нельзя. По мере маркировки объектов из очереди другие объекты, на которые ссылаются их поля ссылочного типа, также рекурсивно помечаются — все эти объекты должны пережить сборку мусора. На этом этапе сборщик завершает поиск мусора, и некоторые объекты, идентифицированные как мусор, перестают считаться таковым — они как бы воскрешаются. Сборщик мусора сжимает освобожденную память, а особый поток CLR очищает очередь на финализацию, выполняя метод финализации для каждого объекта из очереди.

Вызванный снова, сборщик обнаруживает, что финализированные объекты стали мусором, так как ни корни приложения, ни очередь на финализацию больше на них не указывают. Память, занятая этими объектами, попросту освобождается. Важно понять, что для освобождения памяти, занятой объектами, требующими финализации, сборку мусора нужно выполнить дважды. На самом деле может понадобиться и больше операций сборки мусора, поскольку объек-

ты переходят в следующее поколение (но об этом — чуть позже). На рис. 21.7 показан вид управляемой кучи после второй сборки мусора.

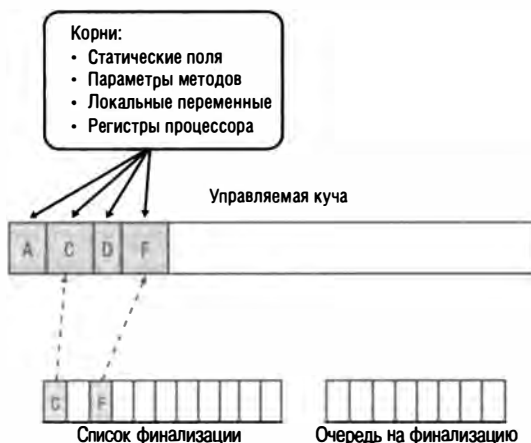


Рис. 21.7. Состояние управляемой кучи после второй сборки мусора

Эталон освобождения ресурсов: принудительная очистка объекта

Метод финализации чрезвычайно полезен, так как предотвращает утечку машинных ресурсов при освобождении памяти, занятой управляемыми объектами. Однако с ним есть проблемы: нельзя гарантировать его вызов в определенное время, а поскольку он не является открытым методом, пользователь класса не может вызвать его явно.

Возможность детерминированного уничтожения или закрытия объекта часто полезна при работе с неуправляемыми типами, которые играют роль оболочки машинных ресурсов, таких как файлы, соединения с базой данных или битовые карты. Например, пусть нужно открыть соединение с базой данных, извлечь из нее ряд записей и закрыть соединение. Нежелательно оставлять соединение открытым до следующей сборки мусора, особенно потому, что это событие может произойти через несколько часов, а то и дней после извлечения записей из базы данных.

В типах, поддерживающих возможность детерминированного уничтожения или закрытия, реализован *эталон освобождения ресурсов* (dispose pattern). В нем перечислены соглашения, которым должен следовать разработчик, определяющий тип, поддерживающий явную очистку. Кроме того, если тип поддерживает эталон освобождения ресурсов, разработчик, использующий тип, будет точно знать, как напрямую уничтожить объект, ставший ненужным.

ПРИМЕЧАНИЕ

Каждый тип, в котором определен метод финализации, должен также поддерживать эталон освобождения ресурсов, описываемый в этом разделе, чтобы у пользователей типа было больше возможностей управлять временем жизни ресурса. Однако существуют типы, которые поддерживают только эталон освобождения ресурсов, а метод финализации у них отсутствует. В эту категорию попадает, к примеру, класс `System.IO.BinaryWriter`. О причинах подобного несоответствия мы поговорим чуть позже в разделе «Интересные аспекты зависимостей».

Ранее был рассмотрен класс `SafeHandle`. Он реализует метод финализации, гарантирующий, что машинный ресурс, оболочкой которого является некий объект, закрывается (или освобождается) при удалении этого объекта сборщиком мусора. Однако разработчик, использующий объект `SafeHandle`, может напрямую закрыть машинный ресурс, потому что класс `SafeHandle` реализует интерфейс `IDisposable`.

Рассмотрим еще раз класс `SafeHandle`, но для краткости сосредоточимся на его фрагментах, связанных с эталоном освобождения ресурсов.

```
// Реализация интерфейса IDisposable сигнализирует пользователям
// этого класса о том, что он поддерживает эталон освобождения ресурсов
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable {

    // Этот открытый метод можно вызвать, чтобы наверняка
    // уничтожить ресурс. Он реализует метод Dispose интерфейса IDisposable
    public void Dispose() {
        // Вызов метода, реально выполняющего очистку
        Dispose(true);
    }

    // Этот открытый метод можно вызвать вместо Dispose
    public void Close() {
        Dispose(true);
    }

    // При сборке мусора этот метод финализации вызывается,
    // чтобы закрыть ресурс
    ~SafeHandle() {
        // Вызов метода, реально выполняющего очистку
        Dispose(false);
    }

    // Этот общий метод реально выполняет очистку
    // Его вызывают метод финализации, а также методы Dispose и Close
    // Поскольку этот класс не изолированный,
    // метод является защищенным и виртуальным
```

```
// Если бы этот класс был изолированным, метод был бы закрытым
protected virtual void Dispose(Boolean disposing) {
    if (disposing) {
        // Объект явно уничтожается или закрывается, но не финализируется
        // Поэтому в этой условной инструкции обращение к полям,
        // ссылающимся на другие объекты, безопасно для кода,
        // так как метод финализации этих объектов еще не вызывался

        // Классу SafeHandle здесь делать ничего не нужно
    }

    // Выполняется уничтожение/закрытие или финализация объекта
    // А происходит вот что:
    // Если ресурс уже освобожден, просто возвращается управление
    // Если значение ownsHandle равно false, возвращается управление
    // Устанавливается флаг, указывающий, что данный ресурс был освобожден
    // Вызов виртуального метода ReleaseHandle
    // Вызов GC.SuppressFinalize(this), запрещающий вызов метода финализации
}
}
```

Реализацию эталона освобождения ресурсов нельзя назвать тривиальной. Поэтому я объясню, что делает весь этот код. Во-первых, в классе `SafeHandle` реализован интерфейс `System.IDisposable`, определенный в FCL так:

```
public interface IDisposable {
    void Dispose();
}
```

Тип, в котором реализован этот интерфейс, «заявляет», что поддерживает эталон освобождения ресурсов. Проще говоря, этот тип поддерживает открытый метод `Dispose` без параметров, который можно явно вызвать для освобождения ресурса, оболочкой которого является объект. Учтите, что память, занятая самим объектом в управляемой куче, при этом не освобождается. Сборщик мусора по-прежнему отвечает за освобождение памяти объекта, и нельзя сказать наверняка, когда он это сделает. Оба не имеющие параметров метода `Dispose` и `Close` должны быть открытыми и неvirtуальными.

ПРИМЕЧАНИЕ

Возможно, вы заметили, что класс `SafeHandle` также поддерживает открытый метод `Close`, просто вызывающий метод `Dispose`. Для удобства некоторые классы, поддерживающие эталон освобождения ресурсов, заодно поддерживают и метод `Close`, но для эталона освобождения он не обязателен. Скажем, класс `System.IO.FileStream` поддерживает как эталон освобождения ресурсов, так и метод `Close`. Программистам кажется более естественным закрывать (`close`), а не уничтожать (`dispose`) файлы. Но у класса `System.Threading.Timer` нет метода `Close`, хотя он поддерживает эталон освобождения ресурсов.

ВНИМАНИЕ

Если в классе определено поле, тип которого реализует эталон освобождения ресурсов, данный эталон также должен быть реализован в этом классе. Метод `Dispose` должен уничтожать объекты, на которые ссылается это поле. Это позволяет при использовании этого класса вызывать для него метод `Dispose`, который, в свою очередь, освобождает ресурсы, занятые самим объектом. На самом деле, это одна из главных причин реализации в типах эталона освобождения ресурсов, а не метода финализации.

Например, эталон освобождения ресурсов реализован в классе `BinaryWriter`. При вызове `Dispose` для объекта `BinaryWriter` он (метод `Dispose`) вызывает метод `Dispose` для потокового объекта, хранимого как поле в объекте `BinaryWriter`. Поэтому при удалении объекта `BinaryWriter` нижележащий поток удаляется, что, в свою очередь, освобождает ресурс машинного потока.

Итак, вы знаете три способа очистки объекта `SafeHandle`: методом `Dispose`, методом `Close` или же с помощью сборщика мусора, вызывающего метод финализации объекта. Код очистки помещается в отдельный защищенный виртуальный метод, который также называют `Dispose`, но он принимает логический параметр `disposing`.

В этот метод помещают весь код, выполняющий очистку. В примере кода `SafeHandle` этот метод устанавливает флаг, означающий, что ресурс был освобожден, а затем вызывает виртуальный метод `ReleaseHandle`, который и освобождает ресурс. Учтите, что эталон освобождения ресурсов предполагает, что для одного объекта методы `Dispose` или `Close` могут вызываться по несколько раз; при первом вызове выполняется освобождение ресурса, а при последующих метод просто возвращает управление (без вбрасывания исключений).

ПРИМЕЧАНИЕ

Возможна ситуация, когда несколько потоков одновременно вызывают методы `Dispose/Close` для одного объекта. Однако эталон освобождения ресурсов не требует синхронизации потоков, потому что код вызывает `Dispose/Close`, только будучи уверенным, что этот объект в данный момент не используется никаким другим потоком. Если нет уверенности в том, что в данном месте кода объект никем не используется, вызывать методы `Dispose/Close` не следует. Лучше подождать, пока сборщик мусора не определит, что объект больше не нужен, а затем освободить ресурс.

Во время вызова метода финализации параметр `disposing` метода `Dispose` получает значение `false`. Это запрещает методу `Dispose` исполнять любой код, ссылающийся на другие управляемые объекты, в классах которых реализован метод финализации. Представьте, что во время завершения работы CLR вы пытаетесь выполнить запись в объект `FileStream` в методе финализации. Неизвестно, будет ли это работать, так как у `FileStream` может уже быть

вызван метод финализации, который закроет соответствующий дисковый файл.

В то же время при вызове метода `Dispose` или `Close` параметр `disposing` метода `Dispose` должен приравниваться к `true`. Это показывает, что объект был закрыт, а не финализирован. В этом случае методу `Dispose` разрешается исполнять код, ссылающийся на другой объект (например, на `FileStream`). Поскольку мы сами определяем логику, нам известно, что объект `FileStream` все еще открыт.

Кстати, если бы класс `SafeHandle` был изолированным, метод `Dispose`, принимающий логический параметр, был бы реализован как закрытый, а не как защищенный виртуальный метод. Но поскольку класс `SafeHandle` не изолирован, любой производный от него класс может переопределить логический метод `Dispose`, чтобы изменить код очистки. В производном классе не будет реализован метод `Dispose` или `Close` без параметров, и он не переопределит метод финализации. Этот производный класс просто унаследует реализацию всех этих методов. Учтите, что переопределенный в производном классе метод `Dispose`, принимающий логический параметр, должен вызывать логический метод `Dispose` базового класса, что позволяет базовому классу выполнить всю необходимую очистку. То же можно сказать о типе `FileStream`, использованном в примере, — он является производным от типа `Stream`, в котором реализованы методы `Close` и `IDisposable.Dispose` без параметров. Тип `FileStream` просто переопределяет метод `Dispose`, принимающий логический параметр, чтобы очистить поле `SafeHandle`, являющееся оболочкой для неуправляемого файлового ресурса.

ВНИМАНИЕ

Следует помнить о проблеме управления версиями. Если в версии 1 базового типа не реализован интерфейс `IDisposable`, он также не будет реализован в следующих версиях. Если же интерфейс `IDisposable` будет добавлен к базовому типу в будущем, ни один из производных типов не узнает, как вызывать методы базового типа, а у базового типа не будет шансов корректно выполнить очистку. В то же время, если в версии 1 интерфейс `IDisposable` будет реализован в базовом типе, его не удастся убрать из последующих версий, потому что иначе производный тип попытается вызывать методы, более не существующие в базовом типе.

Обратите также внимание на ту часть кода, где внутри принимающего логическое значение метода `Dispose` вызывается статический метод `SuppressFinalize` типа `GC`. Если код, где используется объект `SafeHandle`, явно вызывает метод `Dispose` или `Close`, не нужно выполнять его метод финализации, потому что это не более чем бесполезная попытка повторно освободить ресурс. Вызов метода `SuppressFinalize` устанавливает битовый флаг, связанный с объектом, на который ссылается его единственный параметр `this`. При наличии этого флага CLR запрещено перемещать указатель на объект из списка финализации

в очередь на финализацию — это не дает вызвать метод финализации этого объекта и гарантирует, что объект не доживет до следующей сборки мусора. Учтите, что класс `SafeHandle` вызывает метод `SuppressFinalize`, даже когда объект находится в процессе финализации. Ничего плохого в этом нет — ведь объект уже финализируется.

Типы, реализующие эталон освобождения ресурсов

Итак, вы познакомились с реализацией эталона освобождения ресурсов в типах. Теперь посмотрим, как разработчики используют подобные типы. Оставим в стороне класс `SafeHandle` и поговорим о более распространенном классе `System.IO.FileStream`. Класс `FileStream` позволяет открыть файл, прочитать из него и записать в него байты, а затем закрыть его. При создании объекта `FileStream` вызывается Win32-функция `CreateFile`, возвращаемый дескриптор сохраняется в объекте `SafeFileHandle`, а ссылка на этот объект сохраняется как закрытое поле в объекте `FileStream`. Класс `FileStream` также поддерживает ряд дополнительных свойств (например, `Length`, `Position`, `CanRead`) и методов (`Read`, `Write`, `Flush`).

Допустим, нам требуется код, который создает временный файл, записывает в него байты, после чего удаляет файл. Для начала рассмотрим такой вариант:

```
using System;
using System.IO;

public static class Program {
    public static void Main() {
        // Создание байтов для записи во временный файл
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // Создание временного файла
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);

        // Запись байтов во временный файл
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // Удаление временного файла
        File.Delete("Temp.dat"); // Вбрасывается исключение IOException
    }
}
```

К сожалению, если скомпоновать и запустить этот код, работать он, скорее всего, не будет. Дело в том, что вызов статического метода `Delete` объекта

File заставляет Windows удалить открытый файл, поэтому метод Delete вбрасывает исключение `System.IO.IOException` с таким сообщением (процесс не может обратиться к файлу `Temp.dat`, потому что он используется другим процессом):

The process cannot access the file "Temp.dat" because it is being used by another process

В некоторых случаях файл все же удаляется! Если другой поток инициировал сборку мусора между вызовами методов `Write` и `Delete`, поле `SafeFileHandle` объекта `FileStream` вызывает свой метод финализации, который закрывает файл и разрешает выполняться методу `Delete`. Однако вероятность данной ситуации крайне мала, поэтому в 99 случаях из 100 приведенный код работать не будет.

К счастью, в классе `FileStream` реализован эталон освобождения ресурсов. И мы можем изменить исходный текст программы, заставив ее закрывать файл.

```
using System;
using System.IO;

public static class Program {
    public static void Main() {
        // Создание байтов для записи во временный файл
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // Создание временного файла
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);

        // Запись байтов во временный файл
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // Явное закрытие файла после записи
        fs.Dispose();

        // Удаление временного файла
        File.Delete("Temp.dat"); // Теперь эта инструкция
                                // всегда остается работоспособной
    }
}
```

Единственное отличие здесь в том, что я добавил вызов метода `Dispose` объекта `FileStream`. Метод `Dispose` вызывает метод `Dispose`, принимающий как параметр тип `Boolean`, который, в свою очередь, вызывает метод `Dispose` для объекта `SafeFileHandle`. Затем выполняется вызов Win32-функции `CloseHandle`, которая заставляет Windows закрыть файл. Теперь при вызове метода `Delete` объекта `File` Windows видит, что файл не открыт, и успешно удаляет его.

Поскольку класс `FileStream` поддерживает также открытое свойство `Close`, можно переписать предыдущий код, и он будет работать, как раньше:

```
using System;
using System.IO;

public static class Program {
    public static void Main() {

        // Создание байтов для записи во временный файл
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // Создание временного файла
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);

        // Запись байтов во временный файл
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // Явное закрытие файла после записи
        fs.Close();

        // Удаление временного файла
        File.Delete("Temp.dat"); // Теперь эта инструкция
                                // всегда остается работоспособной
    }
}
```

ПРИМЕЧАНИЕ

Не забывайте, что метод `Close` официально не входит в эталон освобождения ресурсов: одни типы поддерживают его, другие — нет.

Учтите, что методы `Dispose` и `Close` — это просто средство заставить объект выполнить самоочистку в определенное время. Эти методы не управляют памятью, занятой объектом в управляемой куче. Это значит, что методы объекта можно вызывать даже после его очистки. Следующий код вызывает метод `Write` после закрытия файла и пытается дописать в файл несколько байтов.

```
using System;
using System.IO;

public static class Program {
    public static void Main() {
        // Создание байтов для записи во временный файл
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // Создание временного файла
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);
```

```
// Запись байтов во временный файл
fs.Write(bytesToWrite, 0, bytesToWrite.Length);

// Явное закрытие файла после записи
fs.Close();

// Попытка записать данные в файл после его закрытия
// Следующая строка вбрасывает исключение ObjectDisposedException
fs.Write(bytesToWrite, 0, bytesToWrite.Length);

// Удаление временного файла
File.Delete("Temp.dat");
}
}
```

Ясно, что попытка записи в закрытый файл не удастся, поэтому при исполнении кода второй вызов метода `Write` вбросит исключение `System.ObjectDisposedException` со следующим сообщением (нет доступа к закрытому файлу):

Cannot access a closed file

В данном случае память не «портится», так как область, выделенная для объекта `FileStream`, все еще существует; просто после явного освобождения объект не может успешно выполнять свои методы.

ВНИМАНИЕ

Определяя собственный тип, реализующий эталон освобождения ресурсов, обязательно сделайте так, чтобы все методы и свойства в случае явной очистки объекта вбрасывали исключение `System.ObjectDisposedException`. При повторных вызовах методы `Dispose` и `Close` никогда не должны вбрасывать исключение `ObjectDisposedException` — они должны просто возвращать управление.

ВНИМАНИЕ

Настоятельно рекомендую в общем случае отказаться от применения методов `Dispose` или `Close`. Сборщик мусора из CLR достаточно хорошо написан, и пусть он делает свою работу сам. Он определяет, когда объект более недоступен коду приложения, и только тогда уничтожает его. Вызывая метод `Dispose` или `Close`, код приложения, в сущности, заявляет, что сам «знает», когда объект становится ненужным приложению. Но за частую приложение не может достоверно судить об этом.

Допустим, у вас есть код метода, создающего новый объект. Ссылка на новый объект передается другому методу, который сохраняет ее в переменной в некотором внутреннем поле (то есть в корне), но вызывающий

метод никогда об этом не узнает. Конечно же, он может вызывать метод `Dispose` или `Close`, но если какой-то код попытается обратиться к этому объекту, будет выброшено исключение `ObjectDisposedException`.

Рекомендую вызывать методы `Dispose` или `Close` только там, где можно точно сказать, что потребуется очистка ресурса (как в случае с попыткой удаления открытого файла), или там, где это заведомо безопасно и позволяет повысить быстродействие, убрав объект из списка финализации и тем самым не позволив ему перейти в следующее поколение.

Инструкция `using` языка C#

Приведенные примеры кода демонстрируют методику явного вызова методов `Dispose` и `Close`. Если вы решили воспользоваться явным вызовом, настоятельно рекомендую поместить его в блок обработки исключений `finally`, так как это гарантирует исполнение кода очистки. Соответственно, код из предыдущего примера лучше переписать так:

```
using System;
using System.IO;

public static class Program {
    public static void Main() {
        // Создание байтов для записи во временный файл
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // Создание временного файла
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);
        try {
            // Запись байтов во временный файл
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);
        }
        finally {
            // Явное закрытие файла после записи
            if (fs != null)
                fs.Dispose();
        }

        // Удаление временного файла
        File.Delete("Temp.dat");
    }
}
```

Здесь хорошо было бы добавить код для обработки исключений — не поленились это сделать. К счастью, в C# есть инструкция `using`, предлагающая

упрощенный синтаксис генерации кода, идентичного показанному. Вот как можно переписать предыдущий код с помощью этой инструкции:

```
using System;
using System.IO;

public static class Program {
    public static void Main() {
        // Создание байтов для записи во временный файл
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // Создание временного файла
        using (FileStream fs = new FileStream("Temp.dat", FileMode.Create)) {
            // Запись байтов во временный файл
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);
        }

        // Удаление временного файла
        File.Delete("Temp.dat");
    }
}
```

Инструкция `using` инициализирует объект и сохраняет в переменной ссылку на него. После этого к этой переменной можно обращаться из кода, расположенного в скобках в инструкции `using`. При компиляции этого кода автоматически создаются блоки `try` и `finally`. Внутри блока `finally` компилятор помещает код, выполняющий приведение типа объекта к интерфейсу `IDisposable`, и вызывает метод `Dispose`. Ясно, что компилятор позволяет использовать инструкцию `using` только с типами, в которых реализован интерфейс `IDisposable`.

ПРИМЕЧАНИЕ

Инструкция `using` языка C# позволяет инициализировать несколько переменных одного типа или использовать переменную, инициализированную ранее.

Инструкция `using` также работает со значимыми типами, реализующими интерфейс `IDisposable`. Это позволяет получить чрезвычайно эффективный и полезный механизм инкапсуляции кода, необходимого для начала и завершения операции. Предположим, нужно запереть блок кода с помощью мьютекса. В классе `Mutex` не реализован интерфейс `IDisposable`, но вызов метода `Dispose` для этого объекта освобождает машинный ресурс. Это не имеет никакого отношения к запираанию. Чтобы иметь упрощенный синтаксис для запираания и отпираания мьютекса, можно определить значимый тип, запирающий и отпирающий объект `Mutex`. Для примера рассмотрим структуру `MutexLock`.

Следующий за ней метод Main демонстрирует эффективное использование этой структуры.

```
using System;
using System.Threading;

// Значимый тип, инкапсулирующий записание и отпирание мьютекса
internal struct MutexLock : IDisposable {
    private readonly Mutex m_mutex;

    // Этот конструктор включает режим записания мьютекса
    public MutexLock(Mutex m) {
        m_mutex = m;
        m_mutex.WaitOne();
    }

    // Этот метод Dispose отключает режим записания мьютекса
    public void Dispose() {
        m_mutex.ReleaseMutex();
    }
}

public static class Program {
    // Этот метод демонстрирует эффективное использование MutexLock
    public static void Main() {
        // Создание мьютекса
        Mutex m = new Mutex();

        // Запираем мьютекс, делаем что-то и отпираем мьютекс
        using (new MutexLock(m)) {
            // Выполнение операций в безопасном в отношении потоков режиме
        }
    }
}
```

Интересные аспекты зависимостей

Тип `System.IO.FileStream` позволяет пользователю открывать файл для чтения и записи. Для повышения быстродействия реализация типа задействует буфер памяти. Тип сбрасывает содержимое буфера в файл только после его заполнения. Тип `FileStream` поддерживает только запись байтов — для записи символов или строк требуется тип `System.IO.StreamWriter`, как показано в следующем коде:

```
FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
```

```
sw.Write("Hi there");
```

```
// Следующий вызов метода Close обязателен
```

```
sw.Close();
```

```
// ПРИМЕЧАНИЕ. Метод StreamWriter.Close закрывает объект FileStream
```

```
// Вручную закрывать объект FileStream не нужно
```

Обратите внимание, что конструктор `StreamWriter` принимает в качестве параметра ссылку на объект `Stream`, тем самым позволяя передать как параметр ссылку на объект `FileStream`. Внутренний код объекта `StreamWriter` сохраняет ссылку на объект `Stream`. При записи в объект `StreamWriter` он выполняет внутреннюю буферизацию данных в свой буфер в памяти. После заполнения буфера `StreamWriter` записывает данные в `Stream`.

После записи данных через объект `BinaryWriter` следует вызвать метод `Dispose` или `Close` (так как в типе `StreamWriter` реализован эталон освобождения ресурсов, его можно использовать с инструкцией `using` языка C#). Оба эти метода делают одно и то же: заставляют `BinaryWriter` сбросить данные в объект `Stream` и закрыть его. В данном примере при закрытии объект `FileStream` сбрасывает свои данные на диск прямо перед вызовом Win32-функции `CloseHandle`.

ПРИМЕЧАНИЕ

Вручную вызывать метод `Dispose` или `Close` для объекта `FileStream` не обязательно: `BinaryWriter` сделает это сам. Если же один из этих методов все-таки вызван явно, `FileStream` обнаружит, что очистка объекта уже выполнена, и вызванный метод просто вернет управление.

Как вы думаете, что было бы, не будь кода, явно вызывающего метод `Dispose` или `Close`? Сборщик мусора однажды правильно определил бы, что эти объекты стали мусором, и финализировал их. Но он не может гарантировать определенной очередности вызова методов финализации. Поэтому если объект `FileStream` завершится первым, он закроет файл. Затем после финализации объекта `StreamWriter` он попытается записать данные в закрытый файл, что вызовет исключение. В то же время, если `StreamWriter` завершается первым, данные благополучно записываются в файл.

Как с этой проблемой справились в Microsoft? Заставить сборщик мусора финализировать объекты в определенном порядке нельзя, так как в объектах могут быть ссылки друг на друга, и тогда сборщик не сможет определить правильную очередность их финализации. В Microsoft нашли выход: в типе `StreamWriter` не реализован метод финализации, поэтому этот тип не может сбросить данные из своего буфера в базовый объект `FileStream`. Таким образом, если вы забыли вручную закрыть объект `StreamWriter`, данные гарантированно будут потеряны. В Microsoft считают, что разработчики не смогут не заметить этой повторяющейся потери данных и исправят код, вставив явный вызов `Close` или `Dispose`.

ПРИМЕЧАНИЕ

В .NET Framework поддерживаются управляемые отладчики (Managed Debugging Assistants, MDA). Когда они включены, .NET Framework выполняет поиск некоторых распространенных ошибок в программах и запускает соответствующий отладчик. В отладчике все это выглядит как вбрасывание исключения. MDA умеет определять ситуации, когда объект `StreamWriter` удален сборщиком мусора до своего закрытия. Чтобы включить данный управляемый отладчик в Visual Studio, откройте проект и выберите в меню команду `Debug ▶ Exceptions`. В диалоговом окне `Exceptions` раскройте узел `Managed Debugging Assistants`, прокрутите страницу вниз до `StreamWriterBufferedDataLost` и отметьте этот элемент флажком `Thrown`, чтобы заставить отладчик Visual Studio останавливаться при каждой потере данных объекта `StreamWriter`.

Мониторинг и контроль времени жизни объектов

Для каждого домена приложения CLR поддерживает *таблицу GC-дескрипторов* (*GC handle table*), с помощью которой приложение отслеживает время жизни объекта или позволяет управлять им вручную. В момент создания домена приложения таблица пуста. Каждый элемент таблицы состоит из указателя на объект в управляемой куче и флага, задающего способ мониторинга или контроля объекта. Приложение добавляет в таблицу и удаляет из таблицы элементы с помощью показанного далее типа `System.Runtime.InteropServices.GCHandle`. Поскольку таблица GC-дескрипторов чаще всего используется в сценариях взаимодействия с неуправляемым кодом, к большинству членов `GCHandle` должен быть применен атрибут `[SecurityCritical]`:

```
// Этот тип определен в пространстве имен System.Runtime.InteropServices
public struct GCHandle {
    // Статические методы, создающие элементы таблицы
    public static GCHandle Alloc(object value);
    public static GCHandle Alloc(object value, GCHandleType type);

    // Статические методы, преобразующие GCHandle в IntPtr
    public static explicit operator IntPtr(GCHandle value);
    public static IntPtr ToIntPtr(GCHandle value);

    // Статические методы, преобразующие IntPtr в GCHandle
    public static explicit operator GCHandle(IntPtr value);
    public static GCHandle FromIntPtr(IntPtr value);

    // Статические методы, сравнивающие два типа GHandles
```

```
public static Boolean operator ==(GCHandle a, GCHandle b);
public static Boolean operator !=(GCHandle a, GCHandle b);

// Экземплярный метод, освобождающий элемент таблицы (индекс равен 0)
public void Free();

// Экземплярное свойство, извлекающее/назначающее
// для элемента ссылку на объект
public object Target { get; set; }

// Экземплярное свойство, равное true при отличном от 0 индексе
public Boolean IsAllocated { get; }

// Для элементов с флагом pinned возвращается адрес объекта
public IntPtr AddrOfPinnedObject();

public override Int32 GetHashCode();
public override Boolean Equals(object o);
}
```

В сущности, для контроля или мониторинга времени жизни объекта вызывается статический метод `Alloc` объекта `GCHandle`, передающий ссылку на этот объект, и тип `GCHandleType`, представляющий собой флаг, задающий способ мониторинга/контроля объекта. Перечислимый тип `GCHandleType` определяется так:

```
public enum GCHandleType {
    Weak = 0, // Мониторинг существования объекта
    WeakTrackResurrection = 1 // Мониторинг существования объекта
    Normal = 2, // Управление временем жизни объекта
    Pinned = 3 // Управление временем жизни объекта
}
```

Вот что означают эти флаги.

- ❑ **Weak** — мониторинг времени жизни объекта. Флаг позволяет узнать, когда сборщик мусора обнаруживает, что объект более недоступен коду приложения. Учтите, что метод финализации объекта мог как выполниться, так и не выполниться, поэтому объект может по-прежнему оставаться в памяти.
- ❑ **WeakTrackResurrection** — мониторинг времени жизни объекта. Флаг позволяет узнать, когда сборщик мусора обнаруживает, что объект более недоступен коду приложения. Учтите, что метод финализации объекта (если таковой имеется) уже точно был выполнен, то есть память, занятая объектом, была освобождена.
- ❑ **Normal** — контроль времени жизни объекта. Флаг заставляет сборщик мусора оставить объект в памяти, даже если в приложении нет переменных

(корней), ссылающихся на него. В ходе сборки мусора память, занятая этим объектом, может быть сжата (перемещена). Метод `Alloc`, не принимающий флаг `GCHandleType`, предполагает, что тип `GCHandleType.Normal` определен.

- **Pinned** — контроль времени жизни объекта. Флаг заставляет сборщик мусора оставить объект в памяти, даже если в приложении нет переменных (корней), ссылающихся на него. В ходе сборки мусора память, занятая этим объектом, не может быть сжата (перемещена). Это обычно бывает полезно, когда нужно передать адрес памяти в неуправляемый код. Неуправляемый код может выполнять запись по этому адресу в управляемой куче, зная, что расположение управляемого объекта после сборки мусора не изменится.

При вызове статический метод `Alloc` объекта `GCHandle` сканирует таблицу GC-дескрипторов домена приложения в поисках элемента, в котором хранится адрес объекта, переданного ему в качестве параметра. При этом устанавливается флаг, переданный в качестве параметра типу `GCHandleType`. Затем метод `Alloc` возвращает экземпляр `GCHandle`. Тип `GCHandle` — это «облегченный» значимый тип, содержащий одно экземплярное поле `IntPtr`, ссылающееся на индекс элемента в таблице. Чтобы освободить этот элемент в таблице GC-дескрипторов, нужно взять экземпляр `GCHandle` и вызвать метод `Free` (который также объявляет недействительным экземпляр, приравнивая значение поля `IntPtr` к нулю).

Вот как сборщик мусора работает с таблицей GC-дескрипторов.

1. Сборщик мусора маркирует все доступные объекты (как описано в начале этой главы). Затем он сканирует таблицу GC-дескрипторов, все объекты с флагом `Normal` или `Pinned` считаются корнями и также маркируются (в том числе все объекты, на которые они ссылаются через свои поля).
2. Сборщик мусора сканирует таблицу GC-дескрипторов в поисках всех записей с флагом `Weak`. Если такая запись ссылается на немаркированный объект, указатель относится к недоступному объекту (мусору) и приравнивается к `null`.
3. Сборщик мусора сканирует список финализации. Если указатель из списка ссылается на немаркированный объект, этот объект начинает считаться недоступным и перемещается из списка финализации в очередь на финализацию. В этот момент объект маркируется, потому что начинает считаться доступным.
4. Сборщик мусора сканирует таблицу GC-дескрипторов в поисках всех элементов с флагом `WeakTrackResurrection`. Если такой элемент ссылается на немаркированный объект (теперь это объект, на который указывает элемент из очереди на финализацию), указатель считается относящимся к недоступному объекту (мусором) и приравнивается к `null`.
5. Сборщик мусора сжимает память, убирая свободные места, оставшиеся на месте недоступных объектов. Учтите, что сборщик иногда предпочитает не

сжимать память, если посчитает, что фрагментация низка и на ее устранение не стоит тратить время. Объекты с флагом `Pinned` не сжимаются (не перемещаются), а объекты, находящиеся рядом, могут перемещаться.

Разобравшись в логике работы сборщика, попробуем использовать эти знания на практике. Наиболее понятны флаги `Normal` и `Pinned`, поэтому начнем с них. Обычно они применяются при взаимодействии с неуправляемым кодом.

Флаг `Normal` ставится, когда нужно передать ссылку на управляемый объект неуправляемому коду при условии, что позже неуправляемый код выполнит обратный вызов управляемого кода, передав ему эту ссылку. В общем случае невозможно передать ссылку на управляемый объект неуправляемому коду, потому что при сборке мусора адрес объекта в памяти может измениться, что сделает указатель недействительным. Чтобы решить эту проблему, можно вызвать метод `Alloc` объекта `GCHandle` и передать ему ссылку на объект и флаг `Normal`. Затем возвращенный экземпляр `GCHandle` нужно привести к типу `IntPtr` и передать полученный результат в неуправляемый код. Когда неуправляемый код выполнит обратный вызов управляемого кода, последний приведет передаваемый тип `IntPtr` обратно к `GCHandle`, после чего запросит свойство `Target`, чтобы получить ссылку на управляемый объект (или его текущий адрес). Когда неуправляемому коду эта ссылка будет более не нужна, следует вызывать метод `Free` объекта `GCHandle`, который позволит очистить объект при следующей сборке мусора (при условии, что для этого объекта нет других корней).

Обратите внимание, что в этой ситуации неуправляемый код не работает с управляемым объектом как таковым, а лишь использует возможность сослаться на него. Однако бывают ситуации, когда неуправляемому коду требуется управляемый объект. Тогда этот объект следует отметить флагом `Pinned` — это запретит сборщику мусора перемещать и сжимать его. В качестве распространенного примера можно упомянуть передачу управляемого объекта `String` в `Win32`-функцию. При этом объект `String` надо обязательно отметить флагом `Pinned`, потому что нельзя передать ссылку на управляемый объект неуправляемому коду из-за возможного перемещения этого объекта сборщиком мусора. В противном случае, если бы объект `String` был перемещен, неуправляемый код выполнял бы чтение из памяти или запись в память, более не содержащую символов объекта `String`, и работа приложения стала бы непредсказуемой.

При использовании для вызова метода механизма `P/Invoke CLR` автоматически устанавливает для аргументов флаг `Pinned` и сбрасывает его, когда неуправляемый метод возвращает управление. Поэтому чаще всего в типе `GCHandle` не приходится самостоятельно явно устанавливать флаг `Pinned` для каких-либо управляемых объектов. Тип `GCHandle` нужно явно использовать для передачи адреса управляемого объекта неуправляемому коду. Затем неуправляемая функция возвращает управление, а неуправляемому коду этот объект

может потребоваться позднее. Чаще всего такая ситуация возникает при асинхронных операциях ввода-вывода.

Допустим, вы выделяете память для байтового массива, который должен заполняться данными по мере их поступления из сокета. Затем вызывается метод `Alloc` типа `GCHandle`, передающий ссылку на массив и флаг `Pinned`. Далее при помощи возвращенного экземпляра `GCHandle` вызывается метод `AddrOfPinnedObject`. Он возвращает `IntPtr` — действительный адрес объекта с флагом `Pinned` в управляемой куче. Затем этот адрес передается неуправляемой функции, которая сразу передает управление управляемому коду. В процессе поступления данных из сокета буфер байтового массива не должен перемещаться в памяти, что и обеспечивается флагом `Pinned`. По завершении операции асинхронного ввода-вывода вызывается метод `Free` объекта `GCHandle`, который разрешает перемещения в буфер при следующей сборке мусора. В управляемом коде по-прежнему должна присутствовать ссылка на этот буфер, обеспечивая разработчику доступ к данным. Эта ссылка не позволит сборщику полностью убрать буфер из памяти.

Следует упомянуть и о таком средстве фиксации объектов внутри кода, как инструкция `fixed`. Вот пример ее применения:

```
unsafe public static void Go() {
    // Выделение места под объекты, которые немедленно превращаются в мусор
    for (Int32 x = 0; x < 10000; x++) new Object();

    IntPtr originalMemoryAddress;
    Byte[] bytes = new Byte[1000]; // Располагаем этот массив
                                   // после мусорных объектов

    // Получаем адрес в памяти массива Byte[]
    fixed (Byte* pbytes = bytes) { originalMemoryAddress = (IntPtr) pbytes; }

    // Принудительная сборка мусора
    // Мусор исчезает, позволяя сжать массив Byte[]
    GC.Collect();

    // Повторное получение адреса массива Byte[] в памяти
    // и сравнение двух адресов
    fixed (Byte* pbytes = bytes) {
        Console.WriteLine("The Byte[] did{0} move during the GC".
            (originalMemoryAddress == (IntPtr) pbytes) ? " not" : null);
    }
}
```

Инструкция `fixed` языка `C#` работает эффективней, чем выделение в памяти фиксированного `GC`-дескриптора. В данном случае она заставляет установить специальный «блокирующий» флаг на локальную переменную `pbytes`. Сборщик мусора, исследуя содержимое этого корня и обнаруживая отличные

`null` значения, понимает, что во время сжатия перемещать объект, на который ссылается эта переменная, нельзя. Компилятор C# создает IL-код, присваивающий локальной переменной `pbytes` адрес объекта из начала блока `fixed`. При достижении конца блока компилятор создает IL-инструкцию, возвращающую переменной `pbytes` значение `null`. Она перестает ссылаться на объект, позволяя удалить этот объект в ходе следующей сборки мусора.

Флаги `Weak` и `WeakTrackResurrection` могут применяться как в сценариях взаимодействия с неуправляемым кодом, так и при использовании только управляемого кода. Флаг `Weak` указывает, что объект уже помечен как мусор, но занимаемая им память пока может оказаться не востребоваваемой. А вот флаг `WeakTrackResurrection` указывает на необходимость возвращения памяти. В то время как флаг `Weak` применяется повсеместно, я еще ни разу не видел применения флага `WeakTrackResurrection` в реальных приложениях.

Предположим, что объект `A` периодически вызывает метод для объекта `B`. Но наличие ссылки на объект `B` со стороны объекта `A` защищает его от сборщика мусора, и вполне возможны сценарии, в которых такое поведение нежелательно. Предположим, что вместо этого нам нужно, чтобы объект `A` вызывал метод объекта `B` при условии, что последний находится в управляемой куче. Для решения этой задачи объекту `A` следует вызвать метод `Alloc` класса `GCHandle`, передав этому методу ссылку на объект `B` и флаг `Weak`. В результате в объекте `A` будет храниться возвращенная ссылка на экземпляр `GCHandle`, а не реальная ссылка на объект `B`.

При отсутствии других корней, сохраняющих объект `B`, теперь он может отправляться в мусорную корзину. Если объекту `A` понадобится вызвать метод объекта `B`, он обратится к предназначенному только для чтения свойству `Target` класса `GCHandle`. Возвращение этим свойством значения, отличного от `null`, указывает, что объект `B` еще существует. После этого код объекта `A` сможет привести возвращенную ссылку к типу объекта `B` и вызвать метод. Если же свойство `Target` возвращает значение `null`, значит, объект `B` уничтожен сборщиком мусора, и объект `A` не будет пытаться вызвать метод объекта `B`. Впрочем, код объекта `A` может вызвать метод `Free` типа `GCHandle`, чтобы разорвать связь с экземпляром `GCHandle`.

Поскольку из-за высоких требований к безопасности при фиксации или сохранении объекта в памяти работать с типом `GCHandle` не просто, в пространство имен `System` был включен вспомогательный класс `WeakReference`. Это не более чем объектно-ориентированная оболочка экземпляра `GCHandle`: конструктор этого класса вызывает метод `Alloc` класса `GCHandle`, его свойство `Target` — одноименное свойство класса `GCHandle`, а его метод финализации — метод `Free` класса `GCHandle`. Для работы с классом `WeakReference` коду не требуется специального разрешения, так как этот класс поддерживает только слабые ссылки; поведение, вызываемое экземплярами `GCHandle`, размещенными в типе `GCHandleType` с флагом `Normal` или `Pinned`, не поддерживается.

Недостатком класса `WeakReference` является необходимость пребывания объекта в куче. Из-за этого объекты `WeakReference` «весят» больше экземпляров `GCHandle`. Кроме того, класс `WeakReference` не реализует эталон освобождения ресурсов (что является ошибкой), соответственно, вы не сможете вручную разблокировать запись в таблице `GCHandle`. Придется ждать, пока не отработает сборщик мусора и не вызовет метод финализации. Класс `WeakReference` появился в версии `.NET Framework 1.0`, а значит, не является обобщенным (обобщенные классы впервые появились в версии 2.0). Поэтому я создал облегченную структуру, которую иногда использую во время компиляции в качестве безопасной оболочки класса `WeakReference`.

```
internal struct WeakReference<T> : IDisposable where T : class {
    private WeakReference m_weakReference;

    public WeakReference(T target) {
        m_weakReference = new WeakReference(target);
    }
    public T Target { get { return (T)m_weakReference.Target; } }
    public void Dispose() { m_weakReference = null; }
}
```

Иногда разработчики спрашивают, существует ли способ создания слабого делегата, при котором один объект регистрирует делегат обратного вызова с событием другого объекта. При этом разработчики не хотят регистрировать событие и насильно продлевать время жизни объекта. Предположим, у нас есть класс `DoNotLiveJustForTheEvent`. Мы хотим создать его экземпляр и зарегистрировать метод обратного вызова для события `Click` объекта `Button`, но это событие не должно сохранять объект `DoNotLiveJustForTheEvent`. Этот объект должен убираться сборщиком мусора и не должен получать уведомления при следующем возникновении события `Click` объекта `Button`. Посмотрим, каким образом можно реализовать подобный код.

Для начала рассмотрим определение класса `DoNotLiveJustForTheEvent`:

```
internal sealed class DoNotLiveJustForTheEvent {
    public void Clicked(Object sender, EventArgs e) {
        MessageBox.Show("Test got notified of button click.");
    }
}
```

Этот код создает форму и две кнопки. Первая кнопка находится на форме слева, а вторая — справа. Для первой кнопки `Button` создается экземпляр класса `DoNotLiveJustForTheEvent`, а метод `Clicked` этого объекта регистрируется в качестве обработчика события `Click`. Для этого я использую класс `WeakEventHandler`, преобразующий делегат `EventHandler` в его слабую версию (далее будет показано, каким образом реализован этот класс). Для события `Click` второй кнопки `Button` я регистрирую обратный вызов, запускающий

сборщик мусора. Чтобы проверить, что все работает корректно, я щелкаю на этой кнопке. Затем кнопки добавляются в коллекцию элементов управления формы, они масштабируются в соответствии с размером клиентской области, и форма выводится на экран:

```
public static void Go() {
    var form = new Form() {
        Text = "Weak Delegate Test",
        FormBorderStyle = FormBorderStyle.FixedSingle
    };

    var btnTest = new Button() {
        Text = "Click me",
        Width = form.Width / 2
    };

    var btnGC = new Button() {
        Text = "Force GC",
        Left = btnTest.Width,
        Width = btnTest.Width
    };

    // WeakEventHandler превращает делегат EventHandler в его слабую версию
    btnTest.Click += new WeakEventHandler(
        new DoNotLiveJustForTheEvent().Clicked)
    { RemoveDelegateCode = eh => btnTest.Click -= eh };

    btnGC.Click += (
        sender, e) => { GC.Collect(); MessageBox.Show("GC complete."); };

    form.Controls.Add(btnTest);
    form.Controls.Add(btnGC);
    form.ClientSize = new Size(btnTest.Width * 2, btnTest.Height);
    form.ShowDialog();
}
```

Так как я не сохраняю ссылку на объект `DoNotLiveJustForEvent` в корневой переменной, при следующем проходе сборщика мусора этот объект будет считаться подлежащим утилизации. Но до этого момента можно щелкать на левой кнопке сколько угодно и наблюдать вызов метода `Clicked` объекта `DoNotLiveJustForEvent`. При этом щелчок на правой кнопке будет отправлять объект `DoNotLiveJustForEvent` в мусор. При последующем щелчке на левой кнопке обработчик `WeakEventHandler` обнаружит исчезновение объекта `DoNotLiveJustForEvent` и отменит свою регистрацию для события `Click` объекта `Button`. А значит, в дальнейшем он вызываться не будет. И конечно, во время следующей сборки мусора объект `WeakEventHandler` вернет занятую им память.

Чтобы понять, как работает мой класс `WeakEventHandler`, сначала нужно рассмотреть работу его базового класса. Класс `WeakEventHandler` является производным от моего абстрактного обобщенного класса `WeakDelegate`:

```
public abstract class WeakDelegate<TDelegate> where TDelegate
    : class /* MulticastDelegate */ {
    private WeakReference<TDelegate> m_weakDelegate;
    private Action<TDelegate> m_removeDelegateCode;

    public WeakDelegate(TDelegate @delegate) {
        var md = (MulticastDelegate)(Object)@delegate;
        if (md.Target == null)
            throw new ArgumentException(
                "There is no reason to make a WeakDelegate to a static method.");

        // Сохранение WeakReference в делегат
        m_weakDelegate = new WeakReference<TDelegate>(@delegate);
    }

    public Action<TDelegate> RemoveDelegateCode {
        set {
            // Сохранение делегата, ссылающегося на код, который умеет удалять
            // объект WeakDelegate при отправке в мусор неслабого делегата
            m_removeDelegateCode = value;
        }
    }

    protected TDelegate GetRealDelegate() {
        // Если реальный делегат еще не отправлен в мусор, возвращаем управление
        TDelegate realDelegate = m_weakDelegate.Target;
        if (realDelegate != null) return realDelegate;

        // Реальный делегат отправлен в мусор, и нам больше не требуется
        // ссылка WeakReference (ее тоже можно отправить в мусор)
        m_weakDelegate.Dispose();

        // Удаление делегата из цепочки (если пользователь указывает способ)
        if (m_removeDelegateCode != null) {
            m_removeDelegateCode(GetDelegate());
            m_removeDelegateCode = null; // Удаляем обработчик из очереди
        }
        return null; // Реальный делегат отправлен в мусор и не может вызываться
    }

    // Все производные классы должны возвращать делегату
    // закрытый метод типа, совпадающего с TDelegate
    public abstract TDelegate GetDelegate();
}
```

```
// Оператор неявного преобразования объекта
// WeakDelegate в реальный делегат
public static implicit operator TDelegate(
    WeakDelegate<TDelegate> @delegate) {
    return @delegate.GetDelegate();
}
}
```

Теперь можно рассмотреть мой класс WeakEventHandler:

```
// Класс, поддерживающий необобщенный делегат EventHandler
public sealed class WeakEventHandler : WeakDelegate<EventHandler> {
    public WeakEventHandler(EventHandler @delegate) : base(@delegate) { }

    /// <summary>
    // Возвращение ссылки на необобщенный делегат EventHandler </summary>
    public override EventHandler GetDelegate() { return Callback; }

    // Сигнатура этого закрытого метода должна совпадать
    // с сигнатурой нужного делегата
    private void Callback(Object sender, EventArgs e) {
        // Если метод не отправлен в мусор, вызываем его
        var eh = base.GetRealDelegate();
        if (eh != null) eh(sender, e);
    }
}
```

Решить проблему слабого делегата оказалось сложнее, чем я предполагал. В CLR и C# есть масса ограничений, которые пришлось обходить. Это сделало код намного более длинным и сложным. Вот список проблем, с которыми мне пришлось столкнуться.

- ❑ Хотелось, чтобы обобщенный аргумент TDelegate моего класса WeakDelegate мог принимать только делегаты, производные от класса System.MulticastDelegate. Но C# не позволяет ограничивать обобщенные аргументы классом MulticastDelegate или даже классом System.Delegate (который является базовым для MulticastDelegate). Поэтому мне пришлось ограничить TDelegate классом (то есть любым ссылочным типом).
- ❑ Хотелось, чтобы, обнаружив отправку в мусор делегата, на который он ссылается, объект WeakDelegate автоматически удалялся из своей цепочки делегатов. Но делегат не умеет определять, какой именно цепочке он принадлежит.
- ❑ Я думал также о передаче события (например, события Click объекта Button) конструктору WeakDelegate. Это поместило бы в класс WeakDelegate код, автоматически удаляющий из события объект WeakDelegate, но, к сожалению, создать переменную, которая ссылалась бы на событие, невозможно. Пользователь класса WeakDelegate может при желании присвоить свойству

RemoveDelegateCode какой-либо делегат, ссылающийся на код, который знает, как удалять делегаты из цепочки или из события; по крайней мере, в этот код я смогу передать для удаления делегат объекта WeakDelegate.

- ❑ CLR трактует различные типы делегатов по-разному, а значит, вы не можете осуществить приведение ссылки одного типа делегата к другому даже при наличии у них одинаковых сигнатур. Скажем, мне хотелось привести ссылку на делегат EventHandler к делегату EventHandler<EventArgs>, благо они обладают одинаковыми сигнатурами. Но CLR запрещает подобное приведение.

Команда разработчиков CLR из Microsoft осведомлена об этих ограничениях и ищет средства, позволяющие приравнивать делегаты с одинаковыми сигнатурами в следующей версии CLR. Из-за этого ограничения для каждого типа делегата должен быть определен свой класс.

Я уже демонстрировал вам тип WeakEventHandler, соответствующий делегату EventHandler. Есть у меня и тип WeakEventHandler<EventArgs>, соответствующий типу делегата EventHandler<EventArgs>:

```
// Частичный класс WeakDelegate обеспечивает поддержку
// обобщенного делегата EventHandler<TEventArgs>
public sealed class WeakEventHandler<TEventArgs> :
    WeakDelegate<EventHandler<TEventArgs>> where TEventArgs : EventArgs {

    public WeakEventHandler(EventHandler<TEventArgs> @delegate) :
        base(@delegate) { }

    /// <summary>
    /// Возвращает ссылку на обобщенный делегат
    /// EventHandler<typeparam name="TEventArgs"/>
    /// </summary>
    public override EventHandler<TEventArgs> GetDelegate() {
        return Callback;
    }

    private void Callback(Object sender, TEventArgs e) {
        // Если метод не отправлен в мусор, вызываем его
        var eh = base.GetRealDelegate();
        if (eh != null) eh(sender, e);
    }
}
```

Если бы среда CLR считала эквивалентными делегаты с одинаковой сигнатурой, я смог бы использовать мой тип WeakEventHandler<EventArgs> для необобщенного события EventHandler, полностью удалив свой класс WeakEventHandler.

Было бы здорово, если бы платформа .NET Framework поддерживала механизм слабых ссылок на делегаты, но пока ничего подобного нет. Тем не менее

подобная возможность уже обсуждается разработчиками CLR в Microsoft и вполне может появиться уже в следующей версии. В этом случае я смог бы легко обойти все ограничения, с которыми мне пришлось столкнуться, получив в итоге более простую и эффективную реализацию.

ВНИМАНИЕ

Познакомившись с мягкими ссылками, разработчики сразу же считают, что они хорошо подходят для сценариев кэширования. Порядок рассуждений примерно таков: «Хорошо бы создать много объектов, содержащих много данных, а затем создать для них мягкие ссылки. Когда программе понадобятся эти данные, она с помощью мягкой ссылки проверит, есть ли поблизости объект, содержащий эти данные, и обнаружив его рядом, воспользуется нужными данными. Это обеспечит высокую производительность». Однако после сборки мусора объекты, содержащие данные, будут уничтожены, и когда программе придется заново создавать данные, ее производительность упадет.

Недостаток такого подхода в том, что сборка мусора не выполняется при переполненной или почти переполненной памяти. Напротив, она происходит, когда поколение 0 заполнено, что случается в какой-то момент после выделения очередных 256 байт памяти. Поэтому объекты удаляются из памяти гораздо чаще, чем нужно, что значительно снижает производительность приложения.

Мягкие ссылки могут быть очень эффективными при кэшировании, но очень сложно создать хороший алгоритм кэширования, обеспечивающий нужное равновесие между расходом памяти и быстродействием. В сущности, необходимо, чтобы в кэше были жесткие ссылки на все объекты, а затем, когда памяти становится мало, жесткие ссылки должны превращаться в мягкие. На сегодняшний момент CLR не поддерживает механизм, позволяющий уведомлять приложение об исчерпании ресурсов памяти. Тем не менее некоторые разработчики приспособились периодически вызывать Win32-функцию `GlobalMemoryStatusEx` и проверять член `dwMemoryLoad` возвращенной структуры `MEMORYSTATUSEX`. Его значение, превышающее 80, означает, что память на исходе и настало время преобразовывать жесткие ссылки в мягкие по выбранному алгоритму: по давности, частоте, времени использования объектов или по другим алгоритмам.

Разработчикам часто требуется связать фрагмент данных с каким-нибудь другим элементом. Например, можно связать данные с потоком или с доменом приложений. Класс `System.Runtime.CompilerServices.ConditionalWeakTable<TKey, TValue>` позволяет связать данные с объектом. Вот как он выглядит:

```
public sealed class ConditionalWeakTable<TKey, TValue>
where TKey : class where TValue : class {
    public ConditionalWeakTable();
```

продолжение ➤


```

public void Add(TKey key, TValue value);
public TValue GetValue(
    TKey key, CreateValueCallback<TKey, TValue> createValueCallback);
public Boolean TryGetValue(TKey key, out TValue value);
public TValue GetOrCreateValue(TKey key);
public Boolean Remove(TKey key);
public delegate TValue CreateValueCallback(TKey key); // Вложенное
// определение делегата
}

```

Для связи произвольных данных с одним или несколькими объектами класса для начала вам потребуется экземпляр этого класса. Затем следует вызвать метод `Add`, передав параметру `key` ссылку на объект, а параметру `value` — данные, которые вы хотите связать с этим объектом. При попытке повторно добавить ссылку на тот же самый объект метод `Add` вбросит исключение `ArgumentException`. Чтобы изменить связанное с объектом значение, нужно удалить ключ и добавить его снова уже с другим значением. Имейте в виду, что так как класс является безопасным в отношении потоков, его могут в конкурентном режиме использовать другие потоки, хотя это не лучшим образом сказывается на производительности. Производительность класса нужно проверять, чтобы узнать, насколько она достаточна именно для вашего сценария. И нет причин ограничивать тип `TValue` классом (только ссылочными типами). В будущем разработчики CLR могут убрать это ограничение для `TValue`, дав возможность, не прибегая к упаковке, связывать с объектами экземпляры значимых типов.

Разумеется, таблицы внутренне сохраняют ссылку `WeakReference` на объект, переданный им в качестве ключа; это гарантирует, что таблица не будет принудительно увеличивать время жизни объекта. Но особенность класса `ConditionalWeakTable` состоит в том, что он гарантирует наличие в памяти значения до тех пор, пока объект идентифицируется в памяти по ключу. Это превосходит способности обычного класса `WeakReference`, в котором значение уничтожается сборщиком мусора, хотя ключ еще существует. Класс `ConditionalWeakTable` может применяться для реализации свойства зависимостей в `Silverlight` и `Windows Presentation Foundation (WPF)`. Он может также внутренне использоваться в динамических языках для динамической связи данных с объектами.

Далее показан код, демонстрирующий применение класса `ConditionalWeakTable`. Он позволяет вызывать метод расширения `GCWatch` для любого объекта, передавая в него некий `ter String`. В результате при отправке объекта в мусор вы получаете извещение через консоль:

```

internal static class ConditionalWeakTableDemo {
    public static void Main() {
        Object o = new Object().GCWatch("My Object created at " + DateTime.Now);
        GC.Collect(); // Оповещение об отправке в мусор отсутствует
    }
}

```

```

GC.KeepAlive(o); // Убедитесь, что объект, на который
                  // ссылается o, существует
o = null;         // Объект, на который ссылается o, можно уничтожить
GC.Collect();      // Оповещение об отправке в мусор
}
}

internal static class GCWatcher {
    // ПРИМЕЧАНИЕ. Аккуратнее обращайтесь с типом Strings
    // из-за объектов-представителей MarshalByRefObject
    private readonly static ConditionalWeakTable<Object,
        NotifyWhenGCd<String>> s_cwt =
        new ConditionalWeakTable<Object, NotifyWhenGCd<String>>();

    private sealed class NotifyWhenGCd<T> {
        private readonly T m_value;

        internal NotifyWhenGCd(T value) { m_value = value; }
        public override string ToString() { return m_value.ToString(); }
        ~NotifyWhenGCd() { Console.WriteLine("GC'd: " + m_value); }
    }

    public static T GCWatch<T>(this T @object, String tag) where T : class {
        s_cwt.Add(@object, new NotifyWhenGCd<String>(tag));
        return @object;
    }
}

```

Воскрешение

Рассматривая процедуру финализации, мы говорили о том, что когда требующий финализации объект «умирает», сборщик мусора возвращает его к жизни, чтобы вызвать его метод финализации, и уже после вызова этого метода объект умирает навсегда. Подытоживая, можно сказать: объект, требующий финализации, сначала умирает, затем воскресает и умирает снова. Оживление мертвого объекта называется *воскрешением* (resurrection).

Акт подготовки к вызову метода финализации объекта — одна из форм воскрешения. Когда сборщик мусора помещает ссылку на объект в очередь на финализацию, объект становится доступным для корня и возвращается к жизни. Это нужно, чтобы код в методе финализации мог обратиться к полям объекта. В конце концов, метод финализации этого объекта возвращает управление, после чего на объект больше не указывает ни один корень, и далее объект, будучи выброшенным из очереди на финализацию, уничтожается навсегда.

Но что если метод финализации объекта исполняет код, который заносит указатель на объект в статическое поле:

```
internal sealed class SomeType {
    ~SomeType() {
        Program.s_ObjHolder = this;
    }
}

public static class Program {
    public static Object s_ObjHolder; // По умолчанию содержит null
    ...
}
```

В этом случае при вызове метода финализации объекта `SomeType` ссылка на него помещается в корень, в результате объект становится доступным для кода приложения. То есть теперь объект воскресает, а сборщик не принимает его за мусор. Приложение может снова задействовать этот объект, но помните, что он уже финализирован, поэтому его использование может дать непредсказуемые результаты. Также учтите, что если в `SomeType` есть поля, ссылающиеся на другие объекты, они тоже оказываются воскрешенными, так как они доступны для корней приложения. При этом для некоторых из этих объектов уже мог вызываться метод финализации.

Как и в реальной жизни (или смерти), в воскрешении нет ничего хорошего и при написании программ его лучше избегать. Воскрешение может быть полезно в тех немногих ситуациях, когда архитектура приложения требует многократного использования одного объекта. Когда объект больше не нужен, выполняется сборка мусора. В своем методе финализации объект присваивает своему указателю `this` другой корень, что запрещает сборщику удалять объект. Но надо сообщить сборщику мусора о необходимости вызова метода финализации после следующего использования объекта. Чтобы реализовать эту возможность, тип `GC` поддерживает статический метод `ReRegisterForFinalize`, принимающий один параметр — ссылку на объект. В следующем коде показано, как изменить метод финализации объекта `SomeType`, чтобы он вызывался после каждого использования объекта:

```
internal sealed class SomeType {
    ~SomeType() {
        Program.s_ObjHolder = this;
        GC.ReRegisterForFinalize(this);
    }
}
```

Вызов метода финализации воскрешает объект, заставляя корень вновь ссылаться на него. Далее этот метод вызывает метод `ReRegisterForFinalize`, который добавляет адрес заданного объекта (`this`) в конец списка финализации. Обнаружив, что этот объект более недоступен (когда статическое поле становится равным `null`), сборщик мусора переместит указатель на объект из

списка финализации в очередь на финализацию, после чего метод финализации будет вызван вновь. Опять же, помните, что при воскрешении объекта воскрешаются все объекты, на которые он ссылается, для всех этих объектов может потребоваться вызов `ReRegisterForFinalize`, но очень часто это бывает невозможно, потому что у разработчика нет доступа к закрытым полям этих объектов!

Этот пример демонстрирует создание объекта, который, постоянно воскрешая себя, становится «бессмертным», но такие объекты обычно нежелательны. Намного чаще в метод финализации добавляется корень, ссылающийся на объект, только после проверки определенного условия.

ПРИМЕЧАНИЕ

Следите, чтобы во время воскрешения метод `ReRegisterForFinalize` вызывался не больше одного раза, иначе метод финализации для объекта будет вызываться неоднократно, так как при каждом вызове `ReRegisterForFinalize` к списку финализации добавляется новый элемент. Когда объект определяется как мусор, все эти элементы перемещаются из списка финализации в очередь на финализацию, что приводит к многократному вызову метода финализации.

В качестве примера разумного применения воскрешения в конце этой главы показан класс `GCNotification`.

Поколения

Как отмечалось в начале этой главы, поколения — это механизм сборщика мусора в CLR, единственным назначением которого является повышение производительности приложения. *Сборщик мусора с поддержкой поколений* (generational garbage collector), который также называют *эфемерным сборщиком мусора* (ephemeral garbage collector), хотя я не использую такой термин в своей книге, работает на основе следующих предположений:

- ❑ чем младше объект, тем короче его время жизни;
- ❑ чем старше объект, тем длиннее его время жизни;
- ❑ сборку мусора в части кучи выполнить быстрее, чем во всей куче.

Справедливость этих предположений для большого набора существующих приложений доказана многочисленными исследованиями, поэтому они повлияли на реализацию сборщика мусора. В этом разделе описывается принцип работы поколений.

Сразу после инициализации в управляемой куче нет объектов. Говорят, что создаваемые в куче объекты составляют поколение 0. Проще говоря, к нулевому поколению относятся только что созданные объекты, которых не касался сбор-

щик мусора. Рисунок 21.8 демонстрирует только что запущенное приложение, разместившее в памяти пять объектов (A–E). Через некоторое время объекты C и E становятся недоступными.



Рис. 21.8. Вид кучи сразу после инициализации: все объекты в ней относятся к поколению 0, сборки мусора еще не было

При инициализации CLR выбирает пороговый размер для поколения 0, например 256 Кбайт (конкретный размер может варьироваться). Если в результате выделения памяти для нового объекта размер поколения 0 превышает пороговое значение, должна начаться сборка мусора. Допустим, объекты A–E занимают 256 Кбайт. Тогда при размещении объекта F должна начаться сборка мусора. Сборщик мусора выясняет, что объекты C и E — это мусор, и выполняет сжатие памяти для объекта D, перемещая его вплотную к объекту B. Между прочим, пороговый размер для поколения 0 в 256 Кбайт был выбран из-за того, что обычно все эти объекты целиком умещаются в кэш второго уровня (L2) процессора, что позволяет значительно повысить скорость дефрагментации памяти. Объекты, пережившие сборку мусора (A, B и D), становятся поколением 1. Объекты из поколения 1 были просмотрены сборщиком мусора один раз. Теперь куча выглядит так, как показано на рис. 21.9.



Рис. 21.9. Вид кучи после одной сборки мусора: выжившие объекты из поколения 0 переходят в поколение 1, поколение 0 пустует

После сборки мусора объектов в поколении 0 не остается. Туда помещаются новые объекты. Как показано на рис. 21.10, приложение продолжает работу и размещает объекты F–K. Также в ходе работы приложения становятся недоступными объекты B, H и J, поэтому занятая ими память должна рано или поздно освободиться.



Рис. 21.10. В поколении 0 появились новые объекты, в поколении 1 — мусор

А теперь представьте, что при попытке размещения объекта L размер поколения 0 превысил пороговое значение 256 Кбайт, поэтому должна начаться сборка мусора. При этом сборщик мусора решает, какие поколения следует обработать. Я уже отмечал, что при инициализации CLR выбирает пороговый размер поколения 0. CLR также выбирает пороговый размер для поколения 1, скажем 2 Мбайт.

Начиная сборку мусора, сборщик определяет, сколько памяти занято поколением 1. Пока поколение 1 занимает намного меньше 2 Мбайт, поэтому сборщик проверяет только объекты поколения 0. Первое допущение сборщика гласит, что у новых объектов время жизни короче. Поэтому в поколении 0, скорее всего, окажется много мусора, и очистка этого поколения освободит много памяти. А поскольку сборщик игнорирует объекты поколения 1, сборка мусора значительно ускоряется.

Ясно, что игнорирование объектов поколения 1 повышает быстродействие сборщика. Однако его производительность растет еще больше благодаря выборочной проверки объектов в управляемой куче. Если корень или объект ссылается на объект из старшего поколения, сборщик игнорирует все внутренние ссылки старшего объекта, сокращая время построения графа доступных объектов. Конечно, возможна ситуация, когда старый объект ссылается на новый. Чтобы не пропустить обновленные поля этих старых объектов, сборщик использует внутренний механизм JIT-компилятора, устанавливающий флаг при изменении ссылочного поля объекта. Он позволяет сборщику выяснить, какие из старых объектов (если они есть) были изменены с момента последней сборки мусора. Остается проверять только старые объекты с измененными полями, чтобы выяснить, не ссылаются ли они на новые объекты из поколения 0¹.

ПРИМЕЧАНИЕ

Тесты быстродействия, проведенные Microsoft, показали, что сборка мусора в поколении 0 занимает меньше 1 мс. Microsoft стремится к тому, чтобы сборка мусора занимала не больше времени, чем обслуживание обычной страничной ошибки.

¹ Когда JIT-компилятор создает машинный код, модифицирующий ссылочное поле внутри объекта, туда входит вызов барьерного метода записи (write barrier method). Этот метод проверяет, принадлежит ли объект, поля которого изменяются, к поколению 1 или 2. В случае положительного результата код барьерного метода записи задает бит во внутренней таблице (card table). Эта таблица содержит по одному биту для каждого 128-байтного диапазона данных в куче. В начале следующего цикла сборки мусора из таблицы определяется, поля каких объектов поколений 1 и 2 изменились с момента прошедшей сборки. Если какой-то из этих объектов ссылается на объект поколения 0, этот объект переживает сборку мусора. После завершения процедуры всем полям таблицы возвращаются нулевые значения. Наличие кода барьерного метода записи негативно сказывается на производительности при записи ссылочных полей у объекта (в отличие от локальных переменных или статических полей). Производительность падает еще больше, если объект принадлежит поколению 1 или 2.

Сборщик мусора с поддержкой поколений также предполагает, что объекты, прожившие достаточно долго, продолжают жить и дальше. Так что велика вероятность, что объекты поколения 1 и впредь останутся доступными в приложении. То есть проверив объекты поколения 1, сборщик нашел бы мало мусора и не смог бы освободить много памяти. Следовательно, сборка мусора в поколении 1, скорее всего, окажется пустой тратой времени. Если в поколении 1 появляется мусор, он просто остается там. Сейчас куча выглядит, как показано на рис. 21.11.

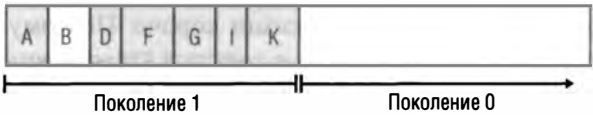


Рис. 21.11. Вид кучи после двух операций сборки мусора: выжившие объекты из поколения 0 переходят в поколение 1 (увеличивая его размер), поколение 0 пустует

Как видите, все объекты из поколения 0, пережившие сборку мусора, перешли в поколение 1. Так как сборщик не проверяет поколение 1, память, занятая объектом *B*, не освобождается, даже если этот объект на момент сборки мусора недоступен. И в этот раз после сборки мусора поколение 0 пустеет, в это поколение попадут новые объекты. Допустим, приложение работает дальше и выделяет память под объекты *L–O*. Во время работы приложение прекращает использовать объекты *G*, *L* и *M*, и они становятся недоступными. В результате куча выглядит так, как показано на рис. 21.12.



Рис. 21.12. В поколении 0 созданы новые объекты, количество мусора в поколении 1 увеличилось

Допустим, в результате размещения объекта *P* размер поколения 0 превысил пороговое значение, что инициировало сборку мусора. Поскольку все объекты поколения 1 занимают в совокупности меньше 2 Мбайт памяти, сборщик вновь решает собрать мусор только в поколении 0, игнорируя недоступные объекты в поколении 1 (*B* и *G*). Куча после сборки мусора показана на рис. 21.13.



Рис. 21.13. Вид кучи после трех операций сборки мусора: выжившие объекты из поколения 0 переходят в поколение 1 (увеличивая его размер); поколение 0 пустеет

На рисунке видно, что поколение 1 медленно, но верно растет. Допустим, поколение 1 выросло до таких размеров, что все его объекты в совокупности заняли 2 Мбайт памяти. В этот момент приложение продолжает работать (потому что сборка мусора только что завершилась) и начинает размещение в памяти объектов *P–S*, которые заполняют поколение 0 до его порогового значения (рис. 21.14).



Рис. 21.14. Новые объекты размещены в поколении 0, в поколении 1 появилось больше мусора

При попытке приложения разместить объект *T* поколение 0 заполняется и начинается сборка мусора. Однако на этот раз сборщик мусора обнаруживает, что место, занятое объектами, превысило пороговое значение. После нескольких операций сборки мусора в поколении 0 велика вероятность, что несколько объектов в поколении 1 стали недоступными (как в нашем примере). Поэтому теперь сборщик мусора проверяет все объекты поколений 1 и 0. После сборки мусора в обоих поколениях куча выглядит так, как показано на рис. 21.15.



Рис. 21.15. Вид кучи после четырех операций сборки мусора: выжившие объекты из поколения 1 переходят в поколение 2, выжившие объекты из поколения 0 переходят в поколение 1, поколение снова 0 пустеет

Все выжившие объекты поколения 0 теперь находятся в поколении 1, а все выжившие объекты поколения 1 — в поколении 2. Как всегда, сразу после сборки мусора поколение 0 пустеет: в нем будут размещаться новые объекты. В поколении 2 находятся объекты, проверенные сборщиком мусора не меньше двух раз. Операций сборки мусора может быть много, но объекты поколения 1 проверяются только тогда, когда их суммарный размер достигает порогового значения — до этого обычно проходит несколько операций сборки мусора в поколении 0.

Управляемая куча поддерживает только три поколения: 0, 1 и 2. Поколения 3 не существует¹. При инициализации в CLR устанавливается пороговое значение для всех трех поколений. Как отмечалось ранее, для поколения 0 оно равно

¹ Статический метод `MaxGeneration` класса `System.GC` возвращает 2.

примерно 256 Кбайт, для поколения 1 — около 2 Мбайт. Что касается поколения 2, то пороговое значение для него составляет около 10 Мбайт. Пороговые значения выбираются таким образом, чтобы добиться максимальной производительности. Чем больше пороговое значение, тем реже требуется сборка мусора. Опять же, повышение производительности достигается при верности исходных предположений: чем младше объект, тем короче его время жизни; чем старше объект, тем длиннее его время жизни.

Сборщик мусора CLR является самонастраивающимся, то есть в процессе работы он анализирует функциональность приложения и адаптируется. Например, если приложение создает множество объектов и пользуется ими очень недолго, сборка мусора в поколении 0 позволяет освободить много памяти. На самом деле, в поколении 0 можно освободить память всех объектов.

Если сборщик видит, что после сборки мусора в поколении 0 остается очень мало выживших объектов, он может снизить порог для поколения 0 с 256 до 128 Кбайт. В этом случае сборка мусора будет выполняться чаще, но это меньше загрузит сборщик, поэтому рабочий набор процесса останется небольшим. В сущности, если все объекты поколения 0 станут мусором, сборщику не придется даже дефрагментировать память — достаточно будет вернуть указатель `NextObjPtr` в начало поколения 0, чтобы посчитать сборку мусора законченной. Замечательный способ освобождения памяти!

ПРИМЕЧАНИЕ

Сборщик мусора отлично работает с приложениями, потоки которых большую часть времени бездействуют, находясь в верхней части стека. Когда у потока появляется работа, он просыпается, создает несколько объектов с коротким временем жизни, возвращает управление и опять засыпает. Такая архитектура реализована во многих приложениях, в том числе в приложениях Windows Forms, ASP.NET Web Forms, веб-службах XML.

В случае приложений ASP.NET поступает клиентский запрос, создается несколько новых объектов, которые выполняют работу от имени клиента, и результат возвращается клиенту. После этого все объекты, созданные для обслуживания клиентского запроса, становятся мусором. Иначе говоря, каждый запрос к приложению ASP.NET генерирует много мусора. Поскольку эти объекты почти сразу после своего создания становятся недоступными, каждая сборка мусора освобождает много памяти. Это позволяет поддерживать очень небольшой рабочий набор и достичь исключительного быстродействия сборщика.

В сущности, функционирование параметров и локальных переменных большинства корней приложения зависит от стека потока. Если стек потока небольшой, сборщик мусора быстро проверяет корни и маркирует доступные объекты. Иначе говоря, чтобы добиться быстрой сборки мусора, нужно избегать глубоких стеков, например, отказавшись от использования рекурсивных методов.

В то же время, если после обработки поколения 0 сборщик мусора обнаруживает множество выживших объектов, значит, удастся освободить мало памяти. В этом случае сборщик мусора может поднять порог для поколения 0, например до 512 Кбайт. В результате сборка мусора выполняется реже, но каждый раз будет освобождаться значительный объем памяти. Кстати, если сборщик освобождает недостаточно памяти, перед вбрасыванием исключения `OutOfMemoryException` он выполняет полную сборку мусора.

Я привел пример того, как сборщик динамически может изменять порог поколения 0, но сходным образом могут меняться пороги для поколений 1 и 2. При сборке мусора в этих поколениях сборщик определяет, сколько памяти было освобождено и сколько объектов осталось. В зависимости от полученных данных он может увеличить или уменьшить пороги для этих поколений, чтобы повысить производительность работы приложения. В итоге сборщик мусора автоматически адаптируется к загрузке памяти, необходимой для конкретного приложения, и это замечательно!

Показанный далее класс `GCNotification` напоминает программу `GCBeep`, которую мы обсуждали чуть раньше. Этот класс вызывает событие при появлении поколения 0 или поколения 2. Это событие заставляет компьютер подать звуковой сигнал, как только сборщик вычислит, сколько времени прошло между сборками и какой объем памяти был выделен. Данный класс позволяет проанализировать код приложения и понять, каким образом оно использует память:

```
public static class GCNotification {
    private static Action<Int32> s_gcDone = null; // Поле события

    public static event Action<Int32> GCDone {
        add {
            // Если зарегистрированные делегаты отсутствуют, начинаем оповещение
            if (s_gcDone == null) { new GenObject(0); new GenObject(2); }
            s_gcDone += value;
        }
        remove { s_gcDone -= value; }
    }

    private sealed class GenObject {
        private Int32 m_generation;
        public GenObject(Int32 generation) { m_generation = generation; }
        ~GenObject() { // Метод финализации
            // Если объект принадлежит нужному нам поколению (или выше),
            // оповещаем делегат о выполненной сборке мусора
            if (GC.GetGeneration(this) >= m_generation) {
                Action<Int32> temp = Interlocked.CompareExchange(
                    ref s_gcDone, null, null);
                if (temp != null) temp(m_generation);
            }
        }
    }
}
```

продолжение ➤

```
// Продолжаем оповещение, пока остается хоть один зарегистрированный
// делегат, домен приложений не выгружен и процесс не завершен
if ((s_gcDone != null)
    && !AppDomain.CurrentDomain.IsFinalizingForUnload()
    && !Environment.HasShutdownStarted) {
    // Для поколения 0 создаем объект; для поколения 2 воскрешаем
    // объект и позволяем сборщику вызвать метод финализации
    // при следующей сборке мусора для поколения 2
    if (m_generation == 0) new GenObject(0);
    else GC.ReRegisterForFinalize(this);
} else { /* Позволяем объекту исчезнуть */ }
}
}
```

Другие возможности сборщика мусора для работы с машинными ресурсами

Иногда машинный ресурс требует много памяти, а управляемый объект, являющийся его оболочкой, занимает очень мало памяти. Наиболее типичный пример — битовая карта. Она может занимать несколько мегабайтов машинной памяти, а управляемый объект может быть очень небольшим, так как содержит только 4- или 8-байтовое значение. С точки зрения CLR до сборки мусора процесс может выделять сотни битовых карт (которые займут мало управляемой памяти). Однако если процесс манипулирует множеством битовых карт, расходование памяти процессом начнет расти с огромной скоростью. Для исправления ситуации в классе GC предусмотрены два статических метода следующего вида:

```
public static void AddMemoryPressure(Int64 bytesAllocated);
public static void RemoveMemoryPressure(Int64 bytesAllocated);
```

Эти методы нужно использовать в классах, являющихся оболочкой для значительных машинных ресурсов, чтобы сообщать сборщику мусора о реальном объеме занятой памяти. Сам сборщик следит за этим показателем, и когда он становится большим, начинается сборка мусора.

Объем некоторых машинных ресурсов ограничен. Раньше в Windows разрешалось создавать всего пять контекстов устройства. Также ограничивалось число файлов, открываемых приложением. Опять же, с точки зрения CLR, до сборки мусора процесс может выделить память для сотен объектов (требующих мало памяти). Однако при количественном ограничении на применение машинных ресурсов попытка задействовать их больше, чем разрешено, обычно

приводит к вбрасыванию исключения. Для таких ситуаций в пространстве имен `System.Runtime.InteropServices` предусмотрен класс `HandleCollector`:

```
public sealed class HandleCollector {
    public HandleCollector(String name, Int32 initialThreshold);
    public HandleCollector(
        String name, Int32 initialThreshold, Int32 maximumThreshold);
    public void Add();
    public void Remove();
    public Int32 Count { get; }
    public Int32 InitialThreshold { get; }
    public Int32 MaximumThreshold { get; }
    public String Name { get; }
}
```

В классе, являющемся оболочкой машинного ресурса с количественным ограничением, должен использоваться экземпляр этого класса, чтобы сообщить сборщику мусора, сколько реально задействовано экземпляров этого ресурса. Внутренний код класса поддерживает счетчик занятых экземпляров, и когда значение счетчика становится большим, происходит сборка мусора.

ПРИМЕЧАНИЕ

Код методов `GC.AddMemoryPressure` и `HandleCollector.Add` вызывает `GC.Collect` для запуска сборщика мусора до достижения поколением 0 своего предела. Обычно настоятельно не рекомендуется принудительно вызывать сборщик мусора, потому что это отрицательно сказывается на производительности приложения. Однако вызов этих методов в классах призван обеспечить доступ приложения к ограниченному числу машинных ресурсов. Если машинных ресурсов окажется недостаточно, произойдет сбой приложения. Для большинства приложений лучше работать медленнее, чем не работать вообще.

Следующий код иллюстрирует использование и результат работы методов сжатия памяти и класса `HandleCollector`:

```
using System;
using System.Runtime.InteropServices;

public static class Program {
    public static void Main() {
        MemoryPressureDemo(0); // 0 вызывает нечастую сборку мусора
        MemoryPressureDemo(10 * 1024 * 1024); // 10 Мбайт вызывают частую
                                           // сборку мусора

        HandleCollectorDemo();
    }
}
```

```

private static void MemoryPressureDemo(Int32 size) {
    Console.WriteLine();
    Console.WriteLine("MemoryPressureDemo, size={0}", size);
    // Создание набора объектов с указанием их логического размера
    for (Int32 count = 0; count < 15; count++) {
        new BigNativeResource(size);
    }

    // В демонстрационных целях очищаем все
    GC.Collect();
    GC.WaitForPendingFinalizers();
}

private sealed class BigNativeResource {
    private Int32 m_size;

    public BigNativeResource(Int32 size) {
        m_size = size;
        if (m_size > 0) {
            // Пусть сборщик думает, что объект занимает больше памяти
            GC.AddMemoryPressure(m_size);
        }
        Console.WriteLine("BigNativeResource create.");
    }

    ~BigNativeResource() {
        if (m_size > 0) {
            // Пусть сборщик думает, что объект освободил больше памяти
            GC.RemoveMemoryPressure(m_size);
        }
        Console.WriteLine("BigNativeResource destroy.");
    }
}

private static void HandleCollectorDemo() {
    Console.WriteLine();
    Console.WriteLine("HandleCollectorDemo");
    for (Int32 count = 0; count < 10; count++) {
        new LimitedResource();
    }

    // В демонстрационных целях очищаем все
    GC.Collect();
    GC.WaitForPendingFinalizers();
}

private sealed class LimitedResource {
    // Создаем объект HandleCollector и передаем ему указание

```

```
// перейти к очистке, когда в куче появится два или более
// объектов LimitedResource
private static HandleCollector s_hc =
    new HandleCollector("LimitedResource", 2);

public LimitedResource() {
    // Сообщаем HandleCollector, что в кучу добавлен еще
    // один объект LimitedResource
    s_hc.Add();
    Console.WriteLine("LimitedResource create. Count={0}", s_hc.Count);
}
~LimitedResource() {
    // Сообщаем HandleCollector, что один объект LimitedResource
    // удален из кучи
    s_hc.Remove();
    Console.WriteLine("LimitedResource destroy. Count={0}", s_hc.Count);
}
}
```

После компиляции и запуска этого кода получаем примерно следующее:

```
MemoryPressureDemo, size=0
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource create.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
BigNativeResource destroy.
```

```
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.
```

```
MemoryPressureDemo, size=10485760
```

```
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource create.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.  
BigNativeResource destroy.
```

```
HandleCollectorDemo
```

```
LimitedResource create. Count=1  
LimitedResource create. Count=2  
LimitedResource create. Count=3  
LimitedResource destroy. Count=3  
LimitedResource destroy. Count=2  
LimitedResource destroy. Count=1
```

```
LimitedResource create. Count=1
LimitedResource create. Count=2
LimitedResource destroy. Count=2
LimitedResource create. Count=2
LimitedResource create. Count=3
LimitedResource destroy. Count=3
LimitedResource destroy. Count=2
LimitedResource destroy. Count=1
LimitedResource create. Count=1
LimitedResource create. Count=2
LimitedResource destroy. Count=2
LimitedResource create. Count=2
LimitedResource destroy. Count=1
LimitedResource destroy. Count=0
```

Прогнозирование успеха операции, требующей много памяти

Иногда приходится реализовывать алгоритмы, в которых используется несколько объектов, в совокупности занимающих большой объем памяти. Если в ходе исполнения этого алгоритма ресурсов памяти оказывается недостаточно, CLR выбрасывает исключение `OutOfMemoryException`. В этом случае вся работа идет насмарку. К тому же приходится перехватывать это исключение и корректно восстанавливать программу.

В пространстве имен `System.Runtime` есть класс `MemoryFailPoint`, позволяющий проверять доступный объем памяти до начала запуска ресурсоемких алгоритмов:

```
public sealed class MemoryFailPoint : CriticalFinalizerObject, IDisposable {
    public MemoryFailPoint(Int32 sizeInMegabytes);
    ~MemoryFailPoint();
    public void Dispose();
}
```

Пользоваться этим классом довольно просто. Сначала нужно создать его экземпляр, передав ему информацию об объеме памяти (в мегабайтах), который потребуется алгоритму (если точная цифра неизвестна, нужно указывать объем с запасом). Код конструктора выполняет следующие проверки, которые обуславливают дальнейшие действия.

1. Достаточно ли свободного места в страничном файле системы и непрерывного виртуального адресного пространства в процессе, чтобы удовлетворить запрос? Учтите, что конструктор вычитает объем памяти, который логически зарезервирован другим вызовом конструктора `MemoryFailPoint`.

2. Если памяти недостаточно, вызывается сборщик мусора, чтобы высвободить память.
3. Если места в страничном файле системы по-прежнему не хватает, делается попытка увеличения страничного файла. Если страничный файл не может быть увеличен до нужного размера, вбрасывается исключение `InsufficientMemoryException`.
4. Если непрерывного виртуального адресного пространства по-прежнему недостаточно, вбрасывается исключение `InsufficientMemoryException`.
5. Если найдено достаточно места в страничном файле и в виртуальном адресном пространстве, запрошенное число мегабайтов резервируется путем добавления этого числа в закрытое статическое поле, определенное в классе `MemoryFailPoint`. Добавление выполняется в безопасном режиме, чтобы несколько потоков могли одновременно создать экземпляры этого класса и быть уверенными, что для них логически зарезервирована запрошенная память (при условии, что в конструкторе не было брошено никакого исключения).

Если конструктор `MemoryFailPoint` вбрасывает исключение `InsufficientMemoryException`, в приложении можно освободить немного используемых ресурсов или снизить производительность (меньше кэшировать данные), чтобы снизить вероятность возникновения в будущем исключения `OutOfMemoryException`. Между прочим, исключение `InsufficientMemoryException` является производным от `OutOfMemoryException`.

ВНИМАНИЕ

Если конструктор `MemoryFailPoint` не вбрасывает исключение, значит, запрошенная память была логически зарезервирована и можно выполнить алгоритм, требующий много памяти. Однако учтите, что физически память еще не выделялась. То есть мы всего лишь немного повысили вероятность успешного выполнения алгоритма и получения необходимой памяти. Класс `MemoryFailPoint` не может гарантировать, что алгоритм получит необходимую память, даже если конструктор не вбросил исключение. Этот класс призван лишь помочь разработчику сделать приложение более надежным.

По окончании алгоритма нужно вызвать метод `Dispose` для созданного объекта `MemoryFailPoint`. Внутренний код `Dispose` просто вычитет (в безопасном режиме) зарезервированное число мегабайтов из статического поля `MemoryFailPoint`. Следующий код демонстрирует использование класса `MemoryFailPoint`:

```
using System;  
using System.Runtime;
```

```
public static class Program {  
    public static void Main() {  
        try {  
            // Логически резервируем 1,5 Гбайт памяти  
            using (MemoryFailPoint mfp = new MemoryFailPoint(1500)) {  
                // Выполняем алгоритм, требующий много памяти  
            } // Метод Dispose логически освобождает 1,5 Гбайт  
        }  
        catch (InsufficientMemoryException e) {  
            // Память не может быть зарезервирована  
            Console.WriteLine(e);  
        }  
    }  
}
```

Программное управление сборщиком мусора

Тип `System.GC` позволяет приложению напрямую управлять сборщиком мусора. Для начала замечу, что узнать максимальное поколение, поддерживаемое управляемой кучей, можно, прочитав значение свойства `GC.MaxGeneration`. Это свойство всегда возвращает 2.

Сборщик также можно заставить собрать мусор, вызвав один из трех статических методов:

```
void GC.Collect(Int32 Generation)  
void GC.Collect()  
void Collect(Int32 generation, GCCollectionMode mode)
```

Первый метод позволяет задать поколение (или несколько поколений), в котором нужно выполнить сбор мусора. Ему передаются целые числа от 0 до значения `GC.MaxGeneration` включительно. Если передать 0, будет выполнена сборка мусора в поколении 0, если 1 — мусор будет собран в поколениях 0 и 1, а если 2 — в поколениях 0, 1 и 2. Версия метода `Collect` без параметров принуждает выполнить полную сборку мусора во всех поколениях, его вызов эквивалентен вызову:

```
GC.Collect(GC.MaxGeneration);
```

Третья перегруженная версия метода `Collect` позволяет передать поколение и параметр `GCCollectionMode`. Описание различных значений этого параметра дано в табл. 21.1.

Таблица 21.1. Значения параметра `GC.CollectionMode`

Значение	Описание
Default	Аналогично вызову метода <code>GC.Collect</code> без флагов. В настоящее время эквивалентно передаче параметра <code>Forced</code> , но в следующих версиях CLR ситуация может измениться
Forced	Иницирует сборку мусора для всех поколений вплоть до указанного вами, включая и само это поколение
Optimized	Сборка мусора осуществляется только при условии качественного конечного результата, выражающегося либо в освобождении большого объема памяти, либо в уменьшении фрагментации. В противном случае вызов метода в этом режиме не дает никакого эффекта

Обычно следует избегать вызова любых методов `Collect`: лучше не вмешиваться в работу сборщика мусора и позволить ему самостоятельно настраивать пороговые значения для поколений, основываясь на реальном поведении приложения. Однако при написании приложения с консольным или графическим интерфейсом его код «владеет» процессом и CLR в этом процессе. В подобных приложениях порой следует собирать мусор принудительно во вполне определенное время. Это можно сделать при помощи метода `GC.CollectionMode` в режиме `Optimized`. Режимы `Default` и `Forced` обычно используют для отладки и тестирования.

Например, имеет смысл вызывать метод `Collect`, если только что произошло некое разовое событие, которое привело к уничтожению множества старых объектов. Вызов `Collect` в такой ситуации очень кстати, ведь основанные на прошлом опыте прогнозы сборщика мусора, скорее всего, для разовых событий окажутся неточными.

Например, в приложении имеет смысл выполнить принудительную сборку мусора во всех поколениях после инициализации приложения или сохранения пользователем файла с данными. Когда на веб-странице размещается элемент управления `Windows Form`, полная сборка мусора выполняется при каждой выгрузке страницы. Не нужно вручную вызывать метод `Collect`, чтобы сократить время отклика приложения, вызывайте его, чтобы уменьшить рабочий набор процесса.

Тип `GC` поддерживает также метод `WaitForPendingFinalizers`, который приостанавливает вызывающий поток, пока поток, обрабатывающий очередь на финализацию, не опустошит ее, вызвав метод финализации для каждого объекта. В большинстве приложений, скорее всего, этот метод вызывать не придется, но иногда я встречаю, например, такой код:

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

Этот код принудительно собирает мусор, освобождая память объектов, не требующих финализации. Однако память, занятую объектами, которые

требуют финализации, освобождать нельзя. Когда первый вызов `Collect` возвращает управление, выделенный поток финализации асинхронно вызывает методы финализации. Вызов `WaitForPendingFinalizers` переводит поток приложения в состояние ожидания до вызова всех методов финализации. Когда `WaitForPendingFinalizers` вернет управление, все финализированные объекты действительно станут мусором. Теперь второй вызов `Collect` снова выполняет принудительную сборку мусора, освобождающую память, занятую только что финализированными объектами.

В некоторых приложениях (особенно это касается серверных приложений, хранящих в памяти множество объектов) время на полную сборку мусора (до второго поколения) оказывается слишком большим. Более того, если сборка мусора длится слишком долго, может завершиться время ожидания клиентских запросов. Чтобы избежать подобных ситуаций, в классе `GC` имеется метод `RegisterForFullGCNotification`. С его помощью и при использовании дополнительных вспомогательных методов (`WaitForFullGCApproach`, `WaitForFullGCComplete` и `CancelFullGCNotification`) можно оповестить приложение о том, что сборщик мусора близок к выполнению полной сборки. В результате приложение сможет вызвать метод `GC.Collect` для принудительной сборки мусора в более подходящее время. Или свяжется с другими серверами, чтобы лучше распределить клиентские запросы. Дополнительную информацию об этих методах вы можете найти в документации на .NET Framework SDK. Имейте в виду, что методы `WaitForFullGCApproach` и `WaitForFullGCComplete` всегда вызываются вместе, так как CLR обрабатывает их попарно.

Наконец, у класса `GC` есть пара статических методов, позволяющих узнать, в каком поколении находится объект в настоящее время:

```
Int32 GetGeneration(Object obj)
Int32 GetGeneration(WeakReference wr)
```

Первая версия `GetGeneration` принимает в качестве параметра ссылку на объект, вторая — ссылку на `WeakReference`. Возвращенное этими методами значение лежит в диапазоне от 0 до `GC.MaxGeneration` включительно.

Следующий код поможет разобраться в работе поколений, он также демонстрирует использование указанных методов типа `GC`:

```
using System;
```

```
internal sealed class GenObj {
    ~GenObj() {
        Console.WriteLine("In Finalize method");
    }
}
```

```
public static class Program {
    public static void Main() {
```

продолжение ➤

```

Console.WriteLine("Maximum generations: " + GC.MaxGeneration);

// Создание нового объекта GenObj в куче
Object o = new GenObj();

// Поскольку этот объект недавно создан, он помещается в поколение 0
Console.WriteLine("Gen " + GC.GetGeneration(o)); // 0

// Сборка мусора переводит объект в следующее поколение
GC.Collect();
Console.WriteLine("Gen " + GC.GetGeneration(o)); // 1

GC.Collect();
Console.WriteLine("Gen " + GC.GetGeneration(o)); // 2

GC.Collect();
Console.WriteLine("Gen " + GC.GetGeneration(o)); // 2
// (максимальное значение)

o = null; // Уничтожаем жесткую ссылку на объект

Console.WriteLine("Collecting Gen 0");
GC.Collect(0); // Сборка мусора в поколении 0
GC.WaitForPendingFinalizers(); // Метод финализации НЕ вызывается

Console.WriteLine("Collecting Gens 0, and 1");
GC.Collect(1); // Сборка мусора в поколениях 0 и 1
GC.WaitForPendingFinalizers(); // Метод финализации НЕ вызывается

Console.WriteLine("Collecting Gens 0, 1, and 2");
GC.Collect(2); // То же, что и Collect()
GC.WaitForPendingFinalizers(); // Вызывается метод финализации
}
}

```

А вот результат компоновки и запуска этого кода:

```

Maximum generations: 2
Gen 0
Gen 1
Gen 2
Gen 2
Collecting Gen 0
Collecting Gens 0, and 1
Collecting Gens 0, 1, and 2
In Finalize method

```

Захват потока

Ранее я объяснил, как функционирует сборщик мусора, но это объяснение было рассчитано на работу только одного потока. В реальном мире весьма вероятно, что несколько потоков будут обращаться к управляемой куче или хотя бы работать с размещенными в ней объектами. Когда в результате работы одного из потоков инициируется сборка мусора, остальные не должны обращаться к каким бы то ни было объектам (включая ссылки на объекты в собственном стеке), так как сборщик мусора может переместить эти объекты, изменив их адреса в памяти.

Итак, когда сборщик начинает сборку мусора, нужно приостановить все потоки, исполняющие управляемый код. У CLR есть несколько механизмов, позволяющих безопасно приостановить исполнение потоков, чтобы можно было собрать мусор, а потоки смогли работать как можно дольше и издержки остались минимальными. Здесь я не хочу вдаваться в детали — достаточно сказать, что специалистам Microsoft пришлось изрядно поработать, чтобы снизить издержки на сборку мусора. Microsoft продолжит совершенствовать эти механизмы, чтобы обеспечить эффективную сборку мусора в дальнейшем.

Собираясь начать сборку мусора, CLR немедленно приостанавливает все потоки, исполняющие управляемый код, а затем изучает указатели команд каждого потока, чтобы выяснить, на каком этапе находится процесс выполнения. Далее CLR сравнивает адрес указателя команды с таблицами, сгенерированными JIT-компилятором, чтобы определить, какой код выполняется потоком.

Если указатель команды потока находится по смещению, указанному в таблице, говорят, что поток достиг *безопасной точки* (safe point), то есть такого места, где его можно приостановить до завершения сборки мусора. В противном случае говорят, что безопасная точка не достигнута, и CLR не может собрать мусор. В этом случае CLR *захватывает* (hijacks) поток и изменяет его стек так, чтобы адрес возврата указывал на специальную внутреннюю функцию, реализованную в CLR. После этого исполнение потока возобновляется. После возвращения управления текущим методом будет исполнена специальная функция, которая приостановит поток.

Однако можно довольно долго ждать, пока метод потока вернет управление, поэтому после возобновления исполнения потока CLR ждет захвата потока около 250 мс, после чего вновь приостанавливает поток и проверяет его указатель команд. Если поток достиг безопасной точки, можно начинать сборку мусора. В противном случае CLR проверяет, не вызван ли другой метод. Если да, CLR вновь изменяет стек, чтобы захватить поток после возвращения из последнего исполняемого им метода. Затем CLR возобновляет поток и ждет еще несколько миллисекунд, прежде чем сделать следующую попытку.

Когда все потоки достигают безопасной точки или захватываются, можно начинать сборку мусора. По его завершении все потоки возобновляются, а приложение продолжает работу. Захваченные потоки возвращаются к методу, который изначально их вызвал.

В этом алгоритме есть маленькая хитрость. Когда среда CLR собирается запустить сборщик мусора, она приостанавливает все потоки, исполняющие управляемый код, но не приостанавливает потоки, исполняющие неуправляемый код. Когда все потоки, исполняющие управляемый код, оказываются в безопасной точке или захватываются, сборщик может приступить к работе. Потоки, исполняющие неуправляемый код, могут продолжать работу, потому что для объектов, которые они используют, должен быть установлен флаг `Pinned`. Если такой поток возвращает управление управляемому коду, его выполнение сразу же приостанавливается до завершения сборки мусора.

Оказывается, CLR прибегает к захвату гораздо чаще, чем использует таблицы, сгенерированные JIT-компилятором и позволяющие узнать, находится ли объект в безопасной точке. Причина в том, что эти таблицы требуют много памяти и увеличивают рабочий набор, что существенно снижает производительность. Поэтому они содержат информацию о разделах кода, в которых есть циклы, не вызывающие другие методы. Если в методе есть цикл, вызывающий другой метод, или если циклов нет вообще, таблицы, сгенерированные JIT-компилятором, не несут в себе много информации, а для приостановки потока используется захват.

Режимы сборки мусора

При запуске CLR выбирается один из режимов сборки мусора, который не может быть изменен до завершения процесса. Существует два основных режима сборки мусора:

- ❑ **Рабочая станция.** Этот режим настраивает сборку мусора для приложений на стороне клиента. Сборщик предполагает, что остальные приложения не используют ресурсы процессора. Существует два подчиненных режима: с *параллельной сборкой мусора* (`concurrent collector`) и без нее. О них мы поговорим чуть позже.
- ❑ **Сервер.** Этот режим настраивает сборку мусора для приложений на стороне сервера. Сборщик предполагает, что на машине не запущено никаких сторонних приложений (клиентских или серверных), поэтому все ресурсы процессора можно бросить на сборку мусора. В этом режиме управляемая куча разбирается на несколько разделов — по одному на процессор. Изначально сборщик мусора использует один поток на один процессор. Каждый поток выполняется в собственном разделе одновременно с другими потоками. Такой подход хорошо работает в случае приложений с единообразным по-

ведением рабочих потоков. Функция доступна на компьютерах с несколькими процессорами. Только в этом случае параллельная обработка потоков позволяет получить прирост производительности.

По умолчанию приложения запускаются в режиме рабочей станции с включенным режимом параллельной сборки мусора. А серверные приложения (например, ASP.NET или SQL Server), обеспечивающие хостинг CLR, могут потребовать загрузки режима сервера. Однако если серверное приложение запускается на однопроцессорной машине, CLR включает режим рабочей станции с отключенным режимом параллельной сборки мусора.

Приложение, обеспечивающее хостинг среды CLR, может заставить ее использовать серверный сборщик мусора путем создания конфигурационного файла (о том, как это сделать, рассказывалось в главах 2 и 3), содержащего элемент `gcServer`. Вот пример конфигурационного файла:

```
<configuration>
  <runtime>
    <gcServer enabled="true"/>
  </runtime>
</configuration>
```

Узнать, запущена ли среда CLR в серверном GC-режиме, можно при помощи логического свойства `IsServerGC` класса `GCSettings`, предназначенного только для чтения:

```
using System;
using System.Runtime; // GCSettings находится в этом пространстве имен

public static class Program {
    public static void Main() {
        Console.WriteLine(
            "Application is running with server GC=" + GCSettings.IsServerGC);
    }
}
```

В многопроцессорной системе, работающей под управлением версии исполняющего механизма для рабочих станций, у сборщика мусора есть дополнительный фоновый поток, параллельно утилизирующий объекты во время работы приложения. Когда поток размещает в памяти объект, вызывающий превышение порога для поколения 0, сборщик сначала приостанавливает все потоки, а затем определяет поколения, в которых нужно выполнить сборку мусора. Если сборщик должен собрать мусор в поколении 0 или 1, он работает, как обычно, но если нужно собрать мусор в поколении 2, размер поколения 0 увеличивается выше порогового, чтобы разместить новый объект, а затем исполнение потоков приложения возобновляется.

Пока работают потоки приложения, отдельный поток сборщика с нормальным приоритетом маркирует все недоступные объекты в фоновом режиме. Этот

поток конкурирует за процессорное время с потоками приложения, замедляя его работу. Однако параллельный сборщик работает только в многопроцессорных системах, поэтому снижение скорости почти незаметно. После того как объекты маркированы, сборщик вновь приостанавливает все потоки и решает, нужно ли дефрагментировать память. Если он принимает положительное решение, память дефрагментируется, ссылки корней исправляются, и исполнение потоков приложения возобновляется — такая сборка мусора обычно проходит быстрее, так как перечень недоступных объектов создается заранее. Однако сборщик может отказаться от дефрагментации памяти, что, на самом деле, предпочтительнее. Если свободной памяти много, сборщик не станет дефрагментировать кучу — это повышает быстродействие, но увеличивает рабочий набор приложения. Прибегая к параллельной сборке, приложение обычно расходует больше памяти, чем при непараллельной сборке.

Подводя итог, можно сказать, что в режиме параллельной сборки впечатление пользователей от работы с программой улучшается, поэтому этот режим лучше подходит для интерактивных приложений с консольным или графическим интерфейсом. Однако в некоторых приложениях режим параллельной сборки только снижает быстродействие и увеличивает потребление памяти. Тестируя приложение, поэкспериментируйте, включая и отключая режим параллельной сборки мусора, чтобы выяснить, какой подход обеспечивает максимальную производительность и минимальный расход памяти для приложения.

Можно запретить CLR использовать режим параллельной сборки, создав конфигурационный файл приложения, содержащий элемент `gcConcurrent` (см. главы 2 и 3).

Вот пример такого файла:

```
<configuration>
  <runtime>
    <gcConcurrent enabled="false"/>
  </runtime>
</configuration>
```

Вдобавок к описанным режимам, сборщик мусора поддерживает выделение памяти без синхронизации. В многопроцессорной системе поколение 0 управляемой кучи разделено на несколько арен памяти — по одной на каждый поток. Это позволяет нескольким потокам выделять память одновременно, не требуя монопольного доступа к куче.

Несмотря на то что конфигурация GC-режима не может быть изменена до завершения процесса, приложение может контролировать сборку мусора при помощи свойства `GCLatencyMode` класса `GCSettings`. Этому свойству могут присваиваться любые значения из перечисления `GCLatencyMode`. Возможные варианты перечислены в табл. 21.2.

Таблица 21.2. Значения, определенные в перечислении `GCLatencyMode`

Значение	Описание
Batch (по умолчанию используется для серверного режима)	В режиме рабочей станции этот скрытый режим отключает подрежим параллельной сборки мусора. Для серверного режима это — единственно возможный скрытый режим
Interactive (по умолчанию используется для режима рабочей станции)	В режиме рабочей станции этот скрытый режим включает подрежим параллельной сборки мусора. Для серверного режима этот скрытый режим невозможен
LowLatency	В режиме рабочей станции этот скрытый режим используется для кратковременных, срочных операций (например, рисования анимации), для которых сборка мусора в поколении 2 из-за снижения производительности может оказаться неприемлемой. Для серверного режима этот скрытый режим невозможен

Последний режим требует дополнительных пояснений. Обычно его включают для реализации операций, для которых важно время выполнения, а затем возвращают режим `Batch` или `Interactive`. Однако в режиме `LowLatency` сборщик мусора действительно обходит вниманием поколение 2, так как это может занять много времени. Разумеется, если вы вызовете метод `GC.Collect()`, поколение 2 также отправится в мусор. То же самое произойдет, если Windows «пожалуется» CLR на недостаток системной памяти (этот вопрос обсуждался ранее в этой главе).

В режиме `LowLatency` приложение часто вбрасывает исключение `OutOfMemoryException`. Соответственно, можно посоветовать включать этот режим на максимально короткое время, избегать размещения в памяти многих объектов, а также больших объектов и возвращаться к режимам `Batch` и `Interactive` при помощи области ограниченного выполнения (см. главу 20). Также помните, что режим `LowLatency` является настройкой уровня процесса, и потоки могут быть запущены параллельно. Эти потоки могут даже менять данную настройку в процессе ее использования другим потоком. В этом случае вы можете добавить обновляющийся счетчик (управляемый при помощи методов `Interlocked`). Вот пример корректного использования режима `LowLatency`:

```
private static void LowLatencyDemo() {
    GCLatencyMode oldMode = GCSettings.LatencyMode;
    System.Runtime.CompilerServices.RuntimeHelpers.PrepareConstrainedRegions();
    try {
        GCSettings.LatencyMode = GCLatencyMode.LowLatency;
        // Здесь выполняется код
    }
    finally {
        GCSettings.LatencyMode = oldMode;
    }
}
```

Большие объекты

Есть и еще один инструмент повышения быстродействия, о котором стоит рассказать. Любые объекты размером 85 000 байт и более считаются большими. Память для них выделяется в специальной куче для больших объектов. Объекты в этой куче финализируются и освобождаются так же, как маленькие объекты, о которых мы здесь говорили. Однако большие объекты никогда не дефрагментируются, так как на перемещение блоков памяти размером в 85 000 байт ушло бы слишком много процессорного времени. Тем не менее не следует писать код с расчетом на то, что большие объекты не перемещаются в памяти, потому что в будущем размер больших объектов может измениться. Чтобы обеспечить неподвижность объекта в памяти, отметьте его флагом Pinned.

Большие объекты всегда считаются частью поколения 2, поэтому их следует создавать лишь для ресурсов, которые должны жить долго. Размещение в памяти короткоживущих больших объектов приведет к необходимости частой сборки мусора в поколении 2, что снижает производительность. Следующая программа подтверждает тот факт, что большие объекты всегда размещаются в поколении 2:

```
using System;
public static class Program {
    public static void Main() {
        Object o = new Byte[85000];
        Console.WriteLine(GC.GetGeneration(o)); // Выводится 2, а не 0
    }
}
```

Все эти механизмы работают прозрачно для кода приложения. Для разработчика они выглядят просто как дополнительная управляемая куча. И существуют они лишь для повышения производительности приложения.

Мониторинг сборки мусора

Существуют методы, которые можно вызвать для мониторинга сборщика мусора в процессе. Так, класс GC поддерживает следующие статические методы, вызываемые для выяснения числа операций сборки мусора в конкретном поколении или для объема памяти, занятого в данный момент объектами в управляемой куче.

```
Int64 GetTotalMemory(Boolean forceFullCollection);
Int32 CollectionCount(Int32 generation);
```

Чтобы выполнить профилирование конкретного блока кода, я часто вставляю до и после него код, вызывающий эти методы, а затем вычисляю разницу.

сказывается на рабочем усора произошло при ис- значит, нужно поработать

отдельным доменом при- 22.

я также набор счетчиков еальном времени самые нные можно просматри- го монитора из состава ' монитору, запустив ути- струментов, в результате на рис. 21.16.

счетчики памяти .NET CLR в окне PerfMon.exe

рки мусора в СЛ-
ватем укажите в
етчиков для мо
еперь системный
ого статистичес
те его и установ
ый инструмент д
CLR Profiler. Он
имки кучи и грас
орый можно ис
илирования и до

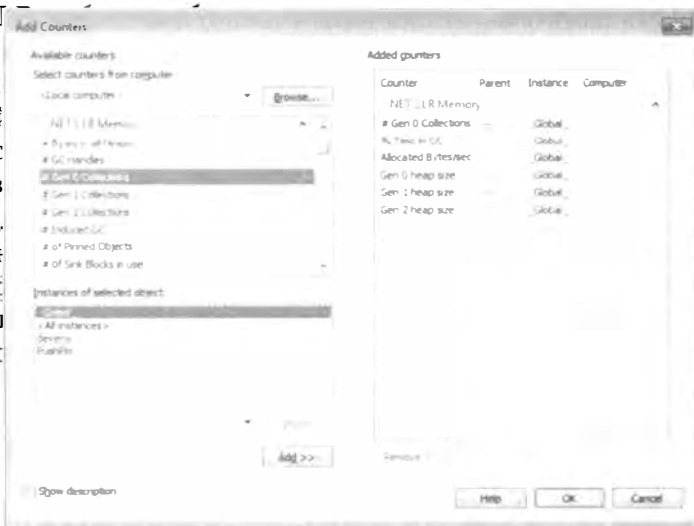


Рис. 21.16. Счетчики памяти .NET CLR в окне PerfMon.exe

Для мониторинга сборки мусора в CLR выберите объект производительности .NET CLR Memory, затем укажите в списке нужное приложение. В завершение выберите набор счетчиков для мониторинга, щелкните на кнопке Add, затем — на кнопке OK. Теперь системный монитор будет в реальном времени строить график выбранного статистического показателя. Чтобы узнать, что означает счетчик, выделите его и установите флажок Show Description.

Еще один замечательный инструмент для мониторинга размещения объектов приложением называется CLR Profiler. Он позволяет выполнять профилирование вызовов, получать снимки кучи и графики использования памяти. Имеется даже API-интерфейс, который можно использовать из тестового кода для запуска и остановки профилирования и добавления комментариев в журналы.

Также доступен исходный код этого инструмента, поэтому разработчики могут изменять его по своему усмотрению. Вы можете найти его в Интернете, указав в строке поиска «CLR profiler». Это исключительно полезный инструмент — настоятельно вам его рекомендую!

Ну и, наконец, можно воспользоваться отладочным расширением (SOS.dll), помогающим при проблемах с памятью и других проблемах CLR. Это расширение позволяет узнать, сколько памяти выделено для процесса в управляемой куче, вывести все объекты, зарегистрированные для финализации и помещенные в очередь, просмотреть записи в таблице GCHandle как для домена приложений, так и для всего процесса, проверить корни, сохраняющие объект в куче живым, и многое другое.

Глава 22. Хостинг CLR и домены приложений

В этой главе обсуждаются две темы, позволяющие по-настоящему оценить достоинства Microsoft .NET Framework, — *хостинг* (hosting) и *домены приложений* (AppDomains). Благодаря хостингу любое приложение может использовать возможности общеязыковой среды CLR, в частности существующие приложения можно, по крайней мере, частично, переписать при помощи управляемого кода. Кроме того, хостинг позволяет настраивать и дополнять приложения программными средствами.

Поддержка дополнений означает возможность включения в свои программы кода сторонних разработчиков. В Microsoft Windows загрузка чужих DLL-библиотек была исключительно рискованным мероприятием. В такой библиотеке очень легко мог оказаться код, разрушающий структуры данных, и код приложений. Кроме того, библиотека могла использовать контекст безопасности приложений для получения доступа к ресурсам, к которым в обычных условиях доступа у нее нет. Домены приложений позволили решить эти проблемы. Именно они дают возможность запускать не пользующийся доверием код сторонних разработчиков, а CLR гарантирует безопасность и целостность структур данных и кода, а также невозможность использовать в неблагоприятных целях контекст безопасности.

Обычно хостинг и домены приложений используют наряду с загрузкой сборок и отражением. Совместное применение этих четырех технологий превращает CLR в невероятно богатую и мощную платформу. Эта глава посвящена в основном хостингу и доменам приложений, а о загрузке сборок и отражении рассказывается в следующей главе. Изучив и освоив эти технологии, вы узнаете, почему нынешние инвестиции в .NET Framework в будущем вернутся вам сторицей.

Хостинг CLR

Платформа .NET Framework работает поверх Microsoft Windows. Это значит, что в ее основе должны лежать технологии, с которыми Windows может взаимодействовать. Для начала все файлы управляемых модулей и сборок должны иметь формат PE (portable executable), являться исполняемыми файлами Windows (EXE) или динамически подключаемыми библиотеками (DLL).

В Microsoft разработали CLR в виде COM-сервера, содержащегося в DLL. То есть разработчики определили для CLR стандартный COM-интерфейс и присвоили этому интерфейсу и COM-серверу глобально уникальные идентификаторы (GUID). При установке .NET Framework COM-сервер, представляющий CLR, регистрируется в реестре Windows как любой другой COM-сервер. Подробнее см. заголовочный файл C++ `MetaHost.h` из .NET Framework SDK — в этом файле определены все GUID-идентификаторы и неуправляемый интерфейс `ICLRMetaHost`.

Любое Windows-приложение может стать хостом для CLR. Однако не следует создавать экземпляры COM-сервера CLR при помощи функции `CoCreateInstance`, вместо этого неуправляемый хост должен вызывать функцию `CLRCreateInstance`, объявленную в файле `MetaHost.h`. Эта функция реализована в библиотеке `MSCorEE.dll`, которая обычно расположена в каталоге `C:\Windows\System32`. Об этой библиотеке обычно говорят как о *согласователе* (*shim*) — она не содержит COM-сервер CLR, а только определяет, какую версию CLR следует создать.

На одной машине допускается установка нескольких версий CLR, но может быть только одна версия файла `MSCorEE.dll` (согласователь)¹. Версия библиотеки `MSCorEE.dll` совпадает с версией самой последней установленной среды CLR, поэтому эта версия `MSCorEE.dll` «знает», как найти любые более ранние версии CLR, которые устанавливались на машине.

Код CLR содержится в файле, имя которого зависит от версии. Для версий 1.0, 1.1 и 2.0 это файл `MSCorWks.dll`, а для версии 4.0 — файл `Clr.dll`. Так как на один компьютер можно установить несколько версий CLR, эти файлы располагаются в разных папках²:

- ❑ версия 1.0 — в папке `C:\Windows\Microsoft.NET\Framework\v1.0.3705`;
- ❑ версия 1.1 — в папке `C:\Windows\Microsoft.NET\Framework\v1.0.4322`;
- ❑ версия 2.0 — в папке `C:\Windows\Microsoft.NET\Framework\v2.0.50727`;
- ❑ версия 4.0 — в папке `C:\Windows\Microsoft.NET\Framework\v4.0.21006`.

Функция `CLRCreateInstance` возвращает интерфейс `ICLRMetaHost`. Хост-приложение может вызывать функцию `GetRuntime` этого интерфейса, указывая ту версию CLR, которую следует создать. После этого согласователь загружает эту версию в текущий процесс.

По умолчанию согласователь анализирует управляемый исполняемый файл и извлекает из него сведения о версии CLR, с которой было скомпоно-

¹ В 64-разрядной версии Windows в действительности установлены два варианта файла `MSCorEE.dll`. Первый — это 32-разрядная версия x86, расположенная в папке `C:\Windows\SysWOW64`, второй — 64-разрядная версия x64 или IA64 (в зависимости от архитектуры процессора), расположенная в папке `C:\Windows\System32`.

² Обратите внимание, что версии .NET Framework 3.0 и 3.5 поставляются с CLR 2.0. Я не указываю, в каких папках находятся версии .NET Framework 3.0 и 3.5, так как DLL-библиотека CLR загружается из папки `v2.0.50727`.

вано и протестировано приложение. Однако приложение может переопределить заданные по умолчанию сведения, записав элементы `requiredRuntime` и `supportedRuntime` в конфигурационный XML-файл (см. главы 2 и 3).

Функция `GetRuntime` возвращает указатель на неуправляемый интерфейс `ICLRRuntimeInfo`, из которого при помощи метода `GetInterface` получается интерфейс `ICLRRuntimeHost`. Вызывая методы этого интерфейса, хост-приложение может выполнять следующие операции:

- ❑ Устанавливать хост-диспетчеры (`host managers`), то есть сообщать CLR, что хост должен участвовать в решениях, связанных с выделением памяти, планированием и синхронизацией потоков, загрузкой сборок и т. п. Кроме того, хосту могут понадобиться уведомления о начале и окончании сборки мусора, а также о завершении определенных операций.
- ❑ Получать информацию о CLR-диспетчерах, то есть запрещать CLR использовать определенные классы или члены. Кроме того, хост может указать подлежащий и неподлежащий отладке код, а также методы, вызываемые при наступлении определенных событий, таких как выгрузка домена приложения, остановка CLR или исключение, вызванное переполнением стека.
- ❑ Инициализировать и запустить CLR.
- ❑ Загружать сборку и исполнять ее код.
- ❑ Останавливать CLR, предотвращая дальнейшее исполнение управляемого кода в Windows-процессе.

Существуют многочисленные доводы в пользу хостинга CLR. К примеру, он дает любому приложению доступ к возможностям CLR и программным средствам, а также хотя бы частично быть написанным на управляемом коде. Многие приложения, обеспечивающие хостинг исполняющей среды, предлагают массу возможностей разработчикам, стремящимся к расширению функциональности. Вот только некоторые возможности:

- ❑ программирование на любом языке;
- ❑ код может компилироваться (а не интерпретироваться) JIT-компилятором, что обеспечивает максимальное быстродействие;
- ❑ поддержка сборки мусора, предотвращающей утечки и повреждение памяти;
- ❑ выполнение кода в безопасной изоляции;
- ❑ хосту не нужно заботиться о предоставлении многофункциональной среды разработки, вместо этого он использует имеющиеся технологии: языки, компиляторы, редакторы, отладчики, средства профилирования и пр.

Тем, кто интересуется подробностями хостинга CLR, я настоятельно рекомендую отличную книгу Стивена Претчнера (Steven Pratschner) «Customizing the Microsoft .NET Framework Common Language Runtime» (Microsoft Press, 2005), несмотря на то что в ней рассматриваются более ранние, чем 4.0, версии CLR.

ПРИМЕЧАНИЕ

Конечно, Windows-процессы могут обойтись и без CLR. Эта среда нужна только для исполнения в процессе управляемого кода. До появления .NET Framework 4.0 внутри Windows-процесса допускался только один экземпляр CLR. То есть процесс мог не содержать CLR вообще или же содержать какую-нибудь из имеющихся версий — 1.0, 1.1 или 2.0. Это было серьезное ограничение. Например, в Microsoft Office Outlook было невозможно загрузить два дополнительных компонента, если они создавались и тестировались на разных версиях .NET Framework.

К счастью, в .NET Framework 4.0 поддерживается возможность загрузки версий 2.0 и 4.0 в один Windows-процесс, позволяя компонентам, написанным для .NET Framework 2.0 и 4.0, работать бок о бок, не испытывая проблем совместимости. Эта без преувеличения фантастическая возможность позволяет применять компоненты .NET Framework в самых разных сценариях. Узнать, какая версия или версии CLR загружены в определенный процесс, можно с помощью утилиты ClrVer.exe.

Загруженную в Windows-процесс среду CLR выгрузить уже нельзя. Методы AddRef и Release не влияют на интерфейс ICLRRuntimeHost. Вы можете только завершить процесс, вынудив Windows очистить все занятые в нем ресурсы.

Домены приложений

В ходе инициализации COM-сервер CLR создает *домен приложений* (AppDomain), представляющий собой логический контейнер для набора сборок. Первый из созданных доменов называют *основным* (default AppDomain), он уничтожается только при завершении Windows-процесса.

Помимо основного, хост, использующий методы неуправляемого COM-интерфейса или методы управляемого типа, может заставить CLR создать дополнительные домены приложений. Их основной задачей является обеспечение изоляции. Домены приложений удобны благодаря нескольким свойствам.

- ❑ **Объекты, созданные одним доменом приложений, недоступны для кода других доменов.** Когда код домена приложений создает объект, домен становится «хозяином» этого объекта. Иначе говоря, время жизни объекта ограничивается временем существования самого домена. Код другого домена может получить доступ к объекту, только используя семантику *продвижения по ссылке* (marshal-by-reference) или *по значению* (marshal-by-value). Это позволяет «герметично» изолировать код в домене приложений, так как код в одном домене не может напрямую ссылаться на объект, созданный в другом домене. Такая изоляция позволяет легко выгружать домены приложений из процесса без влияния на работу других доменов.

- ❑ **Домены приложений можно выгружать.** CLR не поддерживает выгрузку отдельных сборок. Однако можно заставить CLR выгрузить домен приложений со всеми содержащимися в нем в данный момент сборками.
- ❑ **Домены приложений можно индивидуально защищать.** При создании домену приложений можно назначить набор разрешений, определяющий максимальные права запущенных в нем сборок. Это позволяет хосту загружать код и быть уверенным, что этот код не испортит или не прочитает важные структуры данных, используемые самим доменом.
- ❑ **Домены приложений можно индивидуально конфигурировать.** Каждый домен имеет целый набор конфигурационных параметров. Они в основном определяют, как среда CLR должна загружать сборки в домен. Существуют и параметры, определяющие пути поиска, перенаправление привязки к версиям и оптимизацию загрузчика.

ВНИМАНИЕ

Windows предоставляет замечательную возможность запускать каждое приложение в собственном адресном пространстве. Это гарантирует, что код одного приложения не получит доступа к коду и данным другого. Изоляция процессов предотвращает появления брешей в системе безопасности, повреждение данных и другие неприятности, обеспечивая надежность Windows и работающих в этой операционной системе приложений. К сожалению, создание процесса в Windows — операция очень ресурсоемкая. Win32-функция `CreateProcess` выполняется медленно, а виртуализация адресного пространства процесса требует много памяти.

Однако если приложение полностью состоит из гарантированно безопасного управляемого кода, который к тому же не вызывает неуправляемый код, можно запустить несколько управляемых приложений в одном Windows-процессе. Их домены обеспечат изоляцию, необходимую для защиты, конфигурирования и завершения отдельных приложений.

На рис. 22.1 показан отдельный Windows-процесс, в котором работает один COM-сервер CLR, управляющий двумя доменами приложений (кстати, не существует жестких ограничений на количество доменов приложений, которые могут выполняться в одном Windows-процессе). У каждого такого домена есть собственная куча загрузчика, ведущая учет обращений к типам с момента создания домена (эти типы были подробно рассмотрены в главе 4). Каждому типу в куче загрузчика соответствует таблица методов, строки которой указывают на код метода (если этот метод хоть раз исполнялся, его код уже скомпилирован JIT-компилятором).

Кроме того, в каждый домен приложений загружены сборки. В первый (он же основной) загружены три сборки: `MyApp.exe`, `TypeLib.dll` и `System.dll`, во второй — две сборки: `Wintellect.dll` и `System.dll`.

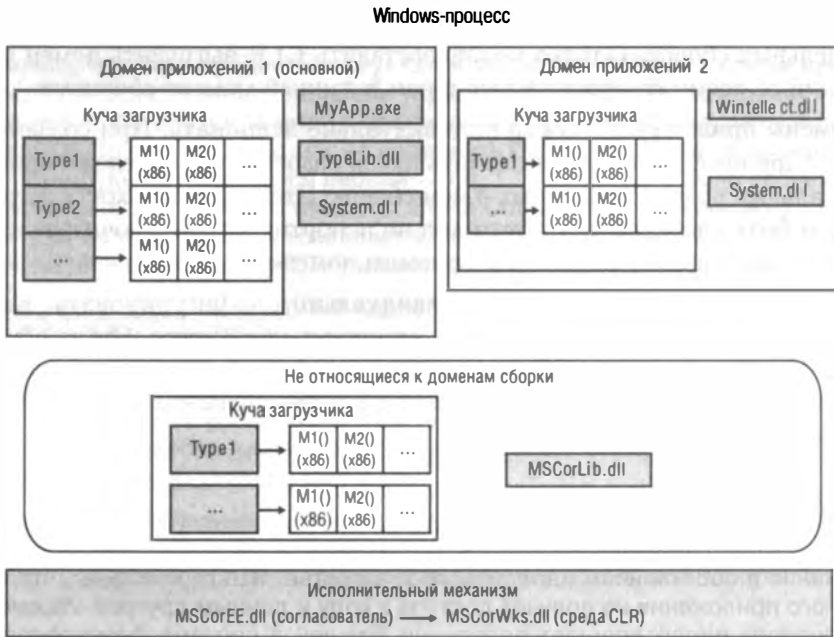


Рис. 22.1. Windows-процесс, являющийся хостом для CLR и двух доменов приложений

Обратите внимание, что сборка **System.dll** загружается в оба домена. Если в обоих доменах используется один тип из **System.dll**, в их кучах загрузчика будут размещены объекты одинаковых типов; память, выделенная под эти объекты, не используется доменами совместно. Более того, когда код домена вызывает определенные в типе методы, IL-код метода JIT-компилируется и результирующий машинный код привязывается к каждому домену в отдельности, то есть он не используется ими совместно.

Хотя отсутствие совместного использования памяти для хранения объектов или машинного кода расточительно, это оправдано, поскольку домены приложений разрабатывались для изоляции; у CLR должна быть возможность выгрузить домен приложений и освободить все его ресурсы, никак не затронув остальные домены. Эту возможность обеспечивает дублирование структур данных CLR. Это также гарантирует, что при использовании разными доменами одного типа статические поля будут задаваться отдельно для каждого домена.

Некоторые сборки предназначены для совместного использования разными доменами. Лучший пример — сборка **MSCorLib.dll**, созданная в Microsoft. Именно ей принадлежат типы **System.Object**, **System.Int32** и другие типы, неотделимые от .NET Framework. Эта сборка автоматически загружается при инициализации CLR, и домены приложений совместно используют ее типы. Для экономии ресурсов **MSCorLib.dll** загружается как сборка, не связанная

с конкретным доменом. Объекты всех типов в этой куче загрузчика и весь машинный код методов этих типов совместно используются всеми доменами процесса. К сожалению, для достижения всех преимуществ от совместного использования ресурсов приходится кое-чем жертвовать: сборки, загруженные без привязки к доменам, нельзя выгружать до завершения процесса. Единственный способ вернуть ресурсы — завершить процесс.

Доступ к объектам из других доменов

Код, расположенный в одном домене приложений, способен взаимодействовать с типами и объектами другого домена. Однако доступ к этим типам и объектам возможен только через тщательно определенные механизмы. Показанный далее пример демонстрирует процедуру создания домена приложений, загрузку в него сборки и конструирование определенного в этой сборке экземпляра типа. Код иллюстрирует различное поведение при конструировании типа, передаваемого путем продвижения по ссылке и по значению, а также типа, который вообще не использует механизм продвижения. Кроме того, демонстрируется, как объекты, переданные посредством разных вариантов продвижения, ведут себя при выгрузке создавшего их домена приложений. В этом примере мало кода, зато много комментариев. Затем следуют подробные объяснения, что именно делает CLR:

```
private static void Marshalling() {  
    // Получаем ссылку на домен, в котором выполняется вызывающий поток  
    AppDomain adCallingThreadDomain = Thread.GetDomain();  
  
    // Каждому домену присваивается значимое имя, облегчающее отладку  
    // Получаем имя домена и выводим его  
    String callingDomainName = adCallingThreadDomain.FriendlyName;  
    Console.WriteLine(  
        "Default AppDomain's friendly name={0}", callingDomainName);  
  
    // Получаем и выводим сборку в домене, содержащем метод Main.  
    String exeAssembly = Assembly.GetEntryAssembly().FullName;  
    Console.WriteLine("Main assembly={0}", exeAssembly);  
  
    // Определяем локальную переменную, ссылающуюся на домен  
    AppDomain ad2 = null;  
  
    // ПРИМЕР 1. Доступ к объектам другого домена приложений  
    // с продвижением по ссылке  
    Console.WriteLine("{0}Demo #1". Environment.NewLine);
```

```

// Создаем новый домен (с теми же параметрами защиты и конфигурирования)
ad2 = AppDomain.CreateDomain("AD #2", null, null);
MarshalByRefType mbrt = null;

// Загружаем нашу сборку в новый домен, конструируем объект
// и продвигаем его обратно в наш домен
// (в действительности мы получаем ссылку на представитель)
mbrt = (MarshalByRefType)
    ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

Console.WriteLine("Type={0}", mbrt.GetType()); // CLR неверно
                                                // определяет тип

// Убеждаемся, что получили ссылку на объект-представитель
Console.WriteLine(
    "Is proxy={0}", RemotingServices.IsTransparentProxy(mbrt));

// Все выглядит так, как будто мы вызываем метод экземпляра
// MarshalByRefType, но на самом деле мы вызываем метод типа
// представителя. Именно представитель переносит поток в тот домен,
// в котором находится объект, и вызывает метод для реального объекта
mbrt.SomeMethod();

// Выгружаем новый домен
AppDomain.Unload(ad2);
// mbrt ссылается на правильный объект-представитель;
// объект-представитель ссылается на неправильный домен

try {
    // Вызываем метод, определенный в типе представителя
    // Поскольку домен приложений неправильный, вбрасывается исключение
    mbrt.SomeMethod();
    Console.WriteLine("Successful call.");
}
catch (AppDomainUnloadedException) {
    Console.WriteLine("Failed call.");
}

// ПРИМЕР 2. Доступ к объектам другого домена
// с продвижением по значению
Console.WriteLine("{0}Demo #2", Environment.NewLine);

// Создаем новый домен (с такими же параметрами защиты
// и конфигурирования, как в текущем)
ad2 = AppDomain.CreateDomain("AD #2", null, null);

// Загружаем нашу сборку в новый домен, конструируем объект
// и продвигаем его обратно в наш домен

```

```
// (в действительности мы получаем ссылку на представитель)
mbrt = (MarshalByRefType)
    ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

// Метод возвращает КОПИЮ возвращенного объекта;
// продвижение объекта происходило по значению, а не по ссылке
MarshalByValType mbvt = mbrt.MethodWithReturn();

// Убеждаемся, что мы НЕ получили ссылку на объект-представитель
Console.WriteLine(
    "Is proxy={0}", RemotingServices.IsTransparentProxy(mbvt));

// Кажется, что мы вызываем метод экземпляра MarshalByRefType,
// и это на самом деле так
Console.WriteLine("Returned object created " + mbvt.ToString());

// Выгружаем новый домен
AppDomain.Unload(ad2);
// mbrt ссылается на действительный объект;
// выгрузка домена не имеет никакого эффекта

try {
    // Вызываем метод объекта; исключение не вбрасывается
    Console.WriteLine("Returned object created " + mbvt.ToString());
    Console.WriteLine("Successful call.");
}
catch (AppDomainUnloadedException) {
    Console.WriteLine("Failed call.");
}

// ПРИМЕР 3. Доступ к объектам другого домена
// без использования механизма продвижения
Console.WriteLine("{0}Demo #3", Environment.NewLine);

// Создаем новый домен (с такими же параметрами защиты
// и конфигурирования, как в текущем)
ad2 = AppDomain.CreateDomain("AD #2", null, null);

// Загружаем нашу сборку в новый домен, конструируем объект
// и продвигаем его обратно в наш домен
// (в действительности мы получаем ссылку на представитель)
mbrt = (MarshalByRefType)
    ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");

// Метод возвращает объект, продвижение которого невозможно
// Вбрасывается исключение
```

```

NonMarshalableType nmt = mbrt.MethodArgAndReturn(callingDomainName);
// До выполнения этого кода дело не дойдет...
}

// Экземпляры допускают продвижение по ссылке через границы доменов
public sealed class MarshalByRefType : MarshalByRefObject {
    public MarshalByRefType() {
        Console.WriteLine("{0} ctor running in {1}",
            this.GetType().ToString(), Thread.GetDomain().FriendlyName);
    }

    public void SomeMethod() {
        Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName)
    }

    public MarshalByValType MethodWithReturn() {
        Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName)
        MarshalByValType t = new MarshalByValType();
        return t;
    }

    public NonMarshalableType MethodArgAndReturn(String callingDomainName)
        // ПРИМЕЧАНИЕ: callingDomainName имеет атрибут [Serializable]
        Console.WriteLine("Calling from '{0}' to '{1}'.",
            callingDomainName, Thread.GetDomain().FriendlyName);
        NonMarshalableType t = new NonMarshalableType();
        return t;
    }
}

// Экземпляры допускают продвижение по значению через границы доменов
[Serializable]
public sealed class MarshalByValType : Object {
    private DateTime m_creationTime = DateTime.Now;
    // ПРИМЕЧАНИЕ: DateTime помечен атрибутом [Serializable]

    public MarshalByValType() {
        Console.WriteLine("{0} ctor running in {1}. Created on {2:D}",
            this.GetType().ToString(),
            Thread.GetDomain().FriendlyName,
            m_creationTime);
    }

    public override String ToString() {
        return m_creationTime.ToString();
    }
}

```

```
// Экземпляры не допускают продвижение между доменами
// [Serializable]
public sealed class NonMarshalableType : Object {
    public NonMarshalableType() {
        Console.WriteLine("Executing in " + Thread.GetDomain().FriendlyName);
    }
}
```

Собрав и выполнив это приложение, мы получим следующее:

```
Default AppDomain's friendly name= Ch22-1-AppDomains.exe
Main assembly=Ch22-1-AppDomains, Version=0.0.0.0,
Culture=neutral, PublicKeyToken=null
```

```
Demo #1
MarshalByRefType ctor running in AD #2
Type=MarshalByRefType
Is proxy=True
Executing in AD #2
Failed call.
```

```
Demo #2
MarshalByRefType ctor running in AD #2
Executing in AD #2
MarshalByValType ctor running in AD #2, Created on Friday, August 07, 2009
Is proxy=False
Returned object created Friday, August 07, 2009
Returned object created Friday, August 07, 2009
Successful call.
```

```
Demo #3
MarshalByRefType ctor running in AD #2
Calling from 'Ch22-1-AppDomains.exe' to 'AD #2'.
Executing in AD #2
Unhandled Exception: System.Runtime.Serialization.SerializationException:
Type 'NonMarshalableType' in assembly 'Ch22-1-AppDomains, Version=0.0.0.0,
Culture=neutral, PublicKeyToken=null' is not marked as serializable.
at MarshalByRefType.MethodArgAndReturn(String callingDomainName)
at Program.Marshalling()
at Program.Main()
is not marked as serializable.
at MarshalByRefType.MethodArgAndReturn(String callingDomainName)
at Program.Marshalling()
at Program.Main()
```

А теперь поговорим о том, что делает этот код и как работает CLR.

В методе Marshalling я первым делом получаю ссылку на объект AppDomain, который идентифицирует домен приложений, где в данный момент выпол-

няется вызывающий поток. В Windows поток всегда создается в контексте одного процесса и проводит в нем всю свою жизнь. Однако между потоками и доменами приложений отсутствует однозначное соответствие. Домены приложений являются порождением CLR, Windows о них ничего не «знает». Так как в одном Windows-процессе может существовать несколько доменов приложений, поток может в разное время выполнять код разных доменов. С точки зрения CLR в каждый момент времени поток выполняет код только в одном из доменов приложений. Поток может запросить у CLR, код какого домена в нем выполняется в текущий момент, вызвав статический метод `GetDomain` класса `System.Threading.Thread` или запросив статическое, предназначенное только для чтения свойство `CurrentDomain` класса `System.AppDomain`.

Создаваемому домену можно присвоить значимое имя — строку типа `String`, используемую затем для идентификации. Обычно это оказывается полезным при отладке. Так как среда CLR создает основной домен до выполнения какого-либо кода, в качестве имени по умолчанию берется имя исполняемого файла. Мой метод `Marshalling` запрашивает имя основного домена через предназначенное только для чтения свойство `FriendlyName` класса `System.AppDomain`.

Далее, метод `Marshalling` запрашивает строгое имя сборки (загруженной в основной домен), которое определяет точку входа в метод `Main`, вызывающий мой метод `Marshalling`. В этой сборке определено несколько типов: `Program`, `MarshalByRefType`, `MarshalByValType` и `NonMarshalableType`. Теперь рассмотрим три во многом сходных друг с другом примера.

Пример 1. Доступ с продвижением по ссылке

В первом примере статический метод `CreateDomain` типа `System.AppDomain` вызывается, чтобы заставить CLR создать новый домен приложений в том же Windows-процессе. Тип `AppDomain` поддерживает несколько перегруженных версий метода `CreateDomain`; я рекомендую вам изучить их и выбрать версию, которая больше всего подойдет вам при написании кода создания нового домена. Моя версия этого метода имеет три аргумента:

- ❑ Строка `String` содержит значимое имя для нового домена. Я передаю ей значение `"AD #2"`.
- ❑ Аргумент `System.Security.Policy.Evidence` содержит политику, которую должна использовать среда CLR для вычисления набора разрешений для домена. В этом аргументе я передаю значение `null`, чтобы новый домен приложений наследовал тот же набор разрешений, что и родительский. Обычно для создания защитной границы вокруг кода домена приложений конструируют объект `System.Security.PermissionSet`, создают в нем необходимые объекты разрешений (экземпляры типов, которые реализуют интерфейс `IPermission`), а затем передают ссылку на результирующий объект `PermissionSet` перегруженной версии метода `CreateDomain`, принимающего `PermissionSet`.
- ❑ Аргумент `System.AppDomainSetup` задает параметры конфигурирования, которые среда CLR должна применить к новому домену. И здесь я передаю

значение `null`, чтобы новый домен приложения наследовал конфигурацию родительского. Чтобы домен получил особую конфигурацию, надо создать объект `AppDomainSetup`, задать его свойства требуемым образом, а затем передать в метод `CreateDomain` ссылку на результирующий объект `AppDomainSetup`.

Код метода `CreateDomain` создает новый домен в процессе. Этому домену присваивается значимое имя, а также параметры защиты и конфигурирования. У нового домена есть собственная куча загрузчика, которая пока пуста, потому что в него не загружено ни одной сборки. При создании домена среда CLR не создает в нем никаких потоков; там не выполняется никакой код, пока вы явно не заставите поток вызвать код домена приложений.

Теперь, чтобы получить экземпляр объекта в новом домене, надо сначала загрузить туда сборку, а затем создать экземпляр определенного в этой сборке типа. Именно эти операции и выполняет открытый экземплярный метод `CreateInstanceAndUnwrap` класса `AppDomain`. Этому методу передаются два параметра: строка `String`, идентифицирующая сборку, которую следует загрузить в новый домен (на нее ссылается переменная `ad2`), и строка `String`, содержащая имя типа, экземпляр которого надо создать. Метод `CreateInstanceAndUnwrap` заставляет вызывающий поток перейти из текущего домена в новый. Теперь поток (который выполняет вызов `CreateInstanceAndUnwrap`) загружает указанную сборку в новый домен, а затем просматривает таблицу метаданных с определениями типов сборки в поисках указанного типа (`MarshalByRefType`). Обнаружив нужный тип, поток вызывает конструктор `MarshalByRefType` без параметров и возвращается обратно в основной домен, чтобы метод `CreateInstanceAndUnwrap` мог вернуть ссылку на новый объект `MarshalByRefType`.

ПРИМЕЧАНИЕ

Существуют перегруженные версии `CreateInstanceAndUnwrap`, которые позволяют вызывать конструктор типа, передавая в него аргументы.

Однако все не так радужно, как могло бы показаться. Ведь CLR не позволяет переменной (корню), находящейся в одном домене, ссылаться на объект из другого. Если бы метод `CreateInstanceAndUnwrap` просто вернул ссылку на объект, это нарушило бы изоляцию, а ведь именно ради нее создавались домены! Поэтому непосредственно перед возвращением ссылки на объект метод `CreateInstanceAndUnwrap` выполняет некоторые дополнительные операции.

Обратите внимание, что тип `MarshalByRefType` наследует от специального базового класса `System.MarshalByRefObject`. Обнаружив, что метод `CreateInstanceAndUnwrap` выполняет продвижение объекта типа, производного от `MarshalByRefObject`, CLR выполняет продвижение объекта по ссылке в другой домен. Поясню, что означает продвижение объекта по ссылке из одного домена (в котором объект был создан) в другой (в котором вызывается метод `CreateInstanceAndUnwrap`).

Когда домену-источнику нужно передать ссылку на объект в целевой домен приложений или вернуть ее обратно, CLR определяет в куче загрузчика этого домена тип *представителя* (проху). Этот тип определяется посредством метаданных исходного типа, представителем которого он является, поэтому выглядит он в точности как исходный тип — у него совпадают все экземплярные члены (свойства, события и методы). Экземплярные поля к типу не относятся, но об этом мы поговорим чуть позже. В новом типе действительно определяются некоторые экземплярные поля, но они не идентичны полям исходного типа данных. Вместо этого эти поля указывают, который из доменов «владеет» реальным объектом и как найти этот объект в домене-владельце. (Внутренне объект-представитель использует экземпляр `GCHandle`, который ссылается на реальный объект. Тип `GCHandle` обсуждается в главе 21.)

После определения этого типа в целевом домене метод `CreateInstanceAndUnwrap` создает экземпляр типа представителя, инициализирует его поля таким образом, чтобы они указывали на домен-источник и реальный объект, и возвращает в целевой домен ссылку на объект-представитель. В моем варианте кода на этот представитель ссылается переменная `mbrt`. При этом возвращенный методом `CreateInstanceAndUnwrap` объект не является экземпляром типа `MarshalByRefType`. Обычно CLR не разрешает приведение к несовместимым типам. Однако в этой ситуации сделано исключение, потому что члены экземпляров нового и исходного типов совпадают. В сущности, если вы используете объект-представитель для вызова метода `GetType`, представитель вводит вас в заблуждение, представляясь объектом `MarshalByRefType`.

Тем не менее есть возможность узнать, что объект, возвращенный методом `CreateInstanceAndUnwrap`, в реальности является ссылкой на объект-представитель. Для этого мы используем открытый статический метод `IsTransparentProxy` типа `System.Runtime.Remoting.RemotingService`, которому в качестве параметра передается ссылка, возвращенная методом `CreateInstanceAndUnwrap`. Если метод `IsTransparentProxy` возвращает значение `true`, значит, объект является представителем.

Итак, мое приложение использует представитель для вызова метода `SomeMethod`. Так как переменная `mbrt` ссылается на объект-представитель, вызывается реализация этого метода в представителе. В ней задействуются информационные поля объекта-представителя для направления вызывающего потока из основного домена приложения в новый. Теперь любые действия этого потока выполняются в контексте безопасности и конфигурации нового домена. Далее поток использует поле `GCHandle` объекта-представителя для поиска в новом домене реального объекта, после чего вызывает для него метод `SomeMethod`.

Есть два способа убедиться, что запрашивающий поток перешел от основного к новому домену. Во-первых, в методе `SomeMethod` я вызываю метод `Thread.GetDomain().FriendlyName`. В результате возвращается AD #2 (что подтверждается выходными данными), так как поток теперь выполняется в новом домене, соз-

данном методом `AppDomain.CreateDomain` с параметром AD #2 в качестве значимого имени. Во-вторых, при пошаговом выполнении кода в отладчике с открытым окном Call Stack строка `[AppDomain Transition]` отмечает переход потока через границу между доменами (рис. 22.2).

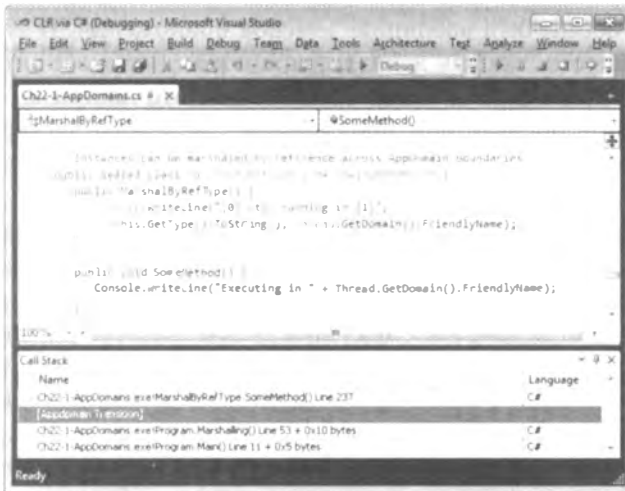


Рис. 22.2. Переход между доменами в окне отладчика Call Stack

Реальный метод `SomeMethod` возвращает управление методу `SomeMethod` представителя, тот переправляет поток обратно в основной домен, в котором и продолжается выполнение кода.

ПРИМЕЧАНИЕ

Когда поток в одном домене вызывает метод другого домена приложений, поток переходит от домена к домену. Это означает, что вызовы метода через границу между доменами приложения выполняются синхронно. Однако считается, что в каждый момент времени поток находится только в одном домене. Для выполнения кода в нескольких доменах приложений одновременно придется создавать дополнительные потоки и заставлять их выполнять нужный код в нужных доменах.

Далее мое приложение вызывает открытый статический метод `Unload` типа `AppDomain`, чтобы заставить CLR выгрузить указанный домен вместе со всеми загруженными в него сборками. Одновременно иницируется сборка мусора для освобождения всех объектов выгружаемого домена. На этом этапе переменная `mbt` основного домена все еще ссылается на нужный объект-представитель; однако сам объект-представитель больше не ссылается на нужный домен приложений (поскольку тот уже выгружен).

Когда основной домен пытается использовать объект-представитель для вызова метода `SomeMethod`, вызывается реализация этого метода, определенная

в представителе. Реализация представителя выясняет, что домен приложений, в котором находился реальный объект, уже выгружен, и метод `SomeMethod` вызывает исключение `AppDomainUnloadedException`, информируя вызывающий код о невозможности выполнения операции.

Как видите, команда разработчиков CLR в Microsoft проделала огромную работу, чтобы обеспечить изоляцию доменов приложений. И это дело исключительной важности, так как данная функциональность все активнее используется при решении повседневных задач. Ясно, что доступ к объектам через границы доменов посредством продвижения по ссылке связан с некоторой потерей производительности, поэтому использование таких операций надо сводить к минимуму.

Я обещал рассказать об экземплярных полях. Они определяются в типе, производном от `MarshalByRefObject`. Однако они определяются не как часть типа представителя и отсутствуют в объекте-представителе. Создавая код, который считывает и изменяет значения полей экземпляров типа, производного от `MarshalByRefObject`, JIT-компилятор генерирует код, использующий объект-представитель (чтобы найти реальные домен и объект), вызывая соответственно метод `FieldGetter` или `FieldSetter` класса `System.Object`. Это закрытые и недокументированные методы; в сущности, для считывания и записи значений в поле в них используется отражение. Так что хотя вы и имеете доступ к полям типа, производного от `MarshalByRefObject`, производительность от этого страдает особенно сильно, потому что для такого доступа среде CLR приходится вызывать методы. Производительность значительно снижается, даже если объект, доступ к которому осуществляется, находится в локальном домене приложений¹.

ПРИМЕЧАНИЕ

Чтобы продемонстрировать, сколь значительной может быть потеря производительности, я написал следующий код:

```
private sealed class NonMBRO : Object { public Int32 x; }
private sealed class MBRO : MarshalByRefObject { public Int32 x; }
private static void FieldAccessTiming(){
    const Int32 count = 10000000;
    NonMBRO nonMbro = new NonMBRO();
    MBRO mbro = new MBRO();
    Int64 time = Stopwatch.GetTimestamp();
    for (Int32 c = 0; c < count; c++) nonMbro.x++;
    Console.WriteLine(
        "{0:N0}", Stopwatch.GetTimestamp() - time); // 134 174
```

¹ Если бы среда CLR требовала закрытости всех полей (что рекомендуется для хорошей инкапсуляции), методы `FieldGetter` и `FieldSetter` не должны были бы существовать. В итоге к полям остался бы только непосредственный доступ из методов, что избавило бы нас от потерь производительности.

```
time = Stopwatch.GetTimestamp();
for (Int32 c = 0; c < count; c++) mbro.x++;
Console.WriteLine(
    "{0:N0}". Stopwatch.GetTimestamp() - time); // 1 533 886
}
```

Доступ к экземпляру поля класса NonMBRO, производного от класса Object, занял 134 174 миллисекунд, в то время как для доступа к классу MBRO, производному от MarshalByRefObject, потребовалось 1 533 886 миллисекунд. Как видите, во втором случае процесс занял в 12 раз больше времени.

Наконец, с точки зрения удобства использования в производном от MarshalByRefObject типе не следует определять какие-либо статические члены. Дело в том, что к статическим членам всегда обращаются в контексте вызывающего домена приложений. Никакой передачи между доменами быть не может, поскольку информация о целевом домене содержится в объекте-представителе, но такой объект при вызове статического члена попросту отсутствует. Модель программирования, предусматривающая выполнение статических членов типа в одном домене, в то время как экземплярные члены выполняются в другом, была бы очень неудачной.

Так как во втором домене отсутствуют корни, исходный объект, на который ссылался представитель, может быть отправлен в мусор. Разумеется, это — не идеальный подход. Однако если бесконечно хранить исходный объект в памяти, он останется там даже после удаления представителя, что тоже не очень хорошо. В CLR эта проблема решается при помощи *диспетчера аренды* (lease manager). Создав для объекта представителя, CLR сохраняет объект в течение 5 минут. Если за это время через представителя не последовало ни одного вызова, объект деактивируется (deactivated) и освобождает память при следующей сборке мусора. После каждого вызова объекта диспетчер обновляет его срок аренды, в результате объект гарантированно остается в памяти еще 2 минуты и только потом деактивируется. При попытке вызвать через представителя объект с истекшим сроком аренды CLR вбрасывает исключение System.Runtime.Remoting.RemotingException.

Для переопределения заданного по умолчанию времени в 5 и 2 минуты используйте виртуальный метод InitializeLifetimeServices типа MarshalByRefObject. Дополнительную информацию по данной теме вы можете найти в SDK-документации на .NET Framework.

Пример 2. Доступ с продвижением по значению

Этот пример похож на предыдущий. Точно так же создается второй домен приложений. Затем вызывается метод CreateInstanceAndUnwrap для загрузки той же сборки в новый домен и создания в нем экземпляра объекта MarshalByRefType. Далее CLR создает для объекта представителя, и переменной mbro (в основном домене) присваивается ссылка на него. Теперь при помощи созданного пред-

ставителя я вызываю метод `MethodWithReturn`. Этот метод без параметров будет выполнен в новом домене, а перед тем как вернуть ссылку на объект основному домену, он создаст экземпляр типа `MarshalByValType`.

Тип `MarshalByValType` не является производным от `System.MarshalByRefObject`, а значит, CLR не может определить тип представителя для создания его экземпляра; то есть объект нельзя продвинуть по ссылке через границу домена.

Однако благодаря наличию у типа `MarshalByValType` настраиваемого атрибута `[Serializable]` метод `MethodWithReturn` может выполнить продвижение объекта по значению. Сейчас мы поговорим о том, что происходит при продвижении объекта по значению из одного домена (исходного) в другой (целевой). А дополнительную информацию о механизмах сериализации и десериализации вы найдете в главе 24.

Когда исходному домену нужно передать ссылку на объект в целевой домен (или возвратить ссылку из целевого домена), CLR сериализует экземплярные поля объекта в байтовый массив, который затем копируется. После этого CLR десериализует байтовый массив в целевом домене. Это вынуждает CLR загрузить в целевой массив сборку (если она еще не загружена), в которой определен десериализованный тип. Далее CLR создает экземпляр типа и использует значения из байтового массива для инициализации полей объекта так, чтобы они полностью совпадали со значениями исходного объекта. Иначе говоря, CLR делает точную копию исходного объекта в целевом домене. Затем метод `MethodWithReturn` возвращает ссылку на эту копию; в результате наш объект оказывается продвинутым по значению через границу домена.

ВНИМАНИЕ

При загрузке сборки CLR использует политики и конфигурацию целевого домена (в частности, у домена приложений может быть другой каталог `AppBase` или информация о перенаправлении версий). Эти различия в политиках могут стать причиной, по которой CLR затрудняется определить местонахождение сборки. Если сборку загрузить не удастся, возникает исключение, а целевой домен приложений не получает ссылку на объект.

На этом этапе объекты в исходном и целевом доменах существуют независимо друг от друга, поэтому их состояния также могут меняться независимо. Если в исходном домене нет корней, предохраняющих исходный объект от уничтожения сборщиком мусора (как в созданном мною приложении), его память будет освобождена при следующей сборке.

Чтобы убедиться, что объект, возвращенный методом `MethodWithReturn`, не является ссылкой на объект-представитель, мое приложение вызывает открытый статический метод `IsTransparentProxy` типа `System.Runtime.Remoting.RemotingService`, передавая ему в качестве параметра ссылку, возвращенную методом `MethodWithReturn`. Как видно из результатов работы программы, метод

IsTransparentProxy возвращает значение `false`, означающее, что объект является реальным объектом, не представителем.

Итак, моя программа использует реальный объект для вызова метода `ToString`. Так как переменная `mbvt` ссылается на реальный объект, вызывается реальная реализация этого метода и никаких переходов между доменами приложений не происходит. Это легко проверить, проанализировав информацию в окне **Call Stack** отладчика: строка `[AppDomain Transition]` там не появится.

Для дополнительного доказательства того, что это не представитель, мое приложение выгружает новый домен, после чего пытается снова вызвать метод `ToString`. В отличие от примера 1 на сей раз запрос успешно выполняется, потому что выгрузка нового домена никак не влияет на объекты, расположенные в основном домене, в том числе на объект, продвинутый по значению.

Пример 3. Доступ без продвижения

Этот пример очень похож на описанные ранее примеры 1 и 2. Точно так же создается новый домен, после чего вызывается метод `CreateInstanceAndUnwrap` для загрузки той же сборки в новый домен приложений и создания в нем объекта `MarshalByValType`. Переменная `mbvt` ссылается на представитель этого объекта.

Затем через этот представитель я вызываю метод `MethodArgAndReturn`, принимающий один аргумент. Так как среда CLR должна контролировать изоляцию домена, она не может просто передать ссылку на аргумент в новый домен приложений. Для объекта, принадлежащего к типу, производному от `MarshalByRefObject`, CLR создает представитель и выполняет продвижение объекта по ссылке. Если тип объекта помечен атрибутом `[Serializable]`, CLR сериализует объект (и его потомков) в байтовый массив, пакует этот массив в новый домен и десериализует его в граф объекта, передав корень графа методу `MethodArgAndReturn`.

В этом примере я передаю объект `System.String` через границы домена. Тип `System.String` не является производным от класса `MarshalByRefObject`, а значит, CLR не может создать представитель. К счастью, объект `System.String` помечен атрибутом `[Serializable]`, поэтому CLR в состоянии продвинуть его по значению, и код будет работать. Обратите внимание, что для типа `String` CLR выполняет специальную оптимизацию. Продвигая объект `String` через границу домена, CLR просто передает ссылку на него; копию объекта она не делает. Подобная оптимизация оказывается возможной благодаря неизменности объектов типа `String`, что не дает коду из одного домена повредить символы объекта `String` из другого. Дополнительную информацию о неизменности объектов данного типа вы найдете в главе 14¹.

¹ Кстати, именно поэтому класс `System.String` является изолированным. Если бы это было не так, вы могли бы определять собственные классы, производные от `String`, и добавлять к ним собственные поля. В результате среда CLR не смогла бы гарантировать неизменность строк.

Внутри метода `MethodArgAndReturn` я вывожу передаваемую в него строку, чтобы показать ее переход через границу домена. Затем я создаю экземпляр типа `NonMarshalableType` и возвращаю ссылку на этот объект в основной домен. Так как тип `NonMarshalableType` не является производным от `System.MarshalByRefObject` и не помечен атрибутом `[Serializable]`, метод `MethodArgAndReturn` не может продвинуть объект по ссылке или по значению. То есть у нас нет способа передать объект через границы домена. Чтобы указать на этот факт, метод `MethodArgAndReturn` вбрасывает в основном домене исключение `SerializationException`. Так как моя программа не умеет его перехватывать, она просто прекращает свою работу.

Выгрузка доменов

Одна из замечательных особенностей CLR — возможность выгрузки доменов приложений. При этом выгружаются и все загруженные в них сборки, а также освобождается куча загрузчика доменов. Провести эту процедуру легко: достаточно вызвать статический метод `Unload` класса `AppDomain` (как показано в моем приложении в начале главы). Это заставляет CLR выполнить набор операций по корректной выгрузке указанного домена.

1. CLR приостанавливает все потоки в процессе, которые когда-либо выполняли управляемый код.
2. CLR проверяет все стеки на наличие потоков, которые в текущий момент выполняют код выгружаемого домена или могут рано или поздно вернуться к выполнению такого кода. CLR вынуждает все потоки, в стеке которых находится выгружаемый домен, вбросить исключение `ThreadAbortException` (при этом выполнение потока возобновляется). В результате потоки переходят к выполнению блоков `finally`, то есть корректно завершают свою работу. При отсутствии кода, перехватывающего исключение `ThreadAbortException`, оно переходит в разряд необработанных и «проглатывается» CLR; поток завершается, но процессу разрешается продолжить работу. Такое поведение отличается от стандартного, потому что в любых других ситуациях при возникновении необработанного исключения CLR уничтожает процесс.

ВНИМАНИЕ

CLR не уничтожит немедленно поток, который выполняет код блока `finally` или `catch`, конструктора класса, критической области или неуправляемый код. Уничтожение таких потоков сделало бы невозможным выполнение кода очистки, восстановления после ошибок, инициализации типа, критического кода или любого кода, который CLR не знает как обрабатывать. Это стало бы причиной непредсказуемого поведения приложений и появлению дыр в системе безопасности. Уничтожаемому потоку разрешается закончить выполнение таких блоков, и только после этого CLR вынуждает его вбросить исключение `ThreadAbortException`.

3. После выгрузки из домена всех потоков, обнаруженных на втором шаге, CLR проходит по куче и устанавливает флаг для каждого объекта-представителя, который ссылается на объект, созданный в выгружаемом домене. Так объекты-представители «узнают», что реальный объект, на который они ссылаются, уничтожен. В результате при попытке вызвать метод «неправильного» объекта-представителя вбрасывается исключение `AppDomainUnloadedException`.
4. CLR инициирует принудительную сборку мусора, чтобы освободить память, занятую объектами выгружаемого домена. Вызываются методы финализации для этих объектов, давая им шанс выполнить нужную очистку.
5. CLR возобновляет работу всех оставшихся потоков. Поток, вызвавший метод `AppDomain.Unload`, продолжает работу; вызовы `AppDomain.Unload` выполняются синхронно.

В моем приложении всю работу выполняет один поток. Всякий раз, когда код приложения вызывает метод `AppDomain.Unload`, в выгружаемом домене не оказывается потоков, поэтому CLR не приходится вбрасывать исключение `ThreadAbortException` (о нем мы поговорим чуть позже).

Кстати, при вызове потоком метода `AppDomain.Unload` CLR ждет 10 секунд, чтобы потоки выгружаемого домена могли его покинуть. Если после этого поток, вызвавший метод `AppDomain.Unload`, не возвращает управление, он вбрасывает исключение `CannotUnloadAppDomainException`, и домен может быть (а может и не быть) выгруженным в будущем.

ПРИМЕЧАНИЕ

Если поток, вызвавший `AppDomain.Unload`, находится в выгружаемом домене, CLR создает другой поток, который пытается выгрузить домен. Первый поток принудительно вбрасывает исключение `ThreadAbortException` и выполняет раскрутку — поиск и очистку всех операций, начатых ниже по стеку вызовов. Новый поток дожидается выгрузки домена, а затем завершается. В случае сбоя выгрузки новый поток попытается обработать исключение `CannotUnloadAppDomainException`, но так как нет кода, выполняемого этим новым потоком, перехватить это исключение нельзя.

Мониторинг доменов

Хост-приложение умеет отслеживать потребляемые доменом ресурсы. Некоторые хосты на основе этой информации принудительно выгружают домен, если вдруг потребление им памяти или ресурсов процессора выходит за разумные с точки зрения хоста пределы. Мониторинг позволяет также сравнить потребление ресурсов различными алгоритмами и сделать выбор. Но так как эта операция влияет на производительность, она инициируется вручную путем

присвоения статическому свойству `MonitoringEnabled` класса `AppDomain` значения `true`. При этом включается мониторинг всех доменов. Выключить его уже нельзя; попытка присвоить свойству `MonitoringEnabled` значение `false` приведет к исключению `ArgumentException`.

При включенном мониторинге код может использовать четыре предназначенных только для чтения свойства класса `AppDomain`:

- ❑ **`MonitoringSurvivedProcessMemorySize`**. Это статическое свойство типа `Int64` возвращает число байтов, используемых в данный момент всеми доменами под управлением текущего экземпляра CLR. Значение верно с момента последней сборки мусора.
- ❑ **`MonitoringTotalAllocatedMemorySize`**. Это свойство экземпляра типа `Int64` возвращает количество байтов, выделенных определенным доменом. Значение верно с момента последней сборки мусора.
- ❑ **`MonitoringSurvivedMemorySize`**. Это свойство экземпляра типа `Int64` возвращает количество байтов, которые в настоящее время используются определенным доменом. Значение верно с момента последней сборки мусора.
- ❑ **`MonitoringTotalProcessorTime`**. Это свойство экземпляра типа `TimeSpan` возвращает процессорное время, использованное определенным доменом.

Следующий класс демонстрирует, как при помощи трех из этих свойств узнать об изменении состояния домена за некий промежуток времени:

```
private sealed class AppDomainMonitorDelta : IDisposable {
    private AppDomain m_appDomain;
    private TimeSpan m_thisADCpu;
    private Int64 m_thisADMemoryInUse;
    private Int64 m_thisADMemoryAllocated;

    static AppDomainMonitorDelta() {
        // Проверяем, что включен режим мониторинга домена
        AppDomain.MonitoringIsEnabled = true;
    }

    public AppDomainMonitorDelta(AppDomain ad) {
        m_appDomain = ad ?? AppDomain.CurrentDomain;
        m_thisADCpu = m_appDomain.MonitoringTotalProcessorTime;
        m_thisADMemoryInUse = m_appDomain.MonitoringSurvivedMemorySize;
        m_thisADMemoryAllocated =
            m_appDomain.MonitoringTotalAllocatedMemorySize;
    }

    public void Dispose() {
        GC.Collect();
        Console.WriteLine("FriendlyName={0}, CPU={1}ms",
```

```

        m_appDomain.FriendlyName.
        (m_appDomain.MonitoringTotalProcessorTime -
        m_thisADCpu).TotalMilliseconds);
Console.WriteLine(
    " Allocated {0:N0} bytes of which {1:N0} survived GCs".
    m_appDomain.MonitoringTotalAllocatedMemorySize -
    m_thisADMemoryAllocated,
    m_appDomain.MonitoringSurvivedMemorySize - m_thisADMemoryInUse);
    }
}

```

А это — пример применения класса AppDomainMonitorDelta:

```

private static void AppDomainResourceMonitoring() {
    using (new AppDomainMonitorDelta(null)) {

        // Выделено около 10 миллионов байтов.
        // которые переживут сборку мусора
        var list = new List<Object>();
        for (Int32 x = 0; x < 1000; x++) list.Add(new Byte[10000]);

        // Выделено около 20 миллионов байтов.
        // которые НЕ переживут сборку мусора
        for (Int32 x = 0; x < 2000; x++) new Byte[10000].GetType();

        // Раскручиваем процессор около 5 секунд
        Int64 stop = Environment.TickCount + 5000;
        while (Environment.TickCount < stop) ;
    }
}

```

Выполнив этот код, мы получим:

```

FriendlyName=03-Ch22-1-AppDomains.exe, CPU=5031.25ms
Allocated 30,159,496 bytes of which 10,085,080 survived GCs

```

Уведомление о первом управляемом исключении домена

С каждым доменом связан набор методов обратного вызова, активизирующихся, когда CLR начинает искать внутри домена блоки catch. Эти методы могут выполнять записи в журнал, кроме того, хост в состоянии использовать этот механизм для отслеживания сгенерированных внутри домена исключений. Обратные вызовы не могут обработать исключение или «поглотить» его; они просто получают уведомление. Для регистрации такого метода обратного вызова достаточно добавить делегат к экземпляру события FirstChanceException класса AppDomain.

Среда CLR в данном случае работает так: при первом вбрасывании исключения она задействует любой из методов обратного вызова `FirstChanceException`, зарегистрированный в домене, ставшем источником исключения. Затем CLR ищет в стеке этого домена блоки `catch`. Если какой-то из этих блоков обрабатывает исключение, выполнение программы возвращается в обычный режим. Если же такой блок отсутствует, CLR идет наверх стека вызывающего домена и снова вбрасывает то же самое исключение (после его сериализации и десериализации). Однако это исключение начинает восприниматься, как новое, и CLR задействует методы обратного вызова `FirstChanceException`, зарегистрированные на данный момент в текущем домене. Процесс продолжается, пока не будет достигнут верх стека потока. Если в результате исключение так и не удастся обработать, CLR завершает процесс.

Использование хостами доменов приложений

Я уже рассказывал о хостах: как они загружают CLR, как хост может заставить CLR создать или выгрузить домен приложений. Теперь, чтобы перевести разговор в более конкретное русло, мы рассмотрим несколько обычных сценариев, касающихся хостинга и доменов приложений. В частности, я расскажу, как разные типы приложений выполняют хостинг CLR и управляют доменами приложений.

Исполняемые приложения

Консольные UI-приложения, сервисные NT-приложения, приложения Windows Forms и приложения Windows Presentation Foundation (WPF) являются *саморазмещающимися* (self-hosted) и снабжены управляемыми EXE-файлами. Инициализировав процесс при помощи такого файла, Windows загружает согласователь, который исследует информацию в заголовке CLR, содержащуюся в сборке приложения (EXE-файле). Эта информация указывает версию CLR, которая использовалась при сборке и тестировании приложения. Именно с ее помощью согласователь определяет, какую версию CLR следует загрузить в процесс. Загрузив и инициализировав среду, согласователь снова исследует заголовок ее сборки, чтобы определить, какой метод является точкой входа приложения (`Main`). CLR вызывает этот метод, и приложение начинает работу.

К работающему коду возможен доступ из других типов. При ссылке на тип из другой сборки CLR локализует эту сборку и загружает ее в тот же домен. Туда же загружаются и все прочие сборки, на которые имеются ссылки. После возвращения управления методом `Main` Windows-процесс завершает свою работу (ликвидируя основной и все прочие домены).

ПРИМЕЧАНИЕ

Кстати, для завершения Windows-процесса со всеми его доменами можно воспользоваться статическим методом `Exit` класса `System.Environment`. Это самый корректный способ завершения процесса, так как данный метод сначала вызывает методы финализации всех объектов в управляемой куче, а затем освобождает все неуправляемые COM-объекты в CLR. После этого вызывается Win32-функция `ExitProcess`.

Приложение может заставить CLR создать дополнительные домены в адресном пространстве процесса. Собственно, именно это и происходит в моем приложении, код которого имеется в начале этой главы.

Полнофункциональные интернет-приложения Silverlight

Версия CLR для разработанной в Microsoft динамической технологии Silverlight отличается от версии обычной платформы .NET Framework для настольных компьютеров. После установки среды Silverlight перемещение на сайт, который использует эту среду, заставляет Silverlight CLR (`CoreClr.dll`) загрузить браузер (причем это может быть вовсе не Internet Explorer — вы можете вообще не использовать Windows). Каждый элемент управления Silverlight на странице работает в собственном домене. Когда пользователь закрывает вкладку или переходит на другой сайт, то элементы управления Silverlight перестают применяться и их домены выгружаются. Код Silverlight в домене запускается в *песочнице* (sandbox) с ограниченной защитой и не может причинить какого-либо вреда пользователю или машине.

Веб-формы ASP.NET и веб-сервисы XML

ASP.NET — это библиотека ISAPI (реализованная в файле `ASPNet_ISAPI.dll`). При первом запросе клиентом URL-адреса, обрабатываемого этой библиотекой, ASP.NET загружает CLR. Когда клиент запрашивает веб-приложение, ASP.NET определяет, были ли уже такие запросы. Если данный запрос является первым, CLR получает команду создать для данного веб-приложения новый домен (каждое приложение идентифицируется собственным виртуальным корневым каталогом). Далее ASP.NET заставляет CLR загрузить в новый домен сборку с типом, поддерживаемым этим веб-приложением, создает экземпляры этого типа и начинает вызывать его методы для исполнения запроса клиента. При наличии ссылок на другие типы CLR загружает в домен веб-приложения дополнительные сборки.

При поступлении запроса к уже работающему веб-приложению ASP.NET вместо нового домена создает новый экземпляр типа веб-приложения в существующем и начинает вызывать его методы. При этом вызываемые методы ока-

зываются уже преобразованными JIT-компилятором в машинный код, поэтому следующие клиентские запросы обрабатываются намного быстрее.

Если клиент запрашивает другое веб-приложение, ASP.NET заставляет CLR создать новый домен. Обычно он появляется в том же рабочем процессе, в котором работают другие домены приложений. Это значит, что в одном Windows-процессе может работать несколько веб-приложений, что повышает производительность системы. В этом случае сборки для разных веб-приложений загружаются в собственный домен каждого из них — это необходимо для изоляции кода и объектов веб-приложения от других веб-приложений.

Замечательная особенность ASP.NET — возможность изменять код веб-сайта без остановки веб-сервера. Когда файл на жестком диске сайта меняется, ASP.NET обнаруживает это, выгружает домен, содержащий старую версию (после завершения текущего запроса), а затем создает новый домен, загружая в него новые версии файлов. При этом ASP.NET использует особый механизм доменов, называемый *теньвым копированием* (shadow copying).

Microsoft SQL Server

Microsoft SQL Server относится к неуправляемым приложениям, так как большая часть кода SQL-сервера написана на неуправляемом языке C++. SQL-сервер поддерживает создание хранимых процедур на управляемом коде. При первом получении запроса на выполнение хранимой процедуры на управляемом коде SQL-сервер загружает CLR. Хранимые процедуры выполняются в собственном защищенном домене, что не позволяет им нарушить работу сервера базы данных. Это — совершенно замечательная функциональность! Ведь разработчики могут выбирать язык программирования для создания хранимых процедур. Кроме того, код компилируется JIT-компилятором в машинный код и выполняется, а не интерпретируется. Также разработчикам таких процедур доступны все типы, определенные в библиотеке FCL или любой другой сборке. В результате разработка хранимых процедур значительно упрощается, а приложения работают намного быстрее. Что еще нужно программисту для счастья?

Будущее и мечты

В будущем в обычных «офисных» приложениях, таких как редакторы и электронные таблицы, пользователи смогут выбирать язык программирования для создания макросов. Эти макросы обеспечат доступ к любым сборкам и типам, поддерживающим CLR. Они будут компилироваться и поэтому быстро выполняться, и, что самое важное, — выполняться в защищенном домене, избавляя пользователей от многих неприятных неожиданностей.

Нетривиальное управление хостингом

В этом разделе рассказывается о более сложных вопросах хостинга CLR. Я хочу показать всю широту возможностей CLR. Если тема этого раздела вас заинтересует, я настоятельно рекомендую обратиться к другой литературе по данной теме.

Применение управляемого кода

Класс `System.AppDomainManager` позволяет хосту менять заданное по умолчанию поведение CLR при помощи управляемого кода. При этом упрощается реализация хоста. Вам требуется только определить собственный класс, потомок класса `System.AppDomainManager`, переопределив все необходимые виртуальные методы. Далее этот класс надо скомпилировать в отдельную сборку и установить ее в глобальный кэш сборок (GAC), предоставив ей тем самым полное доверие.

Затем при запуске Windows-процесса нужно заставить CLR использовать свой производный от `AppDomainManager` класс. Это лучше всего сделать, создав объект `AppDomainSetup` и инициализировав его свойства `AppDomainManagerAssembly` и `AppDomainManagerType` типа `String`. Свойству `AppDomainManagerAssembly` присваивается строка со строгим именем сборки, определяющей ваш класс, производный от класса `AppDomainManager`. Свойству же `AppDomainManagerType` присваивается полное имя этого класса. Кроме того, свойству `AppDomainManager` с помощью элементов `appDomainManagerAssembly` и `appDomainManagerType` можно присвоить конфигурационный XML-файл вашего приложения. Также собственный хост может отправить запрос к интерфейсу `ICLRControl` и вызвать его свойство `SetAppDomainManagerType`, передав туда идентификатор установленной в GAC сборки и имя класса, производного от `AppDomainManager`¹.

Теперь поговорим о функциях класса, производного от `AppDomainManager`. Он позволяет хосту сохранить контроль, даже когда надстройка пытается создать собственный домен. При этом объект, производный от `AppDomainManager`, может редактировать параметры защиты и конфигурирования. Кроме того, он в состоянии помешать созданию нового домена или вернуть ссылку на уже существующий. Когда новый домен уже создан, CLR формирует в нем новый объект, производный от `AppDomainManager`. Он также может редактировать параметры конфигурирования, контекст выполнения между потоками и разрешения, предоставленные сборке.

¹ Конфигурацию класса `AppDomainManager` можно выполнить также при помощи переменных окружения и параметров реестра, но эти механизмы более громоздки и применять их имеет смысл разве что в некоторых тестовых сценариях.

Разработка надежных хост-приложений

Хост может указать CLR, какие действия предпринимать при сбое в управляемом коде. Вот несколько примеров (от наименее до наиболее серьезного):

- ❑ CLR может прервать поток, если тот выполняется слишком долго или долго не возвращает управление (детали см. в следующем разделе).
- ❑ CLR может выгрузить домен. При этом закроются все потоки этого домена, а проблемный код будет выгружен.
- ❑ CLR может отключиться. При этом прекращается выполнение любого кода в процессе, но неуправляемому коду работать разрешено.
- ❑ CLR может выйти из Windows-процесса. При этом сначала закрываются все потоки и выгружаются все домены — выполняются операции очистки, после чего процесс закрывается.

CLR может завершить поток или домен как корректно, так и принудительно. Корректное завершение предусматривает выполнение кода очистки. Иначе говоря, выполняется код в блоках `finally` и вызываются методы финализации объектов. При принудительном завершении код очистки игнорируется. При корректном завершении, в отличие от принудительного, не удастся закрыть поток в блоках `catch` и `finally`. Поток, выполняющий неуправляемый код или находящийся в критической области (Critical Execution Region, CER), завершить вообще нельзя.

Хост может установить так называемую *политику расширения* (escalation policy), определив тем самым поведение CLR при сбоях управляемого кода. Например, SQL-сервер определяет, что должна делать среда CLR при появлении необработанного исключения во время выполнения управляемого кода. Когда в потоке возникает необработанное исключение, CLR сначала пытается корректно завершить поток. Если поток не закрывается за определенное время, CLR пытается перейти от корректного к принудительному завершению потока.

В большинстве случаев происходит именно так. Однако для потоков из *критической области* (critical region) действует другая политика. Поток, находящийся в критической области, запирается в рамках синхронизации потоков, причем отпереть его должен тот же самый поток, например, тот, что вызвал метод `Monitor.Enter`, метод `WaitOne` типа `Mutex` или один из методов `AcquireReaderLock` или `AcquireWriterLock` типа `ReaderWriterLock`¹. Ожидание методов `AutoResetEvent`, `ManualResetEvent` или `Semaphore` не указывает на пребывание потока в критической области, потому что другой поток может освободить этот объект синхронизации. Когда поток находится в критической области, CLR

¹ При любом записании внутренне вызываются методы `BeginCriticalRegion` и `EndCriticalRegion` класса `Thread`, показывающие момент входа в критическую область и выхода из нее. При необходимости вы тоже можете ими воспользоваться. Такая необходимость обычно возникает при взаимодействии с неуправляемым кодом.

полагает, что он работает с данными, совместно используемыми несколькими потоками того же домена. Это и есть наиболее вероятная причина записывания потока. При работе с общими данными простое завершение потока является неудачным решением, потому что к данным могут обращаться другие потоки. Данные же уже повреждены. В итоге мы получаем непредсказуемое поведение домена и бреши в системе безопасности.

Поэтому, когда в потоке в критической области возникает необработанное исключение, CLR сначала пытается свести исключение к корректной выгрузке домена в попытке избавиться от всех используемых потоков и объектов данных. Если домен не выгружается за указанное время, CLR переходит от корректной к принудительной выгрузке.

Возвращение потока в хост

Обычно хост-приложение пытается контролировать собственные потоки. В качестве примера рассмотрим сервер базы данных. При поступлении запроса поток принимает его и пересылает другому потоку, который и должен сделать нужную работу. Может случиться, что второму потоку придется выполнить код, созданный и протестированный не командой разработчиков сервера, а сторонними программистами. Например, он может выполнить хранимую процедуру, написанную на управляемом коде. Что замечательно, сервер базы данных делает это в собственном домене приложений, работающем с максимальными ограничениями безопасности. В результате хранимая процедура не может обратиться к объектам за пределами собственного домена, а также получить доступ к ресурсам, к которым коду обращаться запрещено, например к дисковым файлам или буферу обмена.

Но что если код хранимой процедуры войдет в бесконечный цикл? То есть сервер базы данных «отдает» один из своих потоков для выполнения кода хранимой процедуры, но поток не возвращает управление. Сервер оказывается в опасном положении, и его поведение становится непредсказуемым. Например, может сильно упасть производительность. Может, серверу стоит создать дополнительные потоки? Но на это вам потребуются дополнительные ресурсы (например, место в стеке), кроме того, и этим потокам ничто не может помешать застрять в бесконечном цикле.

Чтобы раз и навсегда решить подобные проблемы, хосту предоставляется право завершения потока. На рис. 22.3 показана типичная архитектура хост-приложения, пытающегося решить проблему вышедшего из-под контроля потока. Вот как это происходит (номера операций соответствуют числам на рисунке).

1. Клиент направляет запрос на сервер.
2. Поток сервера принимает запрос и пересылает его потоку из пула для выполнения работы.

3. Поток из пула принимает клиентский запрос и выполняет доверенный код, то есть код, созданный в компании, в которой создано и протестировано само хост-приложение.
4. Этот доверенный код входит в блок try и вызывает из него другой домен (используя тип, производный от `MarshalByRefObject`). Этот домен содержит код (например, хранимую процедуру), созданный и протестированный сторонними разработчиками. После передачи управления потоком такому коду сервер начинает «нервничать».
5. Хост фиксирует время получения исходного клиентского запроса. Если сторонний код не отвечает клиенту за определенное администратором время, хост вызывает метод `Abort` типа `Thread`, требуя от CLR остановить поток из пула и вынуждая вбросить исключение `ThreadAbortException`.
6. На этом этапе поток пула начинает завершение, вызывая блоки `finally`, чтобы выполнить код очистки. В итоге поток возвращается в домен. Так как программа-заглушка вызвала сторонний код из блока `try`, в ней имеется и блок `catch`, который перехватывает исключение `ThreadAbortException`.
7. В ответ на перехват исключения `ThreadAbortException` хост вызывает метод `ResetAbort` типа `Thread`. Зачем это нужно, я объясню чуть позже.
8. Хост отправляет информацию о сбое клиенту и возвращает поток в пул, чтобы его можно было снова задействовать для обслуживания клиентских запросов.

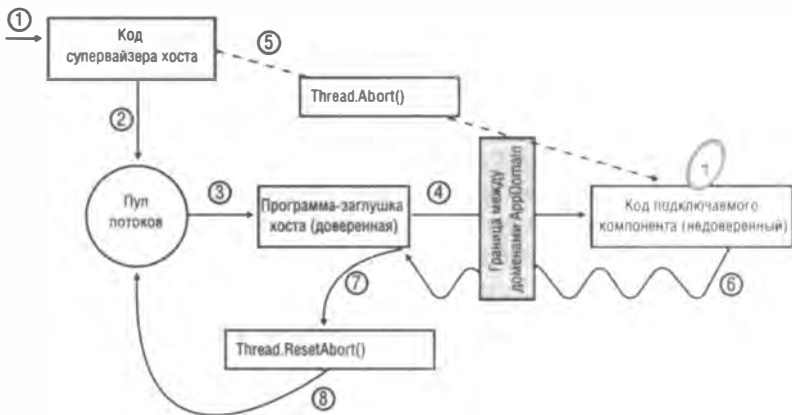


Рис. 22.3. Возвращение хостом контроля над потоком

А сейчас я проясню некоторые непонятные места этой архитектуры. Во-первых, метод `Abort` типа `Thread` выполняется асинхронно. Он отмечает целевой поток флагом `AbortRequested` и немедленно возвращает управление. Обнаружив завершение потока, исполняющая среда пытается перенести этот поток в *безопасное место* (*safe place*). Исполняющая среда считает, что поток находится в безопасном месте, если его можно (по ее мнению) остановить без риска

серьезных последствий. Поток находится в безопасном месте, если выполняет управляемую операцию блокировки, например бездействует или «спит». Перемещение в безопасное место осуществляется путем захвата (см. главу 21). Поток не считается находящимся в безопасном месте, если он выполняет конструктор класса типа, код блока `catch` или `finally`, код в критической области или неуправляемый код.

Как только поток оказывается в безопасном месте, исполняющая среда обнаруживает у него флаг `AbortRequested` и заставляет его вбросить исключение `ThreadAbortException`. Если исключение не перехватывается, оно остается необработанным, выполняются все блоки `finally` и поток корректно завершается. В отличие от всех прочих, оставшихся необработанным, исключение `ThreadAbortException` не приводит к остановке приложения. Исполняющая среда «проглатывает» его, и поток завершается, но приложение и все оставшиеся его потоки продолжают работу.

В рассматриваемом примере хост перехватывает исключение `ThreadAbortException`, получая возможность снова получить контроль над потоком и вернуть его в пул. Однако остается вопрос: что может запретить стороннему коду перехватить исключение `ThreadAbortException`, чтобы сохранить за собой контроль над потоком? У CLR к данному исключению особое отношение. Даже если код перехватывает исключение `ThreadAbortException`, CLR в конце блока `catch` автоматически повторно его вбрасывает.

В связи с этой особенностью CLR возникает другой вопрос: если CLR повторно вбрасывает исключение `ThreadAbortException` в конце блока `catch`, как же хосту удастся перехватить это исключение и восстановить контроль над потоком? В блоке `catch` хоста есть вызов метода `ResetAbort` типа `Thread`. Именно он запрещает CLR повторно вбрасывать исключение `ThreadAbortException` в конце каждого блока `catch`.

Тогда снова возникает вопрос: а что может запретить стороннему коду перехватить исключение `ThreadAbortException` и самому вызвать метод `ResetAbort` типа `Thread`? К счастью, для вызова этого метода у вызывающей программы должно быть разрешение `SecurityPermission` с флагом `ControlThread`, имеющим значение `true`. Создавая домен для кода сторонних разработчиков, хост не предоставляет такое разрешение, а, значит, такой код не сможет сохранить за собой контроль над его потоком.

Должен заметить, что брешь в системе безопасности в данном случае все-таки возможна: когда поток раскручивает исключение `ThreadAbortException`, сторонний код может выполнить блоки `catch` и `finally`, содержащие код с бесконечным циклом, не позволяющим хосту вернуть контроль над потоком. Эта проблема решается при помощи обсуждавшейся ранее политики расширения. Если останавливаемый поток не завершается за разумное время, CLR может перейти от корректной к принудительной остановке, принудительной выгрузке домена, отключению CLR или уничтожению процесса. Следует также заметить, что сторонний код может перехватить исключение `ThreadAbortException`

и вбросить в блоке `catch` какое-то другое исключение. Если оно перехватывается, в конце блока `catch` CLR автоматически повторно вбросит исключение `ThreadAbortException`.

Вместе с тем нужно сказать, что большинство сторонних программ не представляет угрозы — просто с точки зрения хоста они тратят на решение своей задачи слишком много времени. Обычно блоки `catch` и `finally` содержат очень немного кода, выполняемого быстро без каких-либо бесконечных циклов или «долгоиграющих» операций. Поэтому вряд ли вам потребуется политика расширения для возвращения управления потоком хосту.

Кстати, у класса `Thread` есть два метода `Abort`: один без параметров, а второй с параметром `Object`, в котором можно передать любой объект. Перехватив исключение `ThreadAbortException`, код может запросить свое предназначенное только для чтения свойство `ExceptionState`, которое вернет объект, переданный в качестве параметра. Это позволяет потоку, вызвавшему метод `Abort`, передать дополнительную информацию коду, перехватившему исключение `ThreadAbortException`. Хост может использовать это для информирования собственного перехватывающего кода о причине остановки потока.

Глава 23. Загрузка сборок и отражение

В этой главе вы узнаете все о том, как находить информацию о типах, создавать их экземпляры и обеспечивать доступ к их членам несмотря на то, что во время компиляции об этих типах ничего не известно. Сведения, приведенные в этой главе, обычно нужны для создания динамически расширяемых приложений, то есть таких, для которых одна компания создает хост-приложение, а другие — *подключаемые компоненты* (add-ins), которые расширяют функциональность хоста. Тестировать совместную работу хоста и подключаемых компонентов невозможно, так как последние создаются разными компаниями, причем, как правило, уже после выпуска хост-приложения. Вот почему хосту приходится самостоятельно находить подключаемые компоненты во время выполнения.

Динамически расширяемое приложение может использовать хостинг CLR и домены приложений, как описано в главе 22. Хост выполняет код подключаемых компонентов в отдельных доменах приложений с собственными параметрами защиты и конфигурирования. Хост также может выгрузить подключаемый компонент, выгрузив домен приложений, в котором он выполняется. В конце главы я задействую все эти механизмы — включая хостинг CLR, домены приложений, загрузку сборок, обнаружение типов, создание экземпляров типов и отражение — для создания надежного, безопасного и динамически расширяемого приложения.

Загрузка сборок

Как вы уже знаете, когда JIT-компилятор создает IL-код метода, он «смотрит», на какие типы есть ссылки в IL-коде. Далее во время выполнения JIT-компилятор использует таблицы метаданных TypeRef и AssemblyRef сборки, чтобы выяснить, в какой сборке определен упоминаемый тип. Запись таблицы AssemblyRef содержит все части строгого имени сборки. JIT-компилятор собирает все эти части — имя (без расширения и пути), версию, региональные стандарты и открытый ключ — в строку, а затем пытается загрузить сборку с таким именем в домен приложений (если она еще не загружена). Если загружается сборка с нестрогим именем, идентификационная информация представляет собой только имя сборки (без версии, региональных стандартов и открытого ключа).

CLR пытается загрузить эту сборку, используя статический метод `Load` класса `System.Reflection.Assembly`. Этот метод описан в открытой документации и его можно вызывать для явной загрузки сборки в свои приложения. Он представляет собой CLR-эквивалент Win32-функции `LoadLibrary`. В сущности, есть несколько перегруженных версий метода `Load` класса `Assembly`. Вот прототипы наиболее популярных из них:

```
public class Assembly {  
    public static Assembly Load(AssemblyName assemblyRef);  
    public static Assembly Load(String assemblyString);  
    // Менее популярные перегруженные версии не показаны  
}
```

Внутренний код `Load` заставляет CLR применить к сборке политику привязки версии с перенаправлением и ищет нужную сборку сначала в глобальном кэше сборок (GAC), а затем последовательно в базовом каталоге приложения, каталогах закрытых путей и каталоге, указанном в элементе `codeBase` конфигурационного файла. Если методу `Load` передается сборка с нестрогим именем, он не применяет к ней политику, и CLR не ищет ее в GAC. Найдя искомую сборку, `Load` возвращает ссылку на объект `Assembly`, представляющий загруженную сборку. Если указанная сборка не найдена, вбрасывается исключение `System.IO.FileNotFoundException`.

ПРИМЕЧАНИЕ

В небольшом количестве чрезвычайно редких ситуаций может потребоваться загрузить сборку, скопированную для определенной версии Microsoft Windows. В этом случае при определении идентификационной информации сборки можно указать сведения об архитектуре процесса. Например, если в GAC хранятся нейтральная и специализированная (x86) версии сборки, CLR предпочтет специализированную версию (см. главу 3). Однако можно заставить CLR загрузить нейтральную версию, передав в метод `Load` класса `Assembly` такую строку:

```
"SomeAssembly, Version=2.0.0.0, Culture=neutral,  
PublicKeyToken=01234567890abcde, ProcessorArchitecture=MSIL"
```

На момент написания этой книги CLR поддерживает четыре возможных значения параметра `ProcessorArchitecture`: `MSIL` (Microsoft IL), `x86`, `IA64` и `AMD64`.

ВНИМАНИЕ

Метод `Load` есть и у объекта `System.AppDomain`. В отличие от одноименного метода объекта `Assembly`, он является экземплярным методом, позволяющим загружать сборку в домен приложений. Этот метод создан для

неуправляемого кода, позволяя хосту загрузить сборку в определенный домен приложений. Разработчикам управляемого кода лучше его избегать, и вот почему. При вызове метода `Load` объекта `AppDomain` передается строка, идентифицирующая сборку. Этот метод затем применяет политику и ищет сборку в обычных местах: на пользовательском жестком диске или в базовом каталоге. Вспомните, что с каждым доменом приложений связаны параметры, определяющие правила поиска сборки для CLR. Так вот, при загрузке сборки CLR будет руководствоваться параметрами заданного, а не вызывающего домена приложений.

Однако метод `Load` объекта `AppDomain` возвращает ссылку на сборку. В силу того, что класс `System.Assembly` не является потомком `System.MarshalByRefObject`, объект сборки возвращается вызывающему домену приложений путем продвижения по значению. Но теперь для поиска и загрузки сборки CLR задействует параметры вызывающего домена приложений. Если сборку не удастся найти при помощи политики вызывающего домена приложений или в заданных им каталогах поиска, вбрасывается исключение `FileNotFoundException`. Такая ситуация обычно нежелательна, поэтому следует избегать метода `Load` объекта `System.AppDomain`.

В большинстве динамически расширяемых приложений метод `Load` объекта `AppDomain` является предпочтительным механизмом загрузки сборки в домен приложений, но он требует наличия всех частей, идентифицирующих сборку. Часто разработчики создают инструменты или утилиты (такие как `ILDasm.exe`, `PEVerify.exe`, `CorFlags.exe`, `GACUtil.exe`, `SGen.exe`, `SN.exe` и `XSD.exe`), которые определенным образом обрабатывают сборку. Все они принимают параметр командной строки, задающий путь (с расширением) к файлу сборки. Для загрузки сборки по пути вызывается метод `LoadFrom` класса `Assembly`:

```
public class Assembly {  
    public static Assembly LoadFrom(String path):  
        // Менее популярные перегруженные версии не показаны  
}
```

Код `LoadFrom` сначала вызывает метод `GetAssemblyName` класса `System.Reflection.AssemblyName`, который открывает указанный файл, находит запись таблицы метаданных `AssemblyRef`, извлекает идентификационную информацию сборки и возвращает ее в объекте `System.Reflection.AssemblyName` (файл при этом закрывается). Затем `LoadFrom` вызывает метод `Load` класса `Assembly`, передавая ему объект `AssemblyName`. На этом этапе CLR применяет политику перенаправления версий и ищет в определенных местах соответствующую сборку. Найдя сборку, `Load` загружает ее и возвращает объект `Assembly`, представляющий загруженную сборку; именно его возвращает `LoadFrom`. Если методу `Load` не удастся найти сборку, `LoadFrom` загружает сборку по пути, переданному в качестве параметра

в `LoadFrom`. Ясно, что если сборка с теми же идентификационными данными уже загружена, `LoadFrom` просто возвращает объект `Assembly`, представляющий уже загруженную сборку.

Кстати, методу `LoadFrom` можно передать в качестве параметра URL-адрес:

```
Assembly a = Assembly.LoadFrom(@"http://Wintellect.com/SomeAssembly.dll");
```

При получении URL-адреса среда CLR загружает файл, устанавливает его в загрузочный кэш пользователя и уже из него загружает файл. Система должна быть подключена к Интернету, иначе возникнет исключение. Однако если файл уже был загружен в кэш ранее, а браузер Internet Explorer настроен на работу в автономном режиме — команда **Work Offline (Работать автономно)** в меню **File (Файл)**, — будет использоваться файл из кэша, и исключение не возникнет. Вы также можете вызвать метод `UnsafeLoadFrom`, который загрузит уже загруженную веб-сборку, игнорируя параметры защиты.

ВНИМАНИЕ

На одной машине могут находиться разные сборки с одинаковой идентификационной информацией. Так как `LoadFrom` вызывает `Load`, может оказаться, что CLR загрузит не указанный, а другой файл, что чревато непредсказуемым поведением. Настоятельно рекомендуется при каждой компоновке сборки изменять номер редакции — так обеспечивается строгая индивидуальность идентификационной информации всех сборок, а значит, вызов метода `LoadFrom` не принесет неожиданностей.

Конструкторы графического интерфейса и другие инструменты Microsoft Visual Studio обычно используют метод `LoadFile` класса `Assembly`. Этот метод может загрузить сборку по любому пути и его можно задействовать для загрузки сборки с идентичными параметрами в единственный домен приложений. Это удобно в случае, когда при помощи конструктора или другого инструмента были внесены изменения в графический интерфейс приложения, а затем это приложение было собрано заново. При загрузке `LoadFile` среда CLR не сопоставляет автоматически все зависимости, поэтому ваш программный код должен быть зарегистрирован в событиях `AssemblyResolve` и иметь явно загруженные методы обратных вызовов событий всех зависимых сборок.

В тех исключительно редких случаях, когда требуется загрузить сборку из указанного места, запретив CLR поиск сборки и применение какой-либо политики, можно вызвать метод `LoadFile` класса `Assembly`.

Если вы создаете инструмент, который просто анализирует метаданные сборки с использованием отражения (об этом — чуть позже), не выполняя никакого кода сборки, лучше всего для загрузки сборки задействовать метод

ReflectionOnlyLoadFrom или, в некоторых редких случаях, метод ReflectionOnlyLoad класса Assembly. Вот прототипы обоих методов:

```
public class Assembly {  
    public static Assembly ReflectionOnlyLoadFrom(String assemblyFile);  
    public static Assembly ReflectionOnlyLoad(String assemblyString);  
    // Менее популярные перегруженные версии не показаны  
}
```

Метод ReflectionOnlyLoadFrom загружает указанный файл, не получая информацию строгого имени сборки и не выполняя поиск файла в GAC или где-либо еще. Метод ReflectionOnlyLoad выполняет поиск указанной сборки в GAC, базовом каталоге приложения, частных каталогах и каталоге, указанном в элементе codeBase. Однако в отличие от Load этот метод не применяет политику версий, поэтому не предоставляет гарантий, что будет загружена именно та сборка, которая ожидалась. Если вы хотите самостоятельно применить политику версий к сборке, можно передать строку с идентификационной информацией в метод AppDomain класса ApplyPolicy.

При загрузке сборок методом ReflectionOnlyLoadFrom или ReflectionOnlyLoad среда CLR запрещает выполнение какого-либо кода сборки, а при попытке выполнить код вбрасывает исключение InvalidOperationException. Эти методы позволяют инструменту загружать сборки с отложенным подписанием, сборки для процессора другой архитектуры, а также сборки, для загрузки которых нужны особые разрешения.

Часто при использовании отражения для анализа сборки, загруженной одним из указанных двух методов, код должен зарегистрировать метод обратного вызова на событие ReflectionOnlyAssemblyResolve класса AppDomain, чтобы вручную загружать произвольные сборки, задаваемые клиентом (при необходимости вызывая метод ApplyPolicy класса AppDomain); CLR не делает этого автоматически. Будучи вызванным, метод обратного вызова должен вызвать метод ReflectionOnlyLoadFrom или ReflectionOnlyLoad класса Assembly, чтобы явно загрузить указанную сборку и вернуть ссылку на нее.

ПРИМЕЧАНИЕ

Меня часто спрашивают о порядке выгрузки сборки. К сожалению, CLR не позволяет выгружать отдельные сборки. Если бы это было так, возможна была бы ситуация, когда поток возвращается из метода в код выгруженной сборки, и в результате происходит сбой приложения. Однако среда CLR стоит на страже надежности и безопасности, а подобные сбои непроизводительны и не соответствуют ее целям. Чтобы выгрузить сборку, придется выгрузить весь домен приложений, в котором она находится. Подробнее см. главу 22.

Казалось бы, сборки, загруженные методом `ReflectionOnlyLoadFrom` или `ReflectionOnlyLoad`, должно быть разрешено выгрузить. В конце концов, ведь код этих сборок нельзя выполнять. Однако CLR не разрешает выгрузку сборок, загруженных одним из этих методов, по той простой причине, что после загрузки сборок вы всегда сможете использовать отражение для создания объектов, ссылающихся на метаданные, определенные в этих сборках. При выгрузке сборки потребовалось бы каким-то образом сделать объекты недействительными, но отслеживание всех этих связей — слишком сложная и ресурсоемкая задача.

Многие приложения содержат EXE-файлы, зависящие от многих DLL-файлов. При установке этих приложений также должны устанавливаться все файлы. Однако обычно практикуется установка единственного EXE-файла. В этом случае в первую очередь идентифицируйте все DLL-файлы, от которых зависит ваш EXE-файл и которые не являются частью платформы Microsoft .NET Framework. Затем добавьте эти DLL-файлы к вашему проекту в Visual Studio и для каждого добавленного DLL-файла откройте окно свойств и измените значение **Build Action** на **Embedded Resource**. Это действие даст указание компилятору C# добавить DLL-файлы в EXE-файл, который в конечном итоге и будет устанавливаться. На этапе выполнения программы среда CLR не сможет найти зависимые сборки, что может вызвать проблемы. Для решения этих проблем зарегистрируйте методы обратного вызова при помощи события `ResolveAssembly` при инициализации вашего приложения. Программный код должен выглядеть следующим образом:

```
AppDomain.CurrentDomain.AssemblyResolve += (sender, args) => {  
    String resourceName = "AssemblyLoadingAndReflection." +  
        new AssemblyName(args.Name).Name + ".dll";  
  
    using (var stream =  
        Assembly.GetExecutingAssembly().GetManifestResourceStream(  
            resourceName)) {  
        Byte[] assemblyData = new Byte[stream.Length];  
        stream.Read(assemblyData, 0, assemblyData.Length);  
        return Assembly.Load(assemblyData);  
    }  
};
```

При первом вызове в потоке метода, ссылающегося на тип, зависящий от DLL-файла, возникнет событие `AssemblyResolve`, и показанный программный код обратного вызова найдет встроенный DLL-файл и загрузит его путем вызова перегруженного метода `Load`, у которого в качестве аргумента будет использоваться `Byte[]`.

Использование отражения для создания динамически расширяемых приложений

Как вам известно, метаданные — это набор таблиц. При компоновке сборки или модуля компилятор создает таблицы определений типов, полей, методов и т. д. В пространстве имен `System.Reflection` есть несколько типов, позволяющих писать код отражения (или синтаксического разбора) этих таблиц. На самом деле типы из этого пространства имен предлагают модель объектов для отражения метаданных сборки или модуля.

Типы, составляющие эту модель объектов, позволяют легко перечислить все типы из таблицы определений типов, а также получить для каждого из них базовый тип, интерфейсы и ассоциированные с ним флаги. Остальные типы из пространства имен `System.Reflection` дают возможность запрашивать поля, методы, свойства и события типа путем синтаксического разбора соответствующих таблиц метаданных. Можно узнать, какими атрибутами (см. главу 18) помечена та или иная сущность метаданных. Есть даже классы, позволяющие определить указанные сборки и методы и возвращающие в методе байтовый IL-поток. Имея эти данные, можно создать инструмент вроде `ILDasm.exe` разработки Microsoft.

ПРИМЕЧАНИЕ

Нужно иметь в виду, что некоторые типы отражения и часть их членов созданы специально для разработчиков, пишущих компиляторы для CLR. Прикладные разработчики обычно не используют эти типы и члены. В документации к библиотеке FCL не сказано четко, какие типы предназначены для разработчиков компиляторов, а какие — для разработчиков приложений, но если понимать, что некоторые типы и члены отражения предназначены «не для всех», то документация становится менее запутанной.

В реальности приложениям редко требуются типы отражения. Обычно отражение используется в библиотеках классов, которым нужно понять определение типа, чтобы дополнить его. Например, механизм сериализации из FCL (см. главу 24) применяет отражение, чтобы выяснить, какие поля определены в типе. Объект форматирования из механизма сериализации получает значения этих полей и записывает их в поток байтов для пересылки через Интернет. Аналогично, создатели Microsoft Visual Studio используют отражение, чтобы определить, какие свойства показывать разработчикам при размещении элементов на поверхности веб-формы или формы Windows Forms во время ее создания.

Отражение также применяют, когда для решения некоторой задачи во время выполнения приложению нужно загрузить определенный тип из некоторой сборки. Например, приложение может попросить пользователя предоставить имя сборки и типа, чтобы явно загрузить ее, создать экземпляр данного типа и вызывать его методы. Концептуально подобное использование отражения напоминает вызов Win32-функций `LoadLibrary` и `GetProcAddress`. Часто привязку к типам и вызываемым методам, осуществляемую таким образом, называют *поздним связыванием* (late binding) в отличие от *раннего связывания* (early binding), которое имеет место, когда требуемые приложению типы и методы известны при компиляции.

Производительность отражения

Отражение — исключительно мощный механизм, позволяющий во время выполнения обнаруживать и использовать типы и их члены, о которых во время компиляции ничего не было известно. Но у этой мощи есть два серьезных недостатка.

- ❑ При использовании отражения безопасность типов на этапе компиляции не контролируется. Так как в отражении активно применяются строки, вы теряете безопасность типов на этапе компиляции. Например, предположим, мы хотим задействовать отражение, чтобы обнаружить тип `Jef` в сборке, в которой на самом деле есть тип `Jeff`. В этом случае мы используем следующую инструкцию:

```
Type.GetType("Jef");
```

К сожалению, такой код без проблем компилируется, но во время выполнения возникает ошибка, так как в имени типа, переданного в качестве параметра, опечатка.

- ❑ Отражение работает медленно. При использовании отражения имена типов и их члены на момент компиляции не известны — они определяются в процессе выполнения, причем все типы и члены идентифицируются по строковому имени. Это значит, что при отражении постоянно выполняется поиск строк в метаданных сборки пространства имен `System.Reflection`. Часто выполняется строковый поиск без учета регистра, что дополнительно замедляет процесс.

В общем случае вызов метода или доступ к полю или свойству посредством отражения также работает медленно. При использовании отражения перед вызовом метода аргументы требуется сначала упаковать в массив и инициализировать его элементы, а потом при вызове метода извлекать аргументы из массива и помещать их в стек потока. Кроме того, CLR приходится проверять

правильность числа и типа параметров, переданных методу. И наконец, CLR проверяет наличие у вызывающего кода разрешений на доступ к члену.

В силу этих причин лучше не использовать отражение для доступа к члену. Если вы пишете приложение, которое динамически ищет и создает объекты, следуйте одному из перечисленных далее подходов.

- ❑ Порождайте свои типы от базового типа, известного на момент компиляции. Затем, создав экземпляр своего типа во время выполнения, поместите ссылку на него в переменную базового типа (выполнив приведение типа) и вызывайте виртуальные методы базового типа.
- ❑ Реализуйте в типах интерфейсы, известные на момент компиляции. Затем, создав экземпляр своего типа во время выполнения, поместите ссылку на него в переменную того же типа, что и интерфейс (выполнив приведение типа), и вызывайте методы, определенные в интерфейсе.

Я предпочитаю второй подход, так как в первом случае разработчику невозможно выбрать базовый тип, оптимальный для конкретной ситуации. Хотя методика порождения своего типа от базового лучше в отношении контроля версий, потому что вы всегда добавляете члены в базовый тип и наследуете от него свой тип, но не можете добавить члены в интерфейс без принудительного изменения программного кода всех типов, реализующих этот интерфейс, и их повторной компиляции.

В любом случае я настоятельно рекомендую определять базовый тип или интерфейс в их собственной сборке — будет меньше проблем с управлением версиями. Подробнее об этом см. раздел «Создание приложений с поддержкой подключаемых компонентов».

Нахождение типов, определенных в сборке

Отражение часто используется, чтобы выяснить, какие типы определены в сборке. Для получения этой информации FCL предлагает несколько методов. Наиболее популярный — метод `GetExportedTypes` класса `Assembly`. Вот пример кода, который загружает сборку и выводит имена всех определенных в ней открытых экспортированных типов:

```
using System;  
using System.Reflection;
```

```
public static class Program {  
    public static void Main() {  
        String dataAssembly = "System.Data, version=2.0.0.0, " +  
            "culture=neutral, PublicKeyToken=b77a5c561934e089";  
        LoadAssemAndShowPublicTypes(dataAssembly);  
    }  
}
```

продолжение ➤

```
private static void LoadAssemAndShowPublicTypes(String assemId) {
    // Явно загружаем сборку в домен приложений
    Assembly a = Assembly.Load(assemId);

    // Выполняем цикл для каждого открытого типа.
    // экспортируемого загруженной сборкой
    foreach (Type t in a.GetExportedTypes()) {
        // Выводим полное имя типа
        Console.WriteLine(t.FullName);
    }
}
```

Объект Type

Обратите внимание, что приведенный код итеративно обрабатывает массив объектов `System.Type`. Тип `System.Type` — отправная точка для операций с типами и объектами. Это абстрактный тип, производный от `System.Reflection.MemberInfo` (так как тип `Type` может быть членом другого типа). FCL предоставляет несколько типов, производных от `System.Type`. Это типы `System.RuntimeType`, `System.ReflectionOnlyType`, `System.Reflection.TypeDelegator`, а также некоторые типы, определенные в пространстве имен `System.Reflection.Emit`, включая `EnumBuilder`, `GenericTypeParameterBuilder` и `TypeBuilder`.

ПРИМЕЧАНИЕ

Класс `TypeDelegator` позволяет из кода динамически создавать подклассы класса `Type` путем инкапсуляции типа `Type`. Это позволяет переопределить некоторые методы и позволить типу `Type` сделать остальную работу. Такой мощный механизм позволяет переопределить поведение отражения.

Из всех этих типов самый интересный — `System.RuntimeType`. Это внутренний тип библиотеки FCL, поэтому вы не найдете его описание в документации к этой библиотеке. При первом обращении в домене приложений к типу CLR создает экземпляр `RuntimeType` и инициализирует поля объекта информацией о типе.

Как вы помните, в `System.Object` определен открытый неvirtуальный метод `GetType`. Если его вызвать, CLR определит тип указанного объекта и вернет ссылку на его объект `RuntimeType`. Поскольку для каждого типа в домене приложений есть только один объект `RuntimeType`, можно задействовать операторы равенства и неравенства, чтобы выяснить, относятся ли объекты к одному типу:

```
private static Boolean AreObjectsTheSameType(Object o1, Object o2) {
    return o1.GetType() == o2.GetType();
}
```

Помимо вызова метода `GetType` класса `Object` FCL предлагает другие способы получения объекта `Type`:

- ❑ В типе `System.Type` есть несколько перегруженных версий статического метода `GetType`. Все они принимают тип `String`. Эта строка должна содержать полное имя типа (включая его пространства имен). Заметьте: имена элементарных типов, поддерживаемые компилятором (такие как `int`, `string`, `bool` и другие типы языка C#), запрещены, потому что они ничего не значат для CLR. Если строка содержит просто имя типа, метод проверяет, определен ли тип с указанным именем в вызывающей сборке. Если это так, возвращается ссылка на соответствующий объект `RuntimeType`.

Если в вызывающей сборке указанный тип не определен, проверяются типы, определенные в `mscorlib.dll`. Если и после этого тип с указанным именем найти не удастся, возвращается `null` или вбрасывается исключение `System.TypeLoadException` — все зависит от того, какая перегруженная версия метода `GetType` вызывалась и какие ей передавались параметры. В документации на FCL есть исчерпывающее описание этого метода.

В `GetType` можно передать полное имя типа с указанием сборки, например:

```
"System.Int32, mscorlib, Version=2.0.0.0, Culture=neutral,  
  PublicKeyToken=b77a5c561934e089"
```

В этом случае `GetType` будет искать тип в указанной сборке (и при необходимости загрузит ее).

- ❑ В типе `System.Type` есть статический метод `ReflectionOnlyGetType`. Этот метод ведет себя так же, как только что описанный метод `GetType`, за исключением того, что тип загружается только для отражения, но не для выполнения кода.
- ❑ В типе `System.Type` есть экземплярные методы `GetNestedType` и `GetNestedTypes`.
- ❑ В типе `System.Reflection.Assembly` есть экземплярные методы `GetType`, `GetTypes` и `GetExportedTypes`.
- ❑ В типе `System.Reflection.Module` есть экземплярные методы `GetType`, `GetTypes` и `FindTypes`.

ПРИМЕЧАНИЕ

Microsoft использует нотацию Бэкуса–Наура для записи имен типов и имен с указанием сборки, которые используются для написания строк, передаваемых в методы отражения. Знание нотации оказывается очень кстати при использовании отражения и особенно при работе с вложенными типами, обобщенными типами и методами, ссылочными параметрами или массивами. Полное описание нотации вы найдете в документации к FCL или можете выполнить поиск в Интернете по строке «Backus-Naur Form Grammar for Type Names». Вы также можете посмотреть методы `MakeArrayType`, `MakeByRefType`, `MakeGenericType`, и `MakePointerType` класса `Type`.

Во многих языках программирования есть оператор, позволяющий получить объект `Type` по имени типа. Для получения ссылки на `Type` лучше использовать именно такой оператор, а не перечисленные методы, так как при компиляции оператора получается более быстрый код. В С# это оператор `typeof`, хотя обычно его не применяют для сравнения информации о типах, загруженных посредством позднего и раннего связывания, как в следующем примере:

```
private static void SomeMethod(Object o) {  
    // GetType возвращает тип объекта во время выполнения  
    // (позднее связывание)  
    // typeof возвращает тип указанного класса  
    // (раннее связывание)  
    if (o.GetType() == typeof(FileInfo)) { ... }  
    if (o.GetType() == typeof(DirectoryInfo)) { ... }  
}
```

ПРИМЕЧАНИЕ

Первая инструкция `if` проверяет, ссылается ли переменная `o` на объект типа `FileInfo`, но не на тип, производный от `FileInfo`. Иначе говоря, этот код проверяет на точное, а не на совместимое соответствие. Совместимое соответствие обычно достигается путем приведения типов либо использования оператора `is` или `as` языка С#.

Получив ссылку на объект `Type`, можно запросить многие свойства типа и узнать о них много полезного. Большинство свойств, таких как `IsPublic`, `IsSealed`, `IsAbstract`, `IsClass`, `IsValueType` и т. д., описывают флаги, связанные с типом. Другие свойства, к ним относятся `Assembly`, `AssemblyQualifiedName`, `FullName`, `Module` и пр., возвращают имя сборки, в которой определен тип или модуль, и полное имя типа. Можно также запросить свойство `BaseType`, чтобы узнать базовый тип.

В документации FCL описываются все методы и свойства типа `Type`. Но имейте в виду — их очень много. На самом деле тип `Type` предоставляет более 50 открытых экземплярных свойств. А еще есть методы и поля. О некоторых из этих методов я расскажу далее.

Создание иерархии типов, производных от `Exception`

В приложении-примере `ExceptionTree` (исходный текст см. далее) описанные концепции используются, чтобы загрузить в домен приложений определенное подмножество сборок и показать все типы, которые в конечном итоге наследуют от типа `System.Exception`. Кстати, это программа, которую я написал, чтобы создать иерархию исключений, приведенную в главе 20.

```

public static void Go() {
    // Явная загрузка сборок для отражения
    LoadAssemblies();

    // Рекурсивная сборка иерархии класса как строки, разделенной дефисами
    Func<Type, String> ClassNameAndBase = null;
    ClassNameAndBase = t => "-" + t.FullName +
        ((t.BaseType != typeof(Object)) ? ClassNameAndBase(t.BaseType) :
        String.Empty);

    // Определение запроса для нахождения всех открытых типов,
    // унаследованных от Exception в данной сборке домена приложений
    var exceptionTree =
        (from a in AppDomain.CurrentDomain.GetAssemblies()
         from t in a.GetExportedTypes()
         where t.IsClass && t.IsPublic && typeof(Exception).IsAssignableFrom(t)
         let typeHierarchyTemp = ClassNameAndBase(t).Split('-').Reverse().ToArray()
         let typeHierarchy =
             String.Join("-", typeHierarchyTemp, 0, typeHierarchyTemp.Length - 1)
         orderby typeHierarchy
         select typeHierarchy).ToArray();

    // Вывод дерева исключений
    Console.WriteLine("{0} Exception types found.", exceptionTree.Length);
    foreach (String s in exceptionTree) {
        // Для этого типа исключений разделить базовые типы
        String[] x = s.Split('-');

        // Изъять типы, основанные на # базовых типов, и показать
        // унаследованные типы
        Console.WriteLine(new String(' ', 3 * (x.Length - 1)) + x[x.Length - 1]);
    }
}

private static void LoadAssemblies() {
    String[] assemblies = {
        "System, PublicKeyToken={0}",
        "System.Core, PublicKeyToken={0}",
        "System.Data, PublicKeyToken={0}",
        "System.Design, PublicKeyToken={1}",
        "System.DirectoryServices, PublicKeyToken={1}",
        "System.Drawing, PublicKeyToken={1}",
        "System.Drawing.Design, PublicKeyToken={1}",
        "System.Management, PublicKeyToken={1}",
        "System.Messaging, PublicKeyToken={1}",
        "System.Runtime.Remoting, PublicKeyToken={0}",
    }
}

```

```

"System.Security, PublicKeyToken={1}",
"System.ServiceProcess, PublicKeyToken={1}",
"System.Web, PublicKeyToken={1}",
"System.Web.RegularExpressions, PublicKeyToken={1}",
"System.Web.Services, PublicKeyToken={1}",
"System.Windows.Forms, PublicKeyToken={0}",
"System.Xml, PublicKeyToken={0}",
};

String EcmaPublicKeyToken = "b77a5c561934e089";
String MSPublicKeyToken = "b03f5f7f11d50a3a";

// Получение версии сборки, содержащей System.Object
// Мы принимаем одну версию для всех других сборок
Version version = typeof(System.Object).Assembly.GetName().Version;

// Явная загрузка сборок, которые мы хотим отразить
foreach (String a in assemblies) {
    String AssemblyIdentity =
        String.Format(a, EcmaPublicKeyToken, MSPublicKeyToken) +
        ", Culture=neutral, Version=" + version;
    Assembly.Load(AssemblyIdentity);
}
}

```

Создание экземпляра типа

Получив ссылку на объект, производный от `Type`, можно создать экземпляр этого типа. FCL предлагает для этого несколько механизмов.

- ❑ **Методы `CreateInstance` класса `System.Activator`.** Этот класс поддерживает несколько перегруженных версий статического метода `CreateInstance`. При вызове этому методу передается ссылка на объект `Type` либо значение `String`, идентифицирующее тип объекта, который нужно создать. Версии, принимающие тип `Type`, проще: вы передаете методу набор аргументов конструктора, а он возвращает ссылку на новый объект.

Версии `CreateInstance`, в которых желаемый тип задают строкой, чуть сложнее. Во-первых, для них нужна еще и строка, идентифицирующая сборку, в которой определен тип. Во-вторых, эти методы позволяют создавать удаленные объекты, если правильно настроить параметры удаленного доступа. В-третьих, вместо ссылки на новый объект эти версии метода возвращают объект `System.Runtime.Remoting.ObjectHandle` (производный от `System.MarshalByRefObject`).

`ObjectHandle` — это тип, позволяющий передать объект, созданный в одном домене приложений, в другой домен, не загружая в целевой домен приложений сборку, в которой определен этот тип. Подготовившись к работе

с переданным объектом, нужно вызвать метод `Unwrap` объекта `ObjectHandle`. Только после этого загружается сборка, в которой находятся метаданные переданного типа. Если выполняется продвижение объекта по ссылке, создаются тип-представитель и объект-представитель. При продвижении по значению копия десериализуется.

- ❑ **Методы `CreateInstanceFrom` объекта `System.Activator`.** Класс `Activator` также поддерживает несколько статических методов `CreateInstanceFrom`. Они не отличаются от `CreateInstance` за исключением того, что для них всегда нужно задавать строковыми параметрами тип и сборку, в которой он находится. Заданная сборка загружается в вызывающий домен приложений методом `LoadFrom` (а не `Load`) объекта `Assembly`. Поскольку ни один из методов `CreateInstanceFrom` не принимает параметр `Type`, все они возвращают ссылку на тип `ObjectHandle`, с которого нужно снять оболочку.
- ❑ **Методы объекта `System.AppDomain`.** Тип `AppDomain` поддерживает четыре экземплярных метода (у каждого есть несколько перегруженных версий), создающих экземпляр типа: `CreateInstance`, `CreateInstanceAndUnwrap`, `CreateIntanceFrom` и `CreateInstanceFromAndUnwrap`. Они работают совсем как методы `Activator`, но являются экземплярными методами, позволяя задать домен приложений, в котором нужно создать объект. Методы, названия которых оканчиваются на `Unwrap`, удобнее, так как они дают возможность отказаться от дополнительного вызова метода.
- ❑ **Экземплярный метод `InvokeMember` объекта `System.Type`.** При помощи ссылки на объект `Type` можно вызвать метод `InvokeMember`. Последний ищет конструктор, соответствующий переданным параметрам, и создает объект. Новый объект всегда создается в вызывающем домене приложений, а затем возвращается ссылка на него. Позже мы обсудим этот метод подробнее.
- ❑ **Экземплярный метод `Invoke` объекта `System.Reflection.ConstructorInfo`.** При помощи ссылки на объект `Type` можно привязаться к некоторому конструктору и получить ссылку на объект `ConstructorInfo`, чтобы затем вызвать его метод `Invoke`. Новый объект всегда создается в вызывающем домене приложений, а затем возвращается ссылка на новый объект. К этому методу мы тоже вернемся позднее в этой главе.

ПРИМЕЧАНИЕ

Среда CLR не требует, чтобы у значимого типа был конструктор. И это становится проблемой, так как все перечисленные механизмы создают объект путем вызова его конструктора. Однако версии метода `CreateInstance` типа `Activator` позволяют создавать экземпляры значимых типов, не вызывая их конструктор. Чтобы создать экземпляр значимого типа, не вызывая его конструктор, нужно вызвать версию `CreateInstance`, принимающую единственный параметр `Type`, или версию, принимающую параметры `Type` и `Boolean`.

Эти механизмы позволяют создавать объекты любых типов, кроме массивов (то есть типов, производных от `System.Array`) и делегатов (потомков типа `System.MulticastDelegate`). Чтобы создать массив, надо вызвать статический метод `CreateInstance` объекта `Array` (существует несколько перегруженных версий этого метода). Первый параметр всех версий `CreateInstance` — это ссылка на объект `Type`, описывающий тип элементов массива. Прочие параметры `CreateInstance` позволяют задавать размерность и границы массива.

Для создания делегата следует вызвать статический метод `CreateDelegate` объекта `Delegate` (у этого метода также есть несколько перегруженных версий). Первый параметр любой версии `CreateDelegate` — это ссылка на объект `Type`, описывающий тип делегата. Остальные параметры позволяют указать, для какого экземплярного метода объекта или для какого статического метода типа делегат должен служить оболочкой.

Для создания экземпляра обобщенного типа сначала нужно получить ссылку на открытый тип, а затем вызвать открытый экземплярный метод `MakeGenericType` объекта `Type`, передав массив типов, который нужно использовать в качестве параметров типа. Затем надо получить возвращенный объект `Type` и передать его в один из описанных ранее методов. Вот пример:

```
using System;
using System.Reflection;

internal sealed class Dictionary<TKey, TValue> { }

public static class Program {
    public static void Main() {
        // Получаем ссылку на объект Type обобщенного типа
        Type openType = typeof(Dictionary<,>);

        // Закрываем обобщенный тип, используя TKey=String, TValue=Int32
        Type closedType = openType.MakeGenericType(
            new Type[] { typeof(String), typeof(Int32) });

        // Создаем экземпляр закрытого типа
        Object o = Activator.CreateInstance(closedType);

        // Проверяем, работает ли наше решение
        Console.WriteLine(o.GetType());
    }
}
```

Скомпилировав и выполнив этот код, мы получим:

```
Dictionary`2[System.String,System.Int32]
```

Создание приложений с поддержкой подключаемых компонентов

В построении открытых расширяемых приложений неоценимую помощь оказывают интерфейсы. Вместо них можно было бы задействовать базовые классы, но в общем случае интерфейс предпочтительнее, так как позволяет разработчику подключаемого компонента (add-in) выбрать собственные базовые классы. Допустим, вы хотите создать приложение, позволяющее пользователям создавать типы, которые ваше приложение сможет загружать и применять. Такое приложение должно строиться следующим образом.

- ❑ Создайте сборку для хоста, определяющую интерфейс с методами, используемыми как механизм взаимодействия вашего приложения с подключаемыми компонентами. Определяя параметры и возвращаемые значения методов этого интерфейса, постарайтесь задействовать другие интерфейсы или типы, определенные в `MSCorLib.dll`. Если нужно передавать и возвращать собственные типы данных, определите их в этой же сборке. Задав интерфейс, дайте сборке строгое имя (см. главу 3), после чего можете передать ее своим партнерам и пользователям. После публикации нужно избегать любых изменений типов сборки, которые могут нарушить работу подключаемых модулей. В частности, вообще нельзя изменять интерфейс. Но если вы определили типы данных, ничего не случится, если вы добавите в них новые члены. Внеся какие-либо изменения в сборку, нужно развертывать ее вместе с файлом политики издателя (см. главу 3).

ПРИМЕЧАНИЕ

Типы, определенные в `MSCorLib.dll`, можно использовать: CLR всегда загружает ту версию `MSCorLib.dll`, которая соответствует версии самой среды CLR. Кроме того, в процесс всегда загружается только одна версия `MSCorLib.dll`. Иначе говоря, разные версии `MSCorLib.dll` никогда не загружаются совместно (см. главу 3). В итоге несоответствий версий типа не будет, и ваше приложение использует меньше памяти.

- ❑ Разработчики подключаемых компонентов, конечно, определяют свои типы в собственных сборках. Кроме того, их сборки будут ссылаться на вашу интерфейсную сборку. Сторонние разработчики также смогут выдавать новые версии своих сборок, когда захотят: приложение сможет воспринимать подключаемые типы без проблем.
- ❑ Создайте сборку, содержащую типы вашего приложения. Очевидно, она будет ссылаться на интерфейс и типы, определенные в первой сборке. Код сборки вы можете изменять как угодно. Поскольку разработчики подключаемых компонентов не ссылаются на эту сборку, вы можете в любой момент выдать ее новую версию, и это не затронет сторонних разработчиков.

Этот короткий раздел содержит очень важную информацию. Используя типы в разных сборках, нельзя забывать о версиях. Не пожалейте времени и выделите в отдельную сборку типы, которые вы применяете для взаимодействия между сборками. Избегайте изменений этих типов и номера версии такой сборки. Однако если вам действительно нужно изменить определения типов, обязательно поменяйте номер версии и создайте файл политики издателя для новой версии.

А теперь я реализую один очень простой сценарий, в котором используется все, о чем мы говорили. Во-первых, нам нужен код сборки для хоста:

```
using System;
```

```
namespace Wintellect.HostSDK {
    public interface IAddIn {
        String DoSomething(Int32 x);
    }
}
```

Затем идет код сборки подключаемого компонента — библиотеки **AddInTypes.dll**, в которой определены два открытых типа, реализующие интерфейс **HostSDK**:

```
using System;
using Wintellect.HostSDK;

public sealed class AddIn_A : IAddIn {
    public AddIn_A() {
    }
    public String DoSomething(Int32 x) {
        return "AddIn_A: " + x.ToString();
    }
}

public sealed class AddIn_B : IAddIn {
    public AddIn_B() {
    }
    public String DoSomething(Int32 x) {
        return "AddIn_B: " + (x * 2).ToString();
    }
}
```

Третьим рассмотрим код сборки простого хоста (консольного приложения) — файла **Host.exe**. При компоновке этой сборки используется ссылка на **HostSDK.dll**. Определяя используемые подключаемым модулем типы, код хоста предполагает, что искомые типы определены в сборках, файлы которых содержат расширение **dll**, а сами сборки развернуты в том же каталоге, что и **EXE**-файл хоста. На сегодняшний день в CLR нет общепринятого механизма регистрации и нахождения подключаемых модулей — Microsoft планирует реа-

лизовать его в одной из будущих версий CLR. К сожалению, для каждого хоста приходится создавать собственный механизм регистрации и нахождения.

```
using System;
using System.IO;
using System.Reflection;
using System.Collections.Generic;
using Wintellect.HostSDK;

public static class Program {
    public static void Main() {
        // Определяем каталог, содержащий файл Host.exe
        String AddInDir = Path.GetDirectoryName(
            Assembly.GetEntryAssembly().Location);

        // Предполагаем, что сборки подключаемых модулей
        // находятся в одном каталоге с EXE-файлом хоста
        String[] AddInAssemblies = Directory.GetFiles(AddInDir, "*.dll");

        // Создаем набор типов, доступных для использования
        // подключаемыми модулями
        List<Type> AddInTypes = new List<Type>();
        // Загружаем сборки подключаемых модулей;
        // выясняем, какие типы могут использоваться хостом

        foreach (String file in AddInAssemblies) {
            Assembly AddInAssembly = Assembly.LoadFrom(file);
            // Анализирует каждый экспортируемый открытый тип
            foreach (Type t in AddInAssembly.GetExportedTypes()) {
                // Если тип представляет собой класс, реализующий интерфейс IAddIn,
                // значит, он доступен для использования хостом
                if (t.IsClass && typeof(IAddIn).IsAssignableFrom(t)) {
                    AddInTypes.Add(t);
                }
            }
        }

        // Инициализация завершена: хост обнаружил все используемые
        // подключаемые модули

        // Вот пример того, как хост создает и использует объекты
        // подключаемого модуля
        foreach (Type t in AddInTypes) {
            IAddIn ai = (IAddIn) Activator.CreateInstance(t);
            Console.WriteLine(ai.DoSomething(5));
        }
    }
}
```


В этом простом примере не используются домены приложений. Однако в реальной жизни подключаемые модули создаются в собственных доменах приложений с собственными параметрами защиты и конфигурирования. И, конечно же, домен приложений можно выгрузить, если нужно удалить подключаемый модуль из памяти. Чтобы обеспечить обмен между доменами приложений, лучше всего потребовать от разработчиков подключаемых модулей создавать собственные внутренние типы, производные от `MarshalByRefObject`. При создании нового домена приложений хост создаст в нем экземпляр собственного производного от `MarshalByRefObject` типа. Код хоста (в основном домене) будет взаимодействовать с собственным типом (в других доменах), заставляя их загружать сборки подключаемых модулей, создавая и используя экземпляры определенных в этих модулях типов.

Нахождение членов типа путем отражения

В этой главе я уже рассказывал о тех составляющих механизма отражения — загрузке сборок, нахождении типов и создании объектов, — которые необходимы для создания динамически расширяемых приложений. Однако, чтобы обеспечить высокую производительность и безопасность типов во время компиляции, нужно избегать отражения. В динамически расширяемом приложении после создания объекта код хоста обычно приводит объект к интерфейсному типу (это предпочтительный вариант) или базовому классу, известному на момент компиляции; это обеспечивает быстроту доступа к членам объекта и безопасность типов во время компиляции.

В оставшейся части этой главы я рассказываю о некоторых аспектах отражения, применяемых для нахождения и вызова членов типа. Нахождение и вызов членов типа обычно требуется при создании инструментов для разработчиков и средств анализа сборки, ориентированных на выявление определенных структур в программном коде или использование определенных членов. В качестве примера таких инструментов приведу `ILDasm`, `FxCop` и конструкторы форм для приложений `Windows Forms` и `Web Forms`, разрабатываемых в `Visual Studio`. Также в некоторых библиотеках классов существует возможность нахождения и вызова членов типа для предоставления разработчикам насыщенной функциональности. Пример — библиотеки, обеспечивающие сериализацию и десериализацию, а также простую привязку к данным.

Нахождение членов типа

Членами типа могут быть поля, конструкторы, методы, свойства, события и вложенные типы. В FCL есть тип `System.Reflection.MemberInfo` — абстракт-

ный класс, инкапсулирующий набор свойств, общих для всех членов типа. У `MemberInfo` много дочерних классов, каждый из которых инкапсулирует чуть больше свойств отдельных членов типа (рис. 23.1).

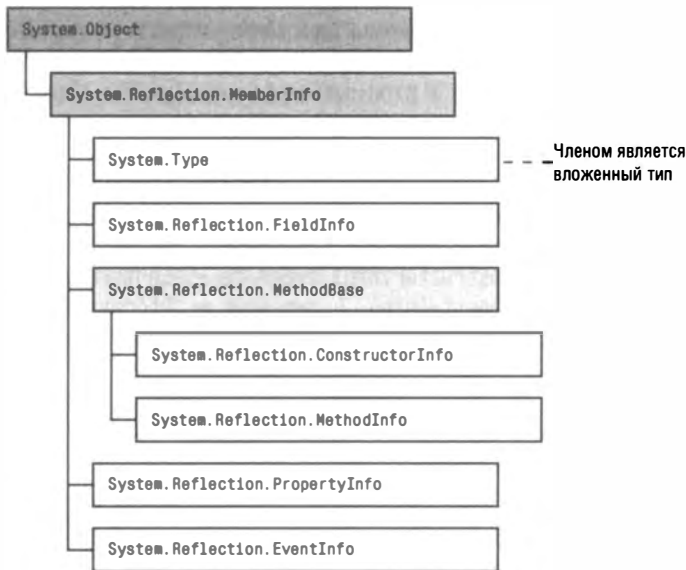


Рис. 23.1. Иерархия типов отражения

Приведенная далее программа демонстрирует, как нужно запрашивать члены типа и выводить информацию о них. Этот код обрабатывает все открытые типы всех сборок, загруженных в вызывающий домен приложений. Для каждого типа вызывается метод `GetMembers`, который возвращает массив объектов типа, производного от `MemberInfo`; каждый объект ссылается на один член из определенных в типе. Переменная `bf` типа `BindingFlags`, передаваемая в метод `GetMembers`, говорит ему, какие члены вернуть (подробнее о типе `BindingFlags` я расскажу чуть позже). Далее для каждого члена выводятся его описание (поле, конструктор, метод, свойство и т. п.) и строковое значение.

```
using System;
using System.Reflection;
```

```
public static class Program {
    public static void Main() {
        // В цикле перечисляем все сборки, загруженные в данный домен приложений
        Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
        foreach (Assembly a in assemblies) {
            WriteLine(0, "Assembly: {0}", a);
        }
    }
}
```

продолжение ➤

```

// Перечисляем типы сборки
foreach (Type t in a.GetExportedTypes()) {
    WriteLine(1, "Type: {0}", t);

    // Находим члены типа
    const BindingFlags bf = BindingFlags.DeclaredOnly |
        BindingFlags.NonPublic | BindingFlags.Public |
        BindingFlags.Instance | BindingFlags.Static;
    foreach (MemberInfo mi in t.GetMembers(bf)) {
        String typeName = String.Empty;
        if (mi is Type)           typeName = "(Nested) Type";
        else if (mi is FieldInfo)  typeName = "FieldInfo";
        else if (mi is MethodInfo) typeName = "MethodInfo";
        else if (mi is ConstructorInfo) typeName = "ConstructoInfo";
        else if (mi is PropertyInfo) typeName = "PropertyInfo";
        else if (mi is EventInfo)  typeName = "EventInfo";

        WriteLine(2, "{0}: {1}", typeName, mi);
    }
}
}
}

private static void WriteLine(Int32 indent, String format,
    params Object[] args) {
    Console.WriteLine(new String(' ', 3 * indent) + format, args);
}
}

```

После компиляции и запуска приложения мы получаем массу информации. Вот ее часть:

```

Assembly: mscorlib, Version=2.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089
Type: System.Object
MethodInfo: Boolean InternalEquals(System.Object, System.Object)
MethodInfo: Int32 InternalGetHashCode(System.Object)
MethodInfo: System.Type GetType()
MethodInfo: System.Object MemberwiseClone()
MethodInfo: System.String ToString()
MethodInfo: Boolean Equals(System.Object)
MethodInfo: Boolean Equals(System.Object, System.Object)
MethodInfo: Boolean ReferenceEquals(System.Object, System.Object)
MethodInfo: Int32 GetHashCode()
MethodInfo: Void Finalize()
MethodInfo: Void FieldSetter(System.String, System.String,
    System.Object)

```

```

MethodInfo: Void FieldGetter(System.String, System.String, System.Object
    ByRef)
MethodInfo: System.Reflection.FieldInfo GetFieldInfo(System.String,
    System.String)
ConstructoInfo: Void .ctor()
Type: System.Collections.IEnumerable
    MethodInfo: System.Collections.IEnumerator GetEnumerator()
Type: System.Collections.Generic.IComparer`1[T]
    MethodInfo: Int32 Compare(T, T)
Type: System.ValueType
    MethodInfo: Int32 GetHashCode()
    MethodInfo: Boolean CanCompareBits(System.Object)
    MethodInfo: Boolean FastEqualsCheck(System.Object, System.Object)
    MethodInfo: Boolean Equals(System.Object)
    MethodInfo: System.String ToString()
ConstructoInfo: Void .ctor()
Type: System.IDisposable
    MethodInfo: Void Dispose()
Type: System.Collections.Generic.IEnumerator`1[T]
    MethodInfo: T get_Current()
    PropertyInfo: T Current
Type: System.ArraySegment`1[T]
    MethodInfo: T[] get_Array()
    MethodInfo: Int32 get_Offset()
    MethodInfo: Int32 get_Count()
    MethodInfo: Int32 GetHashCode()
    MethodInfo: Boolean Equals(System.Object)
    MethodInfo: Boolean Equals(System.ArraySegment`1[T])
    MethodInfo: Boolean op_Equality(System.ArraySegment`1[T],
        System.ArraySegment`1[T])
    MethodInfo: Boolean op_Inequality(System.ArraySegment`1[T],
        System.ArraySegment`1[T])
ConstructoInfo: Void .ctor(T[])
ConstructoInfo: Void .ctor(T[], Int32, Int32)
PropertyInfo: T[] Array
PropertyInfo: Int32 Offset
PropertyInfo: Int32 Count
FieldInfo: T[] _array
FieldInfo: Int32 _offset
FieldInfo: Int32 _count

```

Так как тип `MemberInfo` является корнем иерархии, стоит обсудить его подробнее. В табл. 23.1 показаны некоторые неизменяемые (только для чтения) свойства и методы типа `MemberInfo`, общие для всех членов типа. Как вы помните, `System.Type` наследует от типа `MemberInfo`, поэтому `Type` также обладает всеми перечисленными в таблице свойствами.

Таблица 23.1. Свойства и методы, общие для всех типов, производных от MemberInfo

Имя члена	Тип члена	Описание
Name	Свойство String	Возвращает имя члена. В случае вложенного типа Name возвращает конкатенацию имени типа-контейнера, за которым следуют знак плюс (+) и имя вложенного типа
DeclaringType	Свойство Type	Возвращает тип, объявляющий член
ReflectedType	Свойство Type	Возвращает тип, определяющий член
Module	Свойство Module	Возвращает модуль, объявляющий член
MetadataToken	Свойство Int32	Возвращает маркер метаданных (в рамках модуля), идентифицирующий член
GetCustom-Attributes	Метод, возвращающий Object[]	Возвращает массив, каждый элемент которого идентифицирует экземпляр настраиваемого атрибута, которым помечен этот член. Такие атрибуты можно применять к любому члену
GetCustom-AttributesData	Метод, возвращающий IList<CustomAttributeData>	Возвращает коллекцию, каждый элемент которой идентифицирует экземпляр настраиваемого атрибута, которым помечен этот член. Хотя тип Assembly не унаследован от MemberInfo, он предоставляет те же методы для работы со сборками
IsDefined	Метод, возвращающий Boolean	Возвращает true, если, по крайней мере, один экземпляр настраиваемого атрибута применен к члену

Большинство названий свойств, представленных в таблице, говорят сами за себя. Однако разработчики часто путают свойства DeclaringType и ReflectedType. Поясню различие на примере. Вот определение типа:

```
public sealed class MyType {  
    public override String ToString() { return null; }  
}
```

Что произойдет, если выполнить следующую строку кода?

```
MemberInfo[] members = typeof(MyType).GetMembers();
```

Переменная members — это ссылка на массив, в котором каждый элемент идентифицирует открытый член, определенный в типе MyType или одном из его базовых типов, например System.Object. Если запросить свойство DeclaringType для элемента MemberInfo, идентифицирующего метод ToString, оно вернет MyType, так как метод ToString объявлен или определен в типе MyType. В то же время, если запросить свойство DeclaringType для элемента

MemberInfo, идентифицирующего метод Equals, оно вернет System.Object, так как метод Equals объявлен в System.Object, а не в MyType. Свойство ReflectedType всегда возвращает MyType, поскольку этот тип был задан при вызове метода GetMembers для отражения.

Каждый элемент массива, который вернул GetMembers, — это ссылка на конкретный тип из этой иерархии (если только не задан флаг BindingFlags.DeclaredOnly). Помимо метода GetMembers, возвращающего все члены типа, Type поддерживает методы, возвращающие определенные разновидности членов: GetNestedTypes, GetFields, GetConstructors, GetMethods, GetProperties и GetEvents. Все эти методы возвращают массивы, в которых каждый элемент является ссылкой на объект Type, FieldInfo, ConstructorInfo, MethodInfo, PropertyInfo или EventInfo соответственно.

На рис. 23.2 представлена сводка типов, позволяющих приложению «пройти» по модели объектов отражения. Домен приложений (AppDomain) дает возможность узнать, какие сборки в него загружены, сборка (Assembly) — из каких модулей она состоит, а сборка (Assembly) или модуль (Module) — определяемые в них типы. В свою очередь, тип (Type) позволяет узнать все его члены (вложенные типы, конструкторы, методы, свойства и события). Пространства имен не входят в иерархию, так как они представляют собой синтаксические наборы типов. Если нужно перечислить все пространства имен, определенные в сборке, достаточно перечислить все типы в сборке и просмотреть их свойства Namespace.

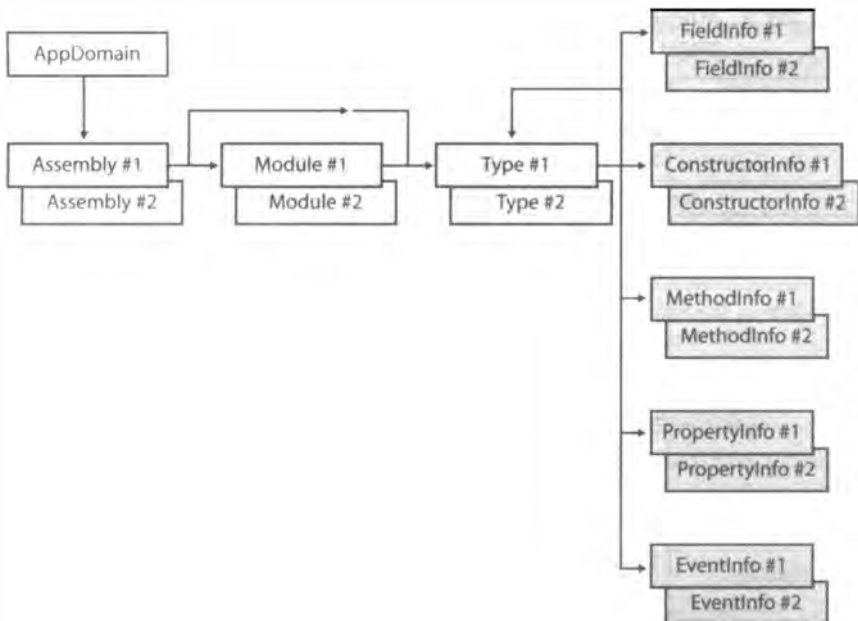


Рис. 23.2. Типы, позволяющие приложению «прогуляться» по объектной модели отражения

Тип позволяет также находить реализуемые им интерфейсы (как это сделать, я покажу позже). И из конструктора, метода, метода-аксессора свойства или метода создания/удаления сообщения можно вызвать метод `GetParameters`, чтобы получить массив объектов `ParameterInfo`, которые информируют вас о типах параметров членов. Можно также запросить свойство `ReturnParameter`, чтобы получить объект `ParameterInfo` с подробной информацией о возвращаемом значении члена. Чтобы получить набор параметров типа для обобщенных типов и методов, можно вызывать метод `GetGenericArguments`. Наконец, чтобы получить набор нестандартных атрибутов, примененных ко всем указанным сущностям, можно вызывать метод `GetCustomAttributes`.

BindingFlags: фильтрация типов возвращаемых членов

Обращаться к членам типа можно, вызывая следующие методы типа `Type`: `GetMembers`, `GetNestedTypes`, `GetFields`, `GetConstructors`, `GetMethods`, `GetProperties` и `GetEvents`. Вызывая любой из этих методов, можно передать экземпляр перечислимого типа `System.Reflection.BindingFlags`. Он содержит набор битовых флагов, объединенных оператором `OR` (ИЛИ), и служит для фильтрации членов, возвращаемых этими методами. Идентификаторы, определяемые в перечислимом типе `BindingFlags`, представлены в табл. 23.2.

Таблица 23.2. Идентификаторы поиска, определяемые в перечислимом типе `BindingFlags`

Идентификатор	Значение	Описание
Default	0x00	Используется, если не указан ни один из других флагов, перечисленных в оставшейся части таблицы
IgnoreCase	0x01	Поиск членов, имя которых совпадает с заданной строкой без учета регистра
DeclaredOnly	0x02	Поиск членов, с которыми тип был объявлен (при этом унаследованные члены игнорируются)
Instance	0x04	Поиск экземплярных членов
Static	0x08	Поиск статических членов
Public	0x10	Поиск открытых членов
NonPublic	0x20	Поиск закрытых членов
FlattenHierarchy	0x40	Поиск статических членов, определенных в базовых типах

Все перечисленные методы возвращают набор членов, у которых есть перегруженная версия, не принимающая никаких параметров. Если не передать

аргумент `BindingFlags`, все эти методы возвращают только открытые члены. Иначе говоря, по умолчанию используется значение:

```
BindingFlags.Public | BindingFlags.Instance | BindingFlags.Static
```

Обратите внимание, что в типе `Type` определены также методы `GetMember`, `GetNestedType`, `GetField`, `GetConstructor`, `GetMethod`, `GetProperty` и `GetEvent`. Они позволяют передать строку, задающую имя члена, информацию о котором надо получить. Именно в такой ситуации оказывается, кстати, флаг `IgnoreCase` типа `BindingFlags`.

Нахождение интерфейсов типа

Для получения набора интерфейсов, наследуемых типом, используют методы `FindInterfaces`, `GetInterface` и `GetInterfaces` типа `Type`. Все они возвращают объекты `Type`, представляющие интерфейс. Они сканируют иерархию наследования типа и возвращают все интерфейсы, определенные в типе, а также базовые типы.

Определить члены типа, реализованные в некотором интерфейсе, довольно сложно, так как один и тот же метод может быть определен в нескольких интерфейсах. Так, в интерфейсах `IBookRetailer` и `IMusicRetailer` может быть определен метод `Purchase`. Чтобы получить объекты `MethodInfo` для некоторого интерфейса, вызывают экземплярный метод `GetInterfaceMap` объекта `Type`, передавая ему в качестве аргумента тип интерфейса. Этот метод возвращает экземпляр `System.Reflection.InterfaceMapping` (значимый тип). В типе `InterfaceMapping` определены четыре открытых поля (табл. 23.3).

Таблица 23.3. Открытые поля типа `InterfaceMapping`

Имя поля	Тип данных	Описание
<code>TargetType</code>	<code>Type</code>	Тип, использованный для вызова <code>GetInterfaceMapping</code>
<code>InterfaceType</code>	<code>Type</code>	Интерфейсный тип, переданный методу <code>GetInterfaceMapping</code>
<code>InterfaceMethods</code>	<code>MethodInfo[]</code>	Массив, каждый элемент которого предоставляет информацию о методе интерфейса
<code>TargetMethods</code>	<code>MethodInfo[]</code>	Массив, каждый элемент которого предоставляет информацию о методе типа, на основе которого реализован соответствующий метод интерфейса

Массивы `InterfaceMethods` и `TargetMethods` симметричны, то есть элемент `InterfaceMethods[0]` идентифицирует объект `MethodInfo` интерфейса, а `TargetMethods[0]` — определенный в типе метод, реализующий этот метод

интерфейса. Вот пример кода, демонстрирующий, как нужно находить интерфейсы и интерфейсные методы по типу:

```
using System;
using System.Reflection;

// Определяем два интерфейса для тестирования
internal interface IBookRetailer : IDisposable {
    void Purchase();
    void ApplyDiscount();
}

internal interface IMusicRetailer {
    void Purchase();
}

// Этот класс реализует два интерфейса
// из этой сборки и один интерфейс,
// определенный в другой сборке
internal sealed class MyRetailer : IBookRetailer, IMusicRetailer,
    IDisposable {
    // Методы интерфейса IbookRetailer
    void IBookRetailer.Purchase() { }
    public void ApplyDiscount() { }

    // Метод интерфейса ImusicRetailer
    void IMusicRetailer.Purchase() { }

    // Метод интерфейса IDisposable
    public void Dispose() { }

    // Метод MyRetailer (не относящийся к интерфейсу)
    public void Purchase() { }
}

public static class Program {
    public static void Main() {
        // Ищем интерфейсы, реализованные в MyRetailer,
        // в которых интерфейс определен в нашей сборке
        // Это выполняется с использованием делегата по отношению
        // к фильтрующему методу, который мы создаем и передаем в FindInterfaces
        Type t = typeof(MyRetailer);
        Type[] interfaces = t.FindInterfaces(TypeFilter,
            typeof(Program).Assembly);
        Console.WriteLine("MyRetailer implements the following " +
            "interfaces (defined in this assembly):");
    }
}
```

```
// Выводим сведения о каждом интерфейсе
foreach (Type i in interfaces) {
    Console.WriteLine("\nInterface: " + i);

    // Получаем методы типа, соответствующие методам интерфейса
    InterfaceMapping map = t.GetInterfaceMap(i);
    for (Int32 m = 0; m < map.InterfaceMethods.Length; m++) {
        // Выводим имена методов интерфейса
        // и типа, в котором он реализован
        Console.WriteLine(" {0} is implemented by {1}",
            map.InterfaceMethods[m], map.TargetMethods[m]);
    }
}

// Возвращает true, если тип удовлетворяет критериям фильтрации
private static Boolean TypeFilter(Type t, Object filterCriteria) {
    // Возвращает true, если интерфейс определен в сборке,
    // определенной в filterCriteria
    return t.Assembly == filterCriteria;
}
}
```

Скомпоновав и выполнив этот код, получаем:

MyRetailer implements the following interfaces (defined in this assembly):

```
Interface: IBookRetailer
Void Purchase() is implemented by Void IBookRetailer.Purchase()
Void ApplyDiscount() is implemented by Void ApplyDiscount()
```

```
Interface: IMusicRetailer
Void Purchase() is implemented by Void IMusicRetailer.Purchase()
```

Обратите внимание, что интерфейса `IDisposable` в результатах работы программы нет, потому что он не определен в сборке EXE-файла.

Вызов членов типа

Теперь, когда мы знаем, как находить члены, определенные в типе, можно перейти к вызову этих членов. Что *конкретно* означает «вызывать», зависит от разновидности члена. Вызов `FieldInfo` позволяет получить или задать значение поля, вызов `ConstructorInfo` — создать экземпляр типа и передать параметры конструктору, вызов `MethodInfo` — вызвать метод, передать параметры и получить возвращаемое значение, вызов `PropertyInfo` — вызвать аксессор `get` или `set` свойства, а вызов `EventInfo` — создать или удалить обработчик события.

Сначала мы поговорим о вызове метода, потому что вызов этого члена самый сложный, а затем узнаем, как вызывать другие члены. В типе `Type` определен метод `InvokeMember`, который позволяет вызвать член. Существует несколько перегруженных версий `InvokeMember`, и я собираюсь рассказать о самой популярной; остальные версии работают аналогично.

```
public abstract class Type : MemberInfo. ... {
    public Object InvokeMember(
        String name,           // Имя члена
        BindingFlags invokeAttr, // Способ поиска членов
        Binder binder,         // Способ сопоставления членов и аргументов
        Object target,         // Объект, на котором нужно вызвать член
        Object[] args,         // Аргументы, которые нужно передать методу
        CultureInfo culture);   // Региональные стандарты, которые используются
                                // при связывании
    ...
}
```

Когда вы вызываете метод `InvokeMember`, он ищет среди членов типа член, соответствующий заданному. Если такого нет, вбрасывается исключение `System.MissingMethodException`, `System.MissingFieldException` или `System.MissingMemberException`, а если он найден, метод `InvokeMember` вызывает его. `InvokeMember` возвращает любое значение, которое вернет вызванный метод; если тот ничего не возвращает, `InvokeMember` возвращает `null`. Если вызываемый метод вбрасывает исключение, `InvokeMember` перехватывает его и вбрасывает исключение `System.Reflection.TargetInvocationException`, при этом в свойстве `InnerException` объекта `System.Reflection.TargetInvocationException` находится объект реального исключения, брошенного методом. Лично мне это не нравится, я бы предпочел, чтобы метод `InvokeMember` не перехватывал исключение, а дал бы ему выйти наружу.

Внутренний код `InvokeMember` выполняет две операции: выбирает подходящий для вызова член (это называется *привязкой*) и вызывает этот член (это называется *вызовом*). При вызове метода `InvokeMember` в параметре `name` ему передается строка с именем члена, к которому он должен привязаться. Но у типа может быть несколько членов с таким именем. В конце концов, даже у одного метода может быть несколько перегруженных версий, имена поля и методов тоже могут совпадать.

Естественно, методу `InvokeMember` нужно привязаться к какому-то одному члену, прежде чем он сможет его вызвать. Все параметры, передаваемые методу `InvokeMember` (кроме `target`), служат для выбора подходящего члена. Познакомимся с ними поближе.

Параметр `binder` идентифицирует объект, чей тип является потомком абстрактного типа `System.Reflection.Binder`. Тип потомка `Binder` инкапсулирует правила, которым следует `InvokeMember` при выборе члена. В базовом типе `Binder` определены абстрактные виртуальные методы `BindToField`, `BindToMethod`,

ChangeType, ReorderArgumentArray, SelectMethod и SelectProperty. Внутренний код InvokeMember вызывает эти методы через объект Binder, переданный в параметре binder.

Специалисты Microsoft определили внутренний (недокументированный) тип System.DefaultBinder — потомок Binder. Этот тип входит в FCL, и Microsoft ожидает, что им будут пользоваться практически все. Некоторые поставщики компиляторов определяют собственные типы, производные от Binder, и будут поставлять их в библиотеке времени выполнения, используемой кодом, созданным их компилятором¹. Если в метод InvokeMember передать в параметре binder значение null, метод задействует объект DefaultBinder.

При вызове методов механизмом привязки им передаются параметры, помогающие выбрать нужный член. Это, конечно же, имя искомого члена, а также флаги BindingFlags плюс все типы и параметры, которые надо передать вызываемому члену.

Ранее я перечислил значения флагов BindingFlags: Default, IgnoreCase, DeclaredOnly, Instance, Static, Public, NonPublic и FlattenHierarchy (см. табл. 23.2). Эти флаги подсказывают механизму привязки, какие члены включить в поиск.

Помимо этих флагов, механизм привязки определяет число аргументов, переданных параметром args метода InvokeMember. Число аргументов еще больше ограничивает набор возможных членов. Затем механизм привязки проверяет тип аргументов, еще больше уменьшая число возможных членов. Но когда дело доходит до типов аргумента, механизм привязки автоматически преобразует некоторые типы, чтобы получить некоторую свободу действий. Например, у типа может быть метод, принимающий единственный параметр Int64. Если вызвать InvokeMember, передав в параметре args ссылку на массив значений Int32, объект DefaultBinder все равно выберет этот метод. При вызове InvokeMember значение Int32 будет преобразовано в Int64. В табл. 23.4 перечислены преобразования, поддерживаемые объектом DefaultBinder.

Таблица 23.4. Преобразования, поддерживаемые объектом DefaultBinder

Исходный тип	Целевой тип
Любой тип	Базовый тип
Любой тип	Реализованный в типе интерфейса
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double
Byte	Char, UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double
SByte	Int16, Int32, Int64, Single, Double

продолжение ➤

¹ Программный код, представленный в данной книге, включает класс SimpleBinder, в котором демонстрируется определение собственного унаследованного от типа Binder типа.

Таблица 23.4 (продолжение)

Исходный тип	Целевой тип
UInt16	UInt32, Int32, UInt64, Int64, Single, Double
Int16	Int32, Int64, Single, Double
UInt32	UInt64, Int64, Single, Double
Int32	Int64, Single, Double
UInt64	Single, Double
Int64	Single, Double
Single	Double
Экземпляр значимого типа	Упакованная версия экземпляра значимого типа

У типа `BindingFlags` есть и другие флаги, которые служат для точной настройки `DefaultBinder` (табл. 23.5).

Таблица 23.5. Флаги `BinderFlags`, используемые объектом `DefaultBinder`

Идентификатор	Значение	Описание
<code>ExactBinding</code>	<code>0x010000</code>	Механизм привязки будет искать член с параметрами, соответствующими типу переданных аргументов
<code>OptionalParamBinding</code>	<code>0x040000</code>	Механизм привязки будет рассматривать любой член, у которого число параметров совпадает с числом переданных аргументов. Этот флаг удобен при наличии членов с параметрами, для которых заданы значения по умолчанию, и методов с переменным числом аргументов. Это флаг учитывается только методом <code>InvokeMember</code> объекта <code>Type</code>

Последний параметр метода `InvokeMember`, `culture`, также служит для привязки. Однако тип `DefaultBinder` игнорирует его. Определив собственный механизм привязки, можно использовать `culture` как вспомогательный параметр для преобразования типов аргументов. Скажем, вызывающий код может передать аргумент `String` со значением `1,23`. Механизм привязки проверяет эту строку, выполняет ее синтаксический разбор с учетом региональных стандартов, заданных параметром `culture`, и преобразует тип аргумента в `Single` (если параметр `culture` задан как `de-DE`) или оставит его, как есть (если параметр `culture` задан как `en-US`).

Мы рассмотрели почти все параметры `InvokeMember`, касающиеся привязки, — осталось обсудить только параметр `target`. Он представляет собой ссылку на объект, чей метод нужно вызвать. Если нужно вызвать статический метод объекта `Type`, в этом параметре следует передать `null`.

Метод `InvokeMember` очень мощный. Он позволяет вызывать методы (о чем мы уже говорили), создавать экземпляры типа (обычно путем вызова его конструктора), а также получать и определять значения полей. Чтобы сообщить `InvokeMember`, какое из этих действий нужно выполнить, применяется один из флагов `BindingFlags` (табл. 23.6).

Таблица 23.6. Флаги `BindingFlags`, используемые методом `InvokeMember`

Идентификатор	Значение	Описание
<code>InvokeMethod</code>	<code>0x0100</code>	Заставляет <code>InvokeMember</code> вызвать метод
<code>CreateInstance</code>	<code>0x0200</code>	Заставляет <code>InvokeMember</code> создать новый объект, вызвав его конструктор
<code>GetField</code>	<code>0x0400</code>	Заставляет <code>InvokeMember</code> получить значение поля
<code>SetField</code>	<code>0x0800</code>	Заставляет <code>InvokeMember</code> установить значение поля
<code>GetProperty</code>	<code>0x1000</code>	Заставляет <code>InvokeMember</code> вызвать метод-аксессор <code>get</code> свойства <code>SetProperty</code>
<code>SetProperty</code>	<code>0x2000</code>	Заставляет <code>InvokeMember</code> вызвать метод-аксессор <code>set</code> свойства

Большинство этих флагов являются взаимоисключающими: при вызове `InvokeMember` может быть указан один и только один из них. Однако можно одновременно указать флаги `GetField` и `GetProperty`, тогда метод `InvokeMember` будет искать сначала поле, а если не найдет его, — подходящее свойство. Аналогично можно определить и сопоставить флаги `SetField` и `SetProperty`. Механизм привязки использует эти флаги для сужения круга возможных кандидатов. Если определить флаг `BindingFlags.CreateInstance`, механизм привязки будет знать, что вправе выбрать только метод-конструктор.

ВНИМАНИЕ

Может показаться, что отражение позволяет без труда привязаться к внутреннему члену и вызвать его, что дает коду приложения возможность обращаться к закрытым членам, к которым компилятор обычно запрещает доступ. Однако система безопасности доступа к коду не дает злоупотреблять всей силой отражения.

Когда вы вызываете метод для привязки к члену, этот метод сначала проверяет, будет ли член, к которому вы пытаетесь привязаться, видимым для вас при компиляции. Если да, то привязка оказывается успешной. Если этот член в обычных обстоятельствах вам недоступен, метод требует разрешение `System.Security.Permissions.ReflectionPermission`, проверяя, установлен ли бит `TypeInfo` флага `System.Security.Permissions.ReflectionPermissionFlags`. Если да, метод привязывается к члену, а если

запрос заканчивается неудачей, вбрасывается исключение `System.Security.SecurityException`.

Если вы обращаетесь к методу для вызова члена, то этот метод производит ту же проверку, что и при привязке к члену. Однако в этом случае он проверяет бит `MemberAccess` флага `ReflectionPermissionFlag`. Если он установлен, член вызывается, в противном случае вбрасывается исключение `SecurityException`.

Конечно, если ваша сборка пользуется полным доверием, проверки системы безопасности будут успешно пройдены, сборка получит разрешение и вызов успешно выполнится. Тем не менее отражение никогда, ни при каких обстоятельствах нельзя использовать для доступа к какому-либо недokumentированному члену типа, так как после выхода следующей версии сборки ваш код перестанет работать.

Один раз привяжись, семь раз вызови

Метод `InvokeMember` типа `Type` предоставляет доступ к любым членам типа (за исключением событий). Однако следует знать, что при каждом вызове метода `InvokeMember` нужно сначала привязаться к некоторому члену и только потом его можно вызвать. Если механизм привязки будет при каждом вызове выбирать подходящий член, на это уйдет много времени, и если делать это часто, снизится быстродействие приложения. Поэтому если планируется часто обращаться к некоторому члену, лучше привязаться к нему один раз, а после этого вызывать его столько, сколько нужно.

Мы уже говорили о том, как привязаться к члену, вызвав один из методов типа `Type`: `GetFields`, `GetConstructors`, `GetMethods`, `GetProperties`, `GetEvents` или любой им подобный. Все они возвращают ссылку на объект, тип которого предлагает методы для прямого доступа к некоторому члену. Типы и методы, вызываемые для доступа к члену, перечислены в табл. 23.7.

Таблица 23.7. Типы и методы для доступа к члену после привязки к нему

Тип	Описание
FieldInfo	Метод <code>GetValue</code> позволяет получить значение поля, метод <code>SetValue</code> — задать его значение
ConstructorInfo	Метод <code>Invoke</code> позволяет создать экземпляр типа и вызвать конструктор
MethodInfo	Метод <code>Invoke</code> позволяет вызвать метода типа
PropertyInfo	Метод <code>GetValue</code> позволяет вызвать аксессор <code>get</code> , метод <code>SetValue</code> — аксессор <code>set</code>
EventInfo	Метод <code>AddEventHandler</code> позволяет вызвать аксессор <code>add</code> события, метод <code>RemoveEventHandler</code> — аксессор <code>remove</code> этого события

Тип `PropertyInfo` предоставляет информацию метаданных свойств (см. главу 10), поддерживая доступные только для чтения свойства `CanRead`, `CanWrite`

и `PropertyType`. Эти свойства показывают, можно ли читать и записывать свойство, а также его тип данных. У `PropertyInfo` есть метод `GetAccessors`, возвращающий массив элементов `MethodInfo` — один для аксессуора `get` (если он существует), второй для аксессуора `set` (если он существует). Более полезные методы `GetMethod` и `SetMethod` этого типа, каждый из которых возвращает только один объект `MethodInfo`. Методы `GetValue` и `SetValue` типа `PropertyInfo` существуют для удобства, их внутренний код получает соответствующие объекты `MethodInfo` и вызывает их. Для поддержки параметрических свойств (индексатора C#) методы `SetValue` и `GetValue` предоставляют параметр `index` типа `Object []`.

Тип `EventInfo` представляет информацию о метаданных событиях (см. главу 11), поддерживая доступное только для чтения свойство `EventHandlerType`, которое возвращает объект `Type` для делегата события. У `EventInfo` также есть методы `AddMethod` и `RemoveMethod`, которые возвращают соответствующие объекты `MethodInfo`. Методы `AddEventHandler` и `RemoveEventHandler` существуют для удобства, их внутренний код получает нужные объекты `MethodInfo` и вызывает их.

Вызывая один из методов, перечисленных в правом столбце табл. 23.7, вы не привязываетесь к члену, а только вызываете его. Любой из этих методов можно вызывать многократно, а поскольку они не требуют привязки, производительность при этом не снижается.

Возможно, вы заметили, что метод `Invoke` типов `ConstructorInfo` и `MethodInfo`, а также методы `GetValue` и `SetValue` типа `PropertyInfo` поддерживают перепрыгнутые версии, принимающие ссылки на объект-потомок `Binder` и некоторые флаги `BindingFlags`. Можно подумать, что эти методы привязываются к члену, но это не так.

При вызове одного из этих методов объект-потомок `Binder` служит для преобразования типов, например из `Int32` в `Int64`, чтобы вызвать уже выбранный метод. Как и в случае параметра `BindingFlags`, здесь можно передать лишь флаг `BindingFlags.SuppressChangeType`. Механизмы привязки могут игнорировать этот флаг, но `DefaultBinder` так не поступает. Обнаружив данный флаг, `DefaultBinder` не преобразует переданные ему аргументы, а если они не соответствуют тем, которые ожидает метод, вбрасывается исключение `ArgumentException`.

Обычно, если для привязки к члену использован флаг `BindingFlags.ExactBinding`, то при вызове этого члена устанавливают флаг `BindingFlags.SuppressChangeType`. Если не использовать эти флаги в паре, то вызов этого члена окажется неудачным, если только переданные аргументы не совпадают в точности с аргументами, ожидаемыми методом. Кстати, если для привязки к члену и его последующего вызова используют метод `InvokeMethod` объекта `MemberInfo`, то обычно указывают либо оба этих флага, либо ни одного.

В следующем далее приложении-примере демонстрируются разные способы применения отражения для доступа к членам типа. Класс `SomeType` представляет

тип с различными членами: закрытым полем (`m_someField`), открытым конструктором (`SomeType`), передающим аргумент типа `Int32` по ссылке, открытым методом (`SomeProp`) и открытым событием (`SomeEvent`). Определен тип `SomeType`, и я также могу предложить три разных метода использования отражения для доступа к членам типу `SomeType`. Каждый метод задействует отражение по-своему.

- ❑ Метод `UseInvokeMemberToBindAndInvokeTheMember` демонстрирует использование метода `InvokeMember` для связывания и вызова члена.
- ❑ Метод `BindToMemberThenInvokeTheMember` демонстрирует связывание члена и его последующий вызов. Этот вариант помогает написанию производительного программного кода для случаев многократного вызова одинаковых членов разными объектами.
- ❑ Метод `BindToMemberCreateDelegateToMemberThenInvokeTheMember` демонстрирует связывание объекта или члена, а затем создание делегата, ссылающегося на этот объект или член. Вызов через делегата является очень быстрым, и этот вариант позволяет еще больше повысить производительность программного кода для случаев многократного вызова одинаковых членов разными объектами.
- ❑ Метод `UseDynamicToBindAndInvokeTheMember` демонстрирует использование в языке C# примитивного типа `dynamic` (см. главу 5) с целью упрощения синтаксиса доступа к членам. К тому же этот вариант может помочь добиться действительно хорошей производительности программного кода для случаев вызова одинаковых членов разными объектами, потому что связывание происходит один раз для каждого типа и затем кэшируется таким образом, чтобы последующий многократный вызов членов происходил быстро. Вы также можете использовать этот вариант с целью вызова членов для объектов различных типов.

```
using System;  
using System.Reflection;  
using Microsoft.CSharp.RuntimeBinder;
```

```
// Это класс для демонстрации отражения  
// У него есть поле, конструктор, метод, свойство и событие  
internal sealed class SomeType {  
    private Int32 m_someField;  
    public SomeType(ref Int32 x) { x *= 2; }  
    public override String ToString() { return m_someField.ToString(); }  
    public Int32 SomeProp {  
        get { return m_someField; }  
        set {  
            if (value < 1)  
                throw new ArgumentOutOfRangeException("value");  
        }  
    }  
}
```

```

    m_someField = value;
}
public event EventHandler SomeEvent;
private void NoCompilerWarnings() { SomeEvent.ToString();}
}

public static class Program {
    private const BindingFlags c_bf = BindingFlags.DeclaredOnly |
        BindingFlags.Public |
        BindingFlags.NonPublic | BindingFlags.Instance;

    public static void Main() {
        Type t = typeof(SomeType);
        UseInvokeMemberToBindAndInvokeTheMember(t);
        Console.WriteLine();

        BindToMemberThenInvokeTheMember(t);
        Console.WriteLine();

        BindToMemberCreateDelegateToMemberThenInvokeTheMember(t);
        Console.WriteLine();

        UseDynamicToBindAndInvokeTheMember(t);
        Console.WriteLine();
    }

    private static void UseInvokeMemberToBindAndInvokeTheMember(Type t) {
        Console.WriteLine("UseInvokeMemberToBindAndInvokeTheMember");

        // Создание экземпляра Type
        Object[] args = new Object[] { 12 }; // Аргументы конструктора
        Console.WriteLine("x before constructor called: " + args[0]);
        Object obj = t.InvokeMember(null, c_bf | BindingFlags.CreateInstance,
            null, null, args);
        Console.WriteLine("Type: " + obj.GetType().ToString());
        Console.WriteLine("x after constructor returns: " + args[0]);

        // Чтение поля и запись в поле
        t.InvokeMember("m_someField", c_bf | BindingFlags.SetField,
            null, obj, new Object[] { 5 });
        Int32 v = (Int32)t.InvokeMember("m_someField", c_bf |
            BindingFlags.GetField,
            null, obj, null);
        Console.WriteLine("someField: " + v);
    }
}

```

```

// Вызов метода
String s = (String)
t.InvokeMember("ToString", c_bf | BindingFlags.InvokeMethod, null, obj,
    null);
Console.WriteLine("ToString: " + s);

// Чтение и запись свойства
try {
    t.InvokeMember("SomeProp", c_bf | BindingFlags.SetProperty,
        null, obj, new Object[] { 0 });
}
catch (TargetInvocationException e) {
    if (e.InnerException.GetType() != typeof(ArgumentOutOfRangeException)
        throw;
    Console.WriteLine("Property set catch.");
}
t.InvokeMember("SomeProp", c_bf | BindingFlags.SetProperty,
    null, obj, new Object[] { 2 });
v = (Int32)t.InvokeMember("SomeProp", c_bf | BindingFlags.GetProperty,
    null, obj, null);
Console.WriteLine("SomeProp: " + v);

// Добавление в событие и удаление из события делегата
// путем вызова соответствующего метода
EventHandler eh = new EventHandler(EventCallback);
t.InvokeMember("add_SomeEvent", c_bf | BindingFlags.InvokeMethod,
    null, obj, new Object[] { eh });
t.InvokeMember("remove_SomeEvent", c_bf | BindingFlags.InvokeMethod,
    null, obj, new Object[] { eh });
}

private static void BindToMemberThenInvokeTheMember(Type t) {
    Console.WriteLine("BindToMemberThenInvokeTheMember");

    // Создание экземпляра
    // ConstructorInfo ctor =
    // t.GetConstructor(new Type[] { Type.GetType("System.Int32&") });
    ConstructorInfo ctor = t.GetConstructor(new Type[] {
        typeof(Int32).MakeByRefType() });
    Object[] args = new Object[] { 12 }; // Аргументы конструктора.
    Console.WriteLine("x before constructor called: " + args[0]);
    Object obj = ctor.Invoke(args);
    Console.WriteLine("Type: " + obj.GetType().ToString());
    Console.WriteLine("x after constructor returns: " + args[0]);

    // Чтение поля и запись в поле
    FieldInfo fi = obj.GetType().GetField("m_someField", c_bf);

```

```
fi.SetValue(obj, 33);
Console.WriteLine("someField: " + fi.GetValue(obj));

// Вызов метода
MethodInfo mi = obj.GetType().GetMethod("ToString", c_bf);
String s = (String)mi.Invoke(obj, null);
Console.WriteLine("ToString: " + s);

// Чтение и запись свойства
PropertyInfo pi = obj.GetType().GetProperty("SomeProp",
    typeof(Int32));
try {
    pi.SetValue(obj, 0, null);
}
catch (TargetInvocationException e) {
    if (e.InnerException.GetType() !=
        typeof(ArgumentOutOfRangeException)) throw;
    Console.WriteLine("Property set catch.");
}
pi.SetValue(obj, 2, null);
Console.WriteLine("SomeProp: " + pi.GetValue(obj, null));

// Добавление делегата в событие и удаление делегата из события
EventInfo ei = obj.GetType().GetEvent("SomeEvent", c_bf);
EventHandler ts = new EventHandler(EventCallback);

// См. ei.EventHandlerType
ei.AddEventHandler(obj, ts);
ei.RemoveEventHandler(obj, ts);
}

// Метод обратного вызова, добавленный в событие
private static void EventCallback(Object sender, EventArgs e) { }

private static void
BindToMemberCreateDelegateToMemberThenInvokeTheMember(Type t) {
    Console.WriteLine("BindToMemberCreateDelegateToMemberThenInvokeTheMember");

    // Создание экземпляра (нельзя создать делегата в конструкторе)
    Object[] args = new Object[] { 12 }; // Аргументы конструктора
    Console.WriteLine("x before constructor called: " + args[0]);
    Object obj = Activator.CreateInstance(t, args);
    Console.WriteLine("Type: " + obj.GetType().ToString());
    Console.WriteLine("x after constructor returns: " + args[0]);

    // ПРИМЕЧАНИЕ. Вы не можете создать делегата в поле
```

```

// Вызов метода
MethodInfo mi = obj.GetType().GetMethod("ToString", c_bf);
var toString = (Func<String>)
    Delegate.CreateDelegate(typeof(Func<String>), obj, mi);
String s = toString();
Console.WriteLine("ToString: " + s);

// Чтение и запись свойства
PropertyInfo pi = obj.GetType().GetProperty("SomeProp",
    typeof(Int32));
var setSomeProp = (Action<Int32>)
    Delegate.CreateDelegate(typeof(Action<Int32>), obj,
    pi.GetSetMethod());
try {
    setSomeProp(0);
}
catch (ArgumentOutOfRangeException) {
    Console.WriteLine("Property set catch.");
}
setSomeProp(2);
var getSomeProp = (Func<Int32>)
    Delegate.CreateDelegate(typeof(Func<Int32>), obj, pi.GetGetMethod());
Console.WriteLine("SomeProp: " + getSomeProp());

// Добавление делегата в событие и удаление делегата из события
EventInfo ei = obj.GetType().GetEvent("SomeEvent", c_bf);
var addSomeEvent = (Action<EventHandler>)
    Delegate.CreateDelegate(typeof(Action<EventHandler>), obj,
    ei.GetAddMethod());
addSomeEvent(EventCallback);
var removeSomeEvent = (Action<EventHandler>)
    Delegate.CreateDelegate(typeof(Action<EventHandler>), obj,
    ei.GetRemoveMethod());
removeSomeEvent(EventCallback);
}

private static void UseDynamicToBindAndInvokeTheMember(Type t) {
    Console.WriteLine("UseDynamicToBindAndInvokeTheMember");

    // Создание экземпляра (нельзя создать делегата в конструкторе)
    Object[] args = new Object[] { 12 }; // Аргументы конструктора
    Console.WriteLine("x before constructor called: " + args[0]);
    dynamic obj = Activator.CreateInstance(t, args);
    Console.WriteLine("Type: " + obj.GetType().ToString());
    Console.WriteLine("x after constructor returns: " + args[0]);

    // Чтение поля и запись в поле
    try {

```

```

    obj.m_someField = 5;
    Int32 v = (Int32)obj.m_someField;
    Console.WriteLine("someField: " + v);
}
catch (RuntimeBinderException e) {
    // Мы попадаем сюда, если поле закрыто
    Console.WriteLine("Failed to access field: " + e.Message);
}

// Вызов метода
String s = (String)obj.ToString();
Console.WriteLine("ToString: " + s);

// Чтение и запись свойства
try {
    obj.SomeProp = 0;
}
catch (ArgumentOutOfRangeException) {
    Console.WriteLine("Property set catch.");
}
obj.SomeProp = 2;
Int32 val = (Int32)obj.SomeProp;
Console.WriteLine("SomeProp: " + val);

// Добавление делегата в событие и удаление делегата из события
obj.SomeEvent += new EventHandler(EventCallback);
obj.SomeEvent -= new EventHandler(EventCallback);
}
}

```

Вот что получится, если скомпоновать и запустить этот код:

```

UseInvokeMemberToBindAndInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
someField: 5
ToString: 5
Property set catch.
SomeProp: 2

```

```

BindToMemberThenInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
someField: 33
ToString: 33

```

```
Property set catch.  
SomeProp: 2
```

```
BindToMemberCreateDelegateToMemberThenInvokeTheMember  
x before constructor called: 12  
Type: SomeType  
x after constructor returns: 24  
ToString: 0  
Property set catch.  
SomeProp: 2
```

```
UseDynamicToBindAndInvokeTheMember  
x before constructor called: 12  
Type: SomeType  
x after constructor returns: 24  
Failed to access field: 'SomeType.m_someField' is inaccessible due to its  
protection level  
ToString: 0  
Property set catch.  
SomeProp: 2
```

Обратите внимание, что конструктор `SomeType` принимает в качестве единственного параметра ссылку на `Int32`. В представленном коде показано, как вызвать этот конструктор и как после завершения конструктора проверить модифицированное значение `Int32`. Далее в начале метода `BindingToMemberFirstAndThenInvokingTheMember` есть вызов метода `GetType` типа `Type`, которому передается строка `"System.Int32&"`. Знак амперсанда (&) в строке позволяет указать на параметр, передаваемый по ссылке. Это предусмотрено нотацией Бэкуса–Наура для записи имен типов (подробнее о ней см. документацию на FCL). Эта строка программного кода может быть записана также в следующем виде:

```
ConstructorInfo ctor = t.GetConstructor(new Type[] {  
    typeof(Int32).MakeByRefType() });
```

Использование описателей привязки для снижения потребления памяти процессом

Во многих приложениях требуется привязка к нескольким типам (то есть объектам `Type`) или их членам (объектам, производным от `MemberInfo`), а эти объекты сохраняются в определенной коллекции. Позже приложение ищет нужный объект в коллекции и вызывает его. Это разумное решение, но есть одна загвоздка: объекты `Type` и объекты-потомки `MemberInfo` занимают много места в памяти. Поэтому если в приложении много таких объектов и к ним надо обращаться часто, объем потребляемой памяти резко возрастает, что отрицательно сказывается на производительности.

Внутренние механизмы CLR поддерживают более компактную форму хранения этой информации. CLR создает такие объекты в приложениях лишь для того, чтобы упростить работу программиста. Самой среде CLR для работы эти большие объекты не нужны. В приложениях, в которых сохраняется и кэшируется много объектов `Type` и объектов-потомков `MemberInfo`, можно сократить потребление памяти, если использовать не объекты, а описатели времени выполнения. В FCL определены три типа таких описателей (все в пространстве имен `System`): `RuntimeTypeHandle`, `RuntimeFieldHandle` и `RuntimeMethodHandle`. Все они — значимые типы с единственным полем `IntPtr`; за счет чего расходуют очень мало ресурсов (то есть памяти). Поле `IntPtr` представляет собой описатель, ссылающийся на тип, поле или метод в куче загрузчика домена приложений. Так что теперь нам достаточно научиться просто и эффективно преобразовывать «тяжелые» объекты `Type` и `MemberInfo` в «легкие» описатели времени выполнения, и наоборот. Это не сложно, если задействовать перечисленные далее методы и свойства.

- ❑ Чтобы преобразовать объект `Type` в `RuntimeTypeHandle`, вызовите статический метод `GetTypeHandle` объекта `Type`, передав ему ссылку на объект `Type`.
- ❑ Чтобы преобразовать `RuntimeTypeHandle` в объект `Type`, вызовите статический метод `GetTypeFromHandle` объекта `Type`, передав ему `RuntimeTypeHandle`.
- ❑ Чтобы преобразовать объект `FieldInfo` в `RuntimeFieldHandle`, запросите экземплярное неизменяемое свойство `FieldHandle` объекта `FieldInfo`.
- ❑ Чтобы преобразовать `RuntimeTypeHandle` в объект `FieldInfo`, вызовите статический метод `GetTypeFromHandle` объекта `FieldInfo`.
- ❑ Чтобы преобразовать объект `MethodInfo` в `RuntimeMethodHandle`, запросите экземплярное неизменяемое свойство `MethodHandle` объекта `MethodInfo`.
- ❑ Чтобы преобразовать `RuntimeTypeHandle` в объект `MethodInfo`, вызовите статический метод `GetMethodFromHandle` объекта `MethodInfo`.

В приводимом далее примере программы создается много объектов `MethodInfo`, которые преобразуются в экземпляры `RuntimeMethodHandle`, а затем выводится информация о разнице в объеме потребляемой памяти:

```
using System;
using System.Reflection;
using System.Collections.Generic;
```

```
public sealed class Program {
    private const BindingFlags c_bf = BindingFlags.FlattenHierarchy |
        BindingFlags.Instance |
        BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic;
```



```

public static void Main() {
    // Выводим размер кучи до отражения
    Show("Before doing anything");

    // Создаем кэш объектов MethodInfo для всех методов из MSCorlib.dll
    List<MethodBase> methodInfos = new List<MethodBase>();
    foreach (Type t in typeof(Object).Assembly.GetExportedTypes()) {
        // Игнорируем обобщенные типы
        if (t.IsGenericTypeDefinition) continue;

        MethodBase[] mb = t.GetMethods(c_bf);
        methodInfos.AddRange(mb);
    }

    // Выводим число методов и размер кучи после привязки всех методов
    Console.WriteLine("# of methods={0:N0}", methodInfos.Count);
    Show("After building cache of MethodInfo objects");

    // Создаем кэш описателей RuntimeMethodHandles
    // для всех объектов MethodInfo
    List<RuntimeMethodHandle> methodHandles =
        methodInfos.ConvertAll<RuntimeMethodHandle>(mb => mb.MethodHandle);
    Show("Holding MethodInfo and RuntimeMethodHandle cache");
    GC.KeepAlive(methodInfos); // Запрещаем сборку мусора в кэше

    methodInfos = null; // Разрешаем сборку мусора в кэше
    Show("After freeing MethodInfo objects");

    methodInfos = methodHandles.ConvertAll<MethodBase>(
        rmh=> MethodBase.GetMethodFromHandle(rmh));
    Show("Size of heap after re-creating MethodInfo objects");
    GC.KeepAlive(methodHandles); // Запрещаем сборку мусора в кэше
    GC.KeepAlive(methodInfos); // Запрещаем сборку мусора в кэше

    methodHandles = null; // Разрешаем сборку мусора в кэше
    methodInfos = null; // Разрешаем сборку мусора в кэше
    Show("After freeing MethodInfo and RuntimeMethodHandles");
}
}

```

Вот что получается, если скомпоновать и запустить этот код:

```

Heap size= 85,000 - Before doing anything
# of methods=48,467
Heap size= 7,065,632 - After building cache of MethodInfo objects
Heap size= 7,453,496 - Holding MethodInfo and RuntimeMethodHandle cache
Heap size= 6,732,704 - After freeing MethodInfo objects
Heap size= 7,372,704 - Size of heap after re-creating MethodInfo objects
Heap size= 192,232 - After freeing MethodInfo and RuntimeMethodHandles

```

Глава 24. Сериализация

Сериализацией (serialization) называется процесс преобразования объекта или графа связанных объектов в поток байтов. Соответственно, обратное преобразование называется *десериализацией* (deserialization). Вот примеры применения этого удивительно полезного механизма:

- ❑ Состояние приложения (граф объекта) можно легко сохранить в файле на диске или в базе данных и восстановить при следующем запуске приложения. ASP.NET сохраняет и восстанавливает состояние сеанса путем сериализации и десериализации.
- ❑ Набор объектов можно скопировать в буфер и вставить в то же или в другое приложение. Этот подход используется в приложениях Windows Forms и Windows Presentation Foundation (WPF).
- ❑ Можно клонировать набор объектов и сохранить как «резервную копию», пока пользователь работает с «основным» набором объектов.
- ❑ Набор объектов можно легко передать по сети в процесс, запущенный на другой машине. Механизм удаленного взаимодействия платформы .NET Framework сериализует и десериализует объекты, продвигаемые по значению. Эта же технология используется при передаче объектов через границы домена (см. главу 22).

Помимо сказанного, можно отметить, что после сериализации объектов в поток байтов в памяти появляется возможность шифрования и сжатия данных.

Не удивительно, что в рамках поддержания столь полезной функциональности работают многие программисты. Однако при этом соответствующий код сложно и мучительно писать, к тому же он подвержен ошибкам. Разработчикам приходится решать проблемы взаимодействия протоколов, несовпадения типов данных клиента и сервера (например, разный порядок следования байтов), обработки ошибок, ссылок одних объектов на другие, параметров in и out, массивов структур — и этот список можно продолжать бесконечно.

Впрочем, в .NET Framework существует встроенный механизм сериализации и десериализации. Это означает, что все упомянутые сложные проблемы уже решены средствами .NET Framework. Разработчик может использовать объекты до сериализации и после десериализации, а всю заботу о том, что происходит между этими двумя процедурами, на себя берет .NET Framework.

В этой главе рассказывается, как реализованы сериализация и десериализация в .NET Framework. Эти процедуры определены практически для всех типов данных. То есть вам не придется предпринимать дополнительных усилий, чтобы сделать свои типы сериализуемыми. Впрочем, существуют и типы, для кото-

рых подобная предварительная подготовка необходима. К счастью, процедуры сериализации допускают расширение, и мы детально рассмотрим данный процесс, позволяющий выполнять различные операции, сериализуя и десериализуя объекты. К примеру, я покажу, как сериализовав одну версию объекта в файл на диске, десериализовать его потом в другую версию.

ПРИМЕЧАНИЕ

В этой главе в основном рассматривается технология сериализации в среде CLR, которая хорошо распознает типы данных CLR и умеет сериализовать поля объектов, помеченные модификаторами `public`, `protected`, `internal` и даже `private`, превращая их в сжатый двоичный поток и тем самым повышая производительность. Для сериализации типов данных CLR в поток XML требуется класс `System.Runtime.Serialization.NetDataContractSerializer`. Платформа .NET Framework предлагает и другие технологии сериализации, разработанные для взаимодействия между CLR-совместимыми и CLR-несовместимыми типами данных. В них используются классы `System.Xml.Serialization.XmlSerializer` и `System.Runtime.Serialization.DataContractSerializer`.

Краткое руководство по сериализации/десериализации

Рассмотрим такой код:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

internal static class QuickStart {
    public static void Main() {
        // Создание графа объектов для последующей сериализации в поток
        var objectGraph = new List<String> {
            "Jeff", "Kristin", "Aidan", "Grant" };
        Stream stream = SerializeToMemory(objectGraph);

        // Обнуляем все для данного примера
        stream.Position = 0;
        objectGraph = null;

        // Десериализация объектов и проверка их работоспособности
        objectGraph = (List<String>) DeserializeFromMemory(stream);
        foreach (var s in objectGraph) Console.WriteLine(s);
    }
}
```

```
private static MemoryStream SerializeToMemory(Object objectGraph) {  
    // Конструирование потока, который будет содержать  
    // сериализованные объекты  
    MemoryStream stream = new MemoryStream();  
    // Задание форматирования при сериализации  
    BinaryFormatter formatter = new BinaryFormatter();  
  
    // Заставляем модуль форматирования сериализовать объекты в поток  
    formatter.Serialize(stream, objectGraph);  
  
    // Возвращение потока сериализованных объектов вызывающему методу  
    return stream;  
}  
  
private static Object DeserializeFromMemory(Stream stream) {  
    // Задание форматирования при сериализации  
    BinaryFormatter formatter = new BinaryFormatter();  
  
    // Заставляем модуль форматирования десериализовать объекты из потока  
    return formatter.Deserialize(stream);  
}  
}
```

Видите, как просто! Метод `SerializeToMemory` создает объект `System.IO.MemoryStream`. Этот объект определяет, куда следует поместить сериализованный блок байтов. Затем метод конструирует объект `BinaryFormatter` (расположенный в пространстве имен `System.Runtime.Serialization.Formatters.Binary`). *Модулем форматирования* (*formatter*) называется тип (он реализует интерфейс `System.Runtime.Serialization.IFormatter`), умеющий сериализовать и десериализовать граф объекта. В библиотеке FCL имеются два модуля форматирования: `BinaryFormatter` (используется в показанном фрагменте кода) и `SoapFormatter` (находящийся в пространстве имен `System.Runtime.Serialization.Formatters.Soap` и реализованный в сборке `System.Runtime.Serialization.Formatters.Soap.dll`).

ПРИМЕЧАНИЕ

Начиная с версии 3.5, в .NET Framework класс `SoapFormatter` считается устаревшим и не рекомендуется к использованию. Однако его имеет смысл применять при отладке кода сериализации, так как он создает доступный для чтения текст в формате XML. Если в выходном коде вы хотите воспользоваться механизмами XML-сериализации и XML-десериализации, обратитесь к классам `XmlSerializer` и `DataContractSerializer`.

Для сериализации графа объектов достаточно вызвать метод `Serialize` модуля форматирования и передать ему, во-первых, ссылку на объект потока

ввода-вывода, во-вторых, ссылку на сериализуемый граф. Поток ввода-вывода указывает, куда следует поместить сериализуемые байты. Его роль может играть объект любого типа, производного от абстрактного базового класса `System.IO.Stream`. То есть граф может быть сериализован в тип `MemoryStream`, `FileStream`, `NetworkStream` и т. п.

Вторым параметром метода `Serialize` является ссылка на объект любого типа: `Int32`, `String`, `DateTime`, `Exception`, `List<String>`, `Dictionary<Int32, DateTime>` и т. п. Объект, на который ссылается параметр `objectGraph`, может, в свою очередь, являться ссылкой. К примеру, параметр `objectGraph` может ссылаться на коллекцию ссылок на набор объектов. А им, в свою очередь, ничто не мешает ссылаться на другие объекты. Метод `Serialize` сериализует в поток все объекты графа.

Модуль форматирования «знает», как сериализовать весь граф, ссылаясь на описывающие тип каждого объекта метаданные. Для отслеживания состояния экземплярных полей в типе каждого объекта в процессе сериализации метод `Serialize` использует отражение. Если какие-то из полей ссылаются на другие объекты, метод сериализует и их.

Модули форматирования снабжены очень умным алгоритмом. Они умеют сериализовать каждый объект в графе всего один раз. Благодаря этому при перекрестной ссылке двух объектов модуль форматирования не входит в бесконечный цикл.

В моем методе `SerializeToMemory` при возвращении управления методом `Serialize` объект `MemoryStream` просто возвращается вызывающему коду. Приложение использует содержимое этого неструктурированного массива байтов тем способом, который считает необходимым. Например, оно может сохранить массив в файле, скопировать в буфер обмена, переслать по сети и т. п.

Метод `DeserializeFromStream` превращает поток ввода-вывода обратно в граф объекта. Эта операция еще проще сериализации. В моем коде был сконструирован объект `BinaryFormatter`, после чего оставалось вызвать его метод `Deserialize`. Этот метод берет в качестве параметра поток ввода-вывода и возвращает ссылку на корень десериализованного графа.

При этом внутри метод `Deserialize` исследует содержимое потока, создает экземпляры всех обнаруженных в потоке объектов и инициализирует все их, присваивая им значения, которые граф объекта имел до сериализации. Обычно затем следует приведение ссылки на возвращенный методом `Deserialize` объект к типу, который ожидает увидеть приложение.

ПРИМЕЧАНИЕ

Вот пример забавного и полезного метода, использующего сериализацию для создания детальной копии, или клона, объекта:

```
private static Object DeepClone(Object original) {
    // Создание временного потока в памяти
    using (MemoryStream stream = new MemoryStream()) {
```

```
// Создания модуля форматирования для сериализации
BinaryFormatter formatter = new BinaryFormatter();

// Эта строка описывается
// в разделе "Контексты потока ввода-вывода"
formatter.Context =
    new StreamingContext(StreamingContextStates.Clone);

// Сериализация графа объекта в поток в памяти
formatter.Serialize(stream, original);

// Возвращение к началу потока в памяти перед десериализацией
stream.Position = 0;

// Десериализация графа в новый набор объектов и возвращение
// корня графа (детальной копии) вызывающему методу
return formatter.Deserialize(stream);
}
}
```

Здесь я хотел бы добавить несколько примечаний. Во-первых, следует следить за тем, чтобы сериализация и десериализация производилась одним и тем же модулем форматирования. К примеру, недопустим код, в котором сериализация графа объекта производится модулем `SoapFormatter`, в то время как десериализацию осуществляет уже `BinaryFormatter`. Если метод `Deserialize` не в состоянии расшифровать содержимое потока, вбрасывается исключение `System.Runtime.Serialization.SerializationException`.

Во-вторых, хотелось бы упомянуть о возможности и полезности сериализации набора графов объектов в единый поток. Например, пусть у нас есть два определения классов:

```
[Serializable] internal sealed class Customer { /* ... */ }
[Serializable] internal sealed class Order { /* ... */ }
```

Тогда в основном классе нашего приложения мы можем определить следующие статические поля:

```
private static List<Customer> s_customers = new List<Customer>();
private static List<Order> s_pendingOrders = new List<Order>();
private static List<Order> s_processedOrders = new List<Order>();
```

Теперь при помощи показанного далее метода можно сериализовать состояние нашего приложения в единый поток:

```
private static void SaveApplicationState(Stream stream) {
    // Конструирование модуля форматирования для сериализации
    BinaryFormatter formatter = new BinaryFormatter();
```

```
// Сериализация всего состояния приложения
formatter.Serialize(stream, s_customers);
formatter.Serialize(stream, s_pendingOrders);
formatter.Serialize(stream, s_processedOrders);
}
```

Для восстановления состояния приложения воспользуемся вот таким методом:

```
private static void RestoreApplicationState(Stream stream) {
    // Конструирование модуля форматирования сериализации
    BinaryFormatter formatter = new BinaryFormatter();

    // Десериализация состояния приложения (выполняется в том же
    // порядке, что и сериализация)
    s_customers = (List<Customer>) formatter.Deserialize(stream);
    s_pendingOrders = (List<Order>) formatter.Deserialize(stream);
    s_processedOrders = (List<Order>) formatter.Deserialize(stream);
}
```

Третий тонкий момент, на который хотелось бы указать, связан со сборками. При сериализации объекта в поток записываются полное имя типа и определяющая его сборка. По умолчанию `BinaryFormatter` выдает полный идентификатор сборки, в который входит ее полное имя (без расширения), номер версии, язык, региональные параметры и открытый ключ. Десериализуя объект, модуль форматирования берет идентификатор сборки и обеспечивает ее загрузку в выполняющийся домен при помощи метода `Load` класса `System.Reflection.Assembly` (о нем шла речь в главе 23).

После загрузки сборки модуль форматирования ищет в нем тип, совпадающий с типом десериализованного объекта. Если сборка не содержит такого типа, вбрасывается исключение и дальнейшая десериализация объектов отменяется. При обнаружении же нужного типа создается его экземпляр, и поля этого экземпляра инициализируются значениями, содержащимися в потоке. В случаях, когда поля типа не полностью совпадают с именами полей, считанными из потока, вбрасывается исключение `SerializationException` и дальнейшая десериализация объектов приостанавливается. Механизмы, позволяющие переопределить такое поведение, мы обсудим чуть позже.

ВНИМАНИЕ

Некоторые расширяемые приложения используют для загрузки сборки метод `Assembly.LoadFrom`, а затем конструируют объекты из найденных в загруженной сборке типов. Такие объекты могут без проблем сериализоваться в поток. Однако при обратной процедуре модуль форматирования пытается загрузить сборку, вызывая метод `Load` класса `Assembly` вместо метода `LoadFrom`. В большинстве случаев CLR не сможет найти файл сборки и вбрасывает исключение `SerializationException`. Такое поведение зачастую становится сюрпризом для

разработчиков. Ведь после корректной сериализации они ожидают такой же корректной десериализации.

Если ваше приложение сериализует объекты, типы которых определены в сборке, загружаемой методом `Assembly.LoadFrom`, рекомендую вам реализовать метод с сигнатурой, совпадающей с сигнатурой делегата `System.ResolveEventHandler`, а перед вызовом метода `Deserialize` зарегистрировать этот метод с событием `AssemblyResolve` класса `System.AppDomain`. После того как метод `Deserialize` вернет управление, отмените регистрацию. Теперь, если модуль форматирования не сможет загрузить сборку, CLR вызовет ваш метод `ResolveEventHandler`. В него будет передан идентификатор не загруженной сборки. Метод извлечет оттуда имя файла и использует его для построения маршрута доступа к нужной сборке. После этого будет вызван метод `Assembly.LoadFrom`, чтобы загрузить сборку и вернуть итоговую ссылку на нее из метода `ResolveEventHandler`.

Итак, мы рассмотрели основы сериализации и десериализации графа объектов. Пришло время узнать, каким образом определять собственные сериализуемые типы, а также рассмотреть механизмы, позволяющие получить дополнительный контроль над сериализацией и десериализацией.

Сериализуемые типы

Конструируя тип, разработчик должен принять решение, допускать или нет сериализацию экземпляров типа. По умолчанию эта процедура не допускается. К примеру, следующий код не работает ожидаемым образом:

```
internal struct Point { public Int32 x, y; }
private static void OptInSerialization() {
    Point pt = new Point { x = 1, y = 2 };
    using (var stream = new MemoryStream()) {
        new BinaryFormatter().Serialize(stream, pt); // исключение
                                                    // SerializationException
    }
}
```

Если запустить такой код, метод `Serialize` вбросит исключение `System.Runtime.Serialization.SerializationException`. Дело в том, что разработчик типа `Point` не указал в явном виде, что объекты данного типа можно сериализовать. Решить проблему можно при помощи настраиваемого атрибута `System.SerializableAttribute`, примененного к типу следующим образом (обратите внимание, что данный атрибут принадлежит пространству имен `System`, а не `System.Runtime.Serialization`):

```
[Serializable]
internal struct Point { public Int32 x, y; }
```


Теперь, если снова скомпоновать и запустить приложение, все начнет работать ожидаемым образом, а объекты типа `Point` будут сериализоваться в поток. Сериализуя граф, модуль форматирования проверяет способность к сериализации каждого из объектов. Если хотя бы один из объектов не допускает данную процедуру, метод `Serialize` вбрасывает исключение `SerializationException`.

ПРИМЕЧАНИЕ

При сериализации графа объектов некоторые типы могут оказаться сериализуемыми, а некоторые — нет. По причинам, связанным с производительностью, модуль форматирования перед сериализацией не проверяет возможность этой операции для всех объектов. А значит, может возникнуть ситуация, когда некоторые объекты окажутся сериализованными в поток до появления исключения `SerializationException`. В результате в потоке ввода-вывода оказываются поврежденные данные. Желательно, чтобы код приложения умел корректно восстанавливаться после такой ситуации. Этого можно добиться, например, сериализуя объекты сначала в `MemoryStream`. Если процедура для всех объектов пройдет успешно, байты из `MemoryStream` можно скопировать в любой другой поток ввода-вывода (то есть в файл или в сеть).

Настраиваемый атрибут `SerializableAttribute` может применяться только к ссылочным типам (классам), значимым типам (структурам), перечислимым типам (перечислениям) и делегатам (имейте в виду, что перечислимые типы и делегаты всегда сериализуемы, поэтому к ним не нужно явно применять атрибут `SerializableAttribute`). Данный атрибут не наследуется производными типами. Соответственно, в следующем примере объект `Person` может быть сериализован, а вот объект `Employee` — нет:

```
[Serializable]
internal class Person { ... }

internal class Employee : Person { ... }
```

Проблему можно решить, применив к типу `Employee` атрибут `SerializableAttribute`:

```
[Serializable]
internal class Person { ... }

[Serializable]
internal class Employee : Person { ... }
```

Видите, как легко решается проблема. А вот обратная ситуация, когда требуется определить тип, производный от базового и не имеющий атрибута `SerializableAttribute`, уже не столь тривиальна. Тут все зависит от конструирования; если базовый тип не допускает сериализации своих экземпляров, его поля нельзя будет подвергнуть этой процедуре, ведь базовый объект факти-

чески является частью производного. Именно поэтому классу `System.Object` назначен атрибут `SerializableAttribute`.

ПРИМЕЧАНИЕ

В общем случае большинство типов лучше делать сериализуемыми. Хотя бы потому, что это дает пользователям большую гибкость. Однако следует учитывать, что при сериализации читаются все поля объекта вне зависимости от того, с каким модификатором они были объявлены — `public`, `protected`, `internal` или `private`. Поэтому если тип содержит конфиденциальные или защищенные данные (например, пароли), вряд ли стоит делать его сериализуемым. Если же вы работаете с типом, не предназначенным для сериализации, и не имеете доступа к его исходному коду, не все потеряно. Чуть позже я расскажу, как сделать такой тип сериализуемым.

Управление сериализацией и десериализацией

После назначения типу настраиваемого атрибута `SerializableAttribute` все экземпляры его полей (открытые, закрытые, защищенные и т. п.) становятся сериализуемыми¹. Впрочем, в типе можно указать некоторые экземпляры как не подлежащие сериализации. В общем случае это делается по двум причинам:

- ❑ Поле содержит информацию, становящуюся недействительной после десериализации. Например, сюда относятся объекты, содержащие обработчик объектов ядра Windows (таких как файлы, процессы, потоки, мьютексы, события, семафоры и т. п.). Их десериализация в другой процесс или на другую машину бессмысленна, так как обработчики привязаны к значениям конкретного процесса.
- ❑ Поля содержат легко обновляемую информацию. В этом случае сделав их не подлежащими сериализации, вы уменьшите объем передаваемых данных, а значит, повысите производительность.

В показанном далее фрагменте кода с помощью настраиваемого атрибута `System.NonSerializedAttribute` помечаются поля, не подлежащие сериализации (имейте в виду, что этот атрибут определен в пространстве имен `System`, а не `System.Runtime.Serialization`):

```
[Serializable]
internal class Circle {
```

продолжение ➤

¹ В C# внутри типов, помеченных атрибутом `[Serializable]`, не стоит определять автоматически реализуемые свойства. Дело в том, что имена полей, генерируемые компилятором, могут меняться после каждой следующей компиляции, что сделает невозможной десериализацию экземпляров типа.

```

private Double m_radius;

[NonSerialized]
private Double m_area;

public Circle(Double radius) {
    m_radius = radius;
    m_area = Math.PI * m_radius * m_radius;
}
...
}

```

Объекты класса `Circle` допускают сериализацию. Но модуль форматирования сериализует только значения поля `m_radius`. Значения поля `m_area` не сериализуются, так как этому полю был назначен атрибут `NonSerializedAttribute`. Его назначают только полям типа, но действие атрибута распространяется на поля и при наследовании другим типом. В пределах одного типа его можно назначить целому набору полей.

Предположим, в нашем коде объект `Circle` конструируется следующим образом:

```
Circle c = new Circle(10);
```

Полю `m_area` было присвоено значение, равное примерно 314,159. Но при сериализации объекта в поток ввода-вывода записывается только значение поля `m_radius`, равное 10. Хотя таково было наше пожелание, при десериализации потока обратно в объект `Circle` возникнет проблема. Поле `m_radius` получит значение 10, а вот поле `m_area` станет равным 0, а не 314,159!

Показанный далее фрагмент кода демонстрирует, как решить эту проблему:

```

[Serializable]
internal class Circle {
    private Double m_radius;

    [NonSerialized]
    private Double m_area;

    public Circle(Double radius) {
        m_radius = radius;
        m_area = Math.PI * m_radius * m_radius;
    }

    [OnDeserialized]
    private void OnDeserialized(StreamingContext context) {
        m_area = Math.PI * m_radius * m_radius;
    }
}

```

Теперь объект `Circle` снабжен методом с настраиваемым атрибутом `System.Runtime.Serialization.OnDeserializedAttribute`¹. При десериализации экземпляра типа модуль форматирования проверяет наличие в типе метода с данным атрибутом. При дальнейшем вызове метода все сериализуемые поля получают корректные значения и к тому же станут доступными для дополнительных операций, без которых невозможна полная десериализация объекта.

В модифицированной версии объекта `Circle` я заставил метод `OnDeserialized` вычислить площадь круга на основе значения поля `m_radius`, помещая результат в поле `m_area`. В результате это поле получает нужное нам значение 314,159.

Кроме настраиваемого атрибута `OnDeserializedAttribute` в пространстве имен `System.Runtime.Serialization` определены также настраиваемые атрибуты `OnSerializingAttribute`, `OnSerializedAttribute` и `OnDeserializingAttribute`, применив которые к методам нашего типа мы получаем дополнительный контроль над сериализацией и десериализацией.

Вот пример класса, применяющего к методу все эти атрибуты:

```
[Serializable]
public class MyType {
    Int32 x, y; [NonSerialized] Int32 sum;

    public MyType(Int32 x, Int32 y) {
        this.x = x; this.y = y; sum = x + y;
    }

    [OnDeserializing]
    private void OnDeserializing(StreamingContext context) {
        // Пример. Присвоение полям значений по умолчанию в новой версии типа
    }

    [OnDeserialized]
    private void OnDeserialized(StreamingContext context) {
        // Пример. Инициализация временного состояния полей
        sum = x + y;
    }

    [OnSerializing]
    private void OnSerializing(StreamingContext context) {
        // Пример. Модификация состояния перед сериализацией
    }
}
```

продолжение ➤

¹ Применение настраиваемого атрибута `System.Runtime.Serialization.OnDeserialized` является более предпочтительным в случае вызова метода при десериализации объекта, чем реализация типом метода `OnDeserialization` интерфейса `System.Runtime.Serialization.IDeserializationCallback`.

```
[OnSerialized]
private void OnSerialized(StreamingContext context) {
    // Пример. Восстановление любого состояния после сериализации
}
}
```

Метод, определенный с этими четырьмя атрибутами, должен принимать единственный параметр `StreamingContext` (подобно мы поговорим о нем позже) и возвращать значение типа `void`. Имя метода может быть произвольным. Объявлять его следует с модификатором `private`, чтобы исключить вызов из обычного кода; модули форматирования имеют достаточный уровень прав, чтобы вызывать закрытые методы.

ПРИМЕЧАНИЕ

При сериализации набора объектов модуль форматирования сначала вызывает все методы объектов, помеченные атрибутом `OnSerializing`. Затем сериализуются поля всех объектов, последней наступает очередь методов объектов с атрибутом `OnSerialized`. Аналогично при десериализации набора объектов модуль форматирования вызывает сначала методы объектов, помеченные атрибутом `OnDeserializing`, затем десериализует поля объектов, в завершение вызывая методы объектов с атрибутом `OnDeserialized`.

Следует учитывать, что при десериализации, обнаружив тип с помеченным атрибутом `OnDeserialized` методом, модуль форматирования добавляет ссылку на этот объект во внутренний список. Этот список просматривается в обратном порядке после десериализации всех объектов, и модуль форматирования вызывает метод с атрибутом `OnDeserialized` для каждого объекта. При этом происходит корректное присвоение значения всем сериализуемым полям и предоставляется доступ ко всем необходимым для полной десериализации объекта операциям. Обратный порядок вызова методов обеспечивает сначала десериализацию внутренних объектов, а уж затем десериализуются включающие их в себя внешние объекты.

К примеру, представьте коллекцию (`Hashtable` или `Dictionary`), для управления набором элементов использующую хэш-таблицу. Тип объектов этой коллекции реализует метод с атрибутом `OnDeserialized`. Несмотря на то что коллекция десериализуется первой (перед ее элементами), указанный метод вызывается в самом конце (после аналогичных методов для всех элементов). Это позволяет элементам завершить десериализацию, корректно инициализировав все поля, и правильно вычислить хэш-код. Затем коллекция создает внутренний сегмент памяти и с помощью хэш-кода элементов помещает их в этот сегмент. Подробный пример с классом `Dictionary` вы найдете в этой главе далее.

Если сериализовать экземпляр типа, добавить к нему новое поле и попытаться десериализовать не содержащий этого поля объект, модуль формати-

рования вбросит исключение `SerializationException`, попутно сообщив о том, что данные в десериализуемом потоке содержат неверное количество членов. Подобная проблема часто возникает в новых версиях сценариев, так как при переходе от старой версии к типу добавляются новые поля. К счастью, можно воспользоваться атрибутом `System.Runtime.Serialization.OptionalFieldAttribute`.

Атрибут `OptionalFieldAttribute` назначается каждому новому полю, добавляемому к типу. Встретив поле с таким атрибутом, модуль форматирования не вбрасывает исключение `SerializationException`, даже если данные в потоке не содержат такого поля.

Сериализация экземпляров типа

В этом разделе подробно рассматривается тема сериализации полей объекта. Эта тема поможет вам понять нетривиальные приемы сериализации и десериализации, которым посвящен остаток данной главы.

Для облегчения работы модуля форматирования в FCL включен тип `FormatterServices` из пространства имен `System.Runtime.Serialization`. Он обладает только статическими методами и не допускает создания экземпляров. Вот каким образом модуль форматирования автоматически сериализует объект, типу которого назначен атрибут `SerializableAttribute`:

1. Модуль форматирования вызывает метод `GetSerializableMembers` класса `FormatterServices`:

```
public static MemberInfo[] GetSerializableMembers(
    Type type, StreamingContext context);
```

Для получения открытых и закрытых экземплярных полей (исключая поля с атрибутом `NonSerializedAttribute`) этот метод использует отражения. Он возвращает массив объектов `MemberInfo` — по одному объекту на каждое сериализуемое экземплярное поле.

2. Полученный массив объектов `System.Reflection.MemberInfo` передается статическому методу `GetObjectData` класса `FormatterServices`:

```
public static Object[] GetObjectData(Object obj, MemberInfo[] members);
```

Этот метод возвращает массив `Object`, в котором каждый элемент определяет значение поля сериализованного объекта. Массивы `Object` и `MemberInfo` параллельны. То есть нулевой элемент массива `Object` представляет собой значение члена, фигурирующего в массиве `MemberInfo` под нулевым индексом.

3. Модуль форматирования записывает в поток идентификатор сборки и полное имя типа.
4. Модуль форматирования пересчитывает элементы двух массивов, записывая в поток ввода-вывода имя каждого члена и его значение.

А вот как выглядит процедура автоматической десериализации объекта, тип которого помечен атрибутом `SerializableAttribute`:

1. Модуль форматирования читает из потока ввода-вывода идентификатор сборки и полное имя типа. Если сборка еще не загружена в домен, он загружает ее (как описано ранее). При невозможности загрузки вбрасывается исключение `SerializationException`, и десериализация объекта останавливается. Если же сборка успешно загружена, модуль форматирования передает статическому методу `GetTypeFromAssembly` класса `FormatterServices` ее идентификатор и полное имя типа:

```
public static Type GetTypeFromAssembly(Assembly assem, String name);
```

Метод возвращает объект `System.Type`, содержащий информацию о типе десериализованного объекта.

2. Модуль форматирования вызывает статический метод `GetUninitializedObject` класса `FormatterServices`:

```
public static Object GetUninitializedObject(Type type);
```

Этот метод выделяет память под новый объект, но не вызывает его конструктор. При этом все байты объекта инициализируются значением `null` или `0`.

3. Тем же способом, что и раньше, модуль форматирования создает и инициализирует массив `MemberInfo`, вызывая метод `GetSerializableMembers` класса `FormatterServices`. Данный метод возвращает набор полей, которые были сериализованы и теперь нуждаются в десериализации.
4. Из содержащихся в потоке ввода-вывода данных модуль форматирования создает и инициализирует массив `Object`.
5. Ссылки на только что размещенный в памяти объект, массив `MemberInfo`, и параллельный ему массив `Object` со значениями полей передаются статическому методу `PopulateObjectMembers` класса `FormatterServices`:

```
public static Object PopulateObjectMembers(
    Object obj, MemberInfo[] members, Object[] data);
```

Этот метод по очереди просматривает элементы массивов, инициализируя каждое поле соответствующим значением. В результате объект оказывается полностью десериализованным.

Управление сериализованными и десериализованными данными

Как уже упоминалось в этой главе, управлять процессами сериализации и десериализации лучше всего при помощи атрибутов `OnSerializing`, `OnSerialized`, `OnDeserializing`, `OnDeserialized`, `NonSerialized` и `OptionalField`. Однако иногда встречаются сценарии, для которых данных атрибутов недостаточно. Кроме того, работа модулей форматирования основана на отражении, а это — не быстрый процесс, значительно замедляющий сериализацию и десериализацию объектов. Чтобы получить полный контроль над данными процедурами и исключить отражение, тип может реализовать интерфейс `System.Runtime.Serialization.ISerializable`, определяемый следующим образом:

```
public interface ISerializable {  
    void GetObjectData(SerializationInfo info, StreamingContext context);  
}
```

Внутри этого интерфейса имеется всего один метод — `GetObjectData`. Но большинство реализующих его типов реализуют также специальный конструктор, который кратко описан далее.

ВНИМАНИЕ

Основной проблемой интерфейса `ISerializable` является тот факт, что его должны реализовывать и все производные типы, гарантировано вызывая метод `GetObjectData` базового класса и специальный конструктор. Кроме того, реализацию данного интерфейса типом нельзя отменить, потому что это приведет к потере совместимости с производными типами. Впрочем, для изолированных типов реализация интерфейса `ISerializable` всегда происходит без проблем. Кроме того, избежать потенциальных неприятностей, связанных с данным интерфейсом, можно при помощи описанных ранее настраиваемых атрибутов.

ВНИМАНИЕ

Интерфейс `ISerializable` и специальный конструктор предназначены для модуля форматирования. Вызов метода `GetObjectData` другим кодом потенциально может привести к возвращению конфиденциальной информации. Кроме того, другой код может при этом создать объект, передающий поврежденные данные. Поэтому методу `GetObjectData` и специальному конструктору рекомендуется назначить следующий атрибут:

```
[SecurityPermissionAttribute(SecurityAction.Demand,  
    SerializationFormatter = true)]
```

При сериализации графа модуль форматирования просматривает каждый объект. Если тип объекта реализует интерфейс `ISerializable`, модуль форма-

тирования игнорирует все пользовательские атрибуты и конструирует новый объект `System.Runtime.Serialization.SerializationInfo`, содержащий реальный набор всех подлежащих сериализации значений объекта.

В конструируемый объект `SerializationInfo` модуль форматирования передает два параметра: `Type` и `System.Runtime.Serialization.IFormatterConverter`. Первый параметр идентифицирует сериализуемый объект. Для уникальной идентификации типа требуется два фрагмента данных: строковое имя типа и идентификатор его сборки (включающий имя сборки, ее версию, региональные стандарты и открытый ключ). Готовый объект `SerializationInfo` получает полное имя типа (запросив свойство `FullName`), сохраняя его в закрытом поле. Для получения полного имени типа используйте свойство `FullTypeName` класса `SerializationInfo`. Аналогично конструктор получает определяющую тип сборку (запрашивая сначала свойство `Module` класса `Type`, затем свойство `Assembly` класса `Module` и, наконец, свойство `FullName` класса `Assembly`), сохраняя полученную строку в закрытом поле. Для получения идентификатора сборки используйте поле `AssemblyName` класса `SerializationInfo`.

ПРИМЕЧАНИЕ

Задать свойства `FullTypeName` и `AssemblyName` класса `SerializationInfo` не всегда возможно. Для изменения типа после сериализации рекомендуется вызвать метод `SetType` класса `SerializationInfo` и передать ему ссылку на желаемый объект `Type`. Это гарантирует корректность задания полного имени и определяющей сборки. Пример применения данного метода вы найдете в этой главе далее.

После создания и инициализации объекта `SerializationInfo` модуль форматирования передает ссылку на него в метод `GetObjectData` типа. Именно метод `GetObjectData` определяет, какая информация необходима для сериализации объекта, и добавляет эту информацию к объекту `SerializationInfo`. Определение необходимой для сериализации информации происходит при помощи одной из множества перегруженных версий метода `AddValue` типа `SerializationInfo`. Для каждого фрагмента данных, который вы хотите добавить, вызывается один метод `AddValue`.

Показанный далее код демонстрирует, каким образом тип `Dictionary<TKey, TValue>` реализует интерфейсы `ISerializable` и `IDeserializationCallback`, добиваясь контроля над сериализацией и десериализацией своих объектов:

```
[Serializable]
public class Dictionary<TKey, TValue>: ISerializable,
    IDeserializationCallback {
    // Здесь закрытые поля (не показанные)

    private SerializationInfo m_siInfo; // Только для десериализации
```

```

// Специальный конструктор (требуемый интерфейсом ISerializable)
// для управления десериализацией
[SecurityPermissionAttribute(
    SecurityAction.Demand, SerializationFormatter = true)]
protected Dictionary<SerializationInfo info, StreamingContext context> {
    // Во время десериализации сохраним
    // SerializationInfo для OnDeserialization
    m_siInfo = info;
}

// Метод управления сериализацией
[SecurityCritical]
public virtual void GetObjectData(
    SerializationInfo info, StreamingContext context) {

    info.AddValue("Version", m_version);
    info.AddValue("Comparer", m_comparer, typeof(IEqualityComparer<TKey>));
    info.AddValue("HashSize", (m_buckets == null) ? 0 : m_buckets.Length);
    if (m_buckets != null) {
        KeyValuePair<TKey, TValue>[] array =
            new KeyValuePair<TKey, TValue>[Count];
        CopyTo(array, 0);
        info.AddValue(
            "KeyValuePairs", array, typeof(KeyValuePair<TKey, TValue>[]));
    }
}

// Метод, вызываемый после десериализации всех ключей/значений объектов
public virtual void IDeserializationCallback.OnDeserialization(
    Object sender) {
    if (m_siInfo == null) return; // Никогда не присваивается.
                                // возвращение управления

    Int32 num = m_siInfo.GetInt32("Version");
    Int32 num2 = m_siInfo.GetInt32("HashSize");
    m_comparer = (IEqualityComparer<TKey>)
        m_siInfo.GetValue("Comparer", typeof(IEqualityComparer<TKey>));
    if (num2 != 0) {
        m_buckets = new Int32[num2];
        for (Int32 i = 0; i < m_buckets.Length; i++) m_buckets[i] = -1;
        m_entries = new Entry<TKey, TValue>[num2];
        m_freeList = -1;
        KeyValuePair<TKey, TValue>[] pairArray = (
            KeyValuePair<TKey, TValue>[]) m_siInfo.GetValue(

```

```

        "KeyValuePairs". typeof(KeyValuePair<TKey, TValue>[]));
    if (pairArray == null)
        ThrowHelper.ThrowSerializationException(
            ExceptionResource.Serialization_MissingKeys);

    for (Int32 j = 0; j < pairArray.Length; j++) {
        if (pairArray[j].Key == null)
            ThrowHelper.ThrowSerializationException(
                ExceptionResource.Serialization_NullKey);

        Insert(pairArray[j].Key, pairArray[j].Value, true);
    }
} else { m_buckets = null; }
m_version = num;
m_siInfo = null;
}

```

Каждый метод `AddValue` принимает в качестве параметра имя типа `String` и набор данных. Обычно эти данные принадлежат к простым значимым типам, таким как `Boolean`, `Char`, `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Single`, `Double`, `Decimal` и `DateTime`. Впрочем, методу `AddValue` можно передать ссылку на тип `Object`, к примеру, `String`. После того как метод `GetObjectData` добавит всю необходимую для сериализации информацию, управление возвращается модулю форматирования.

ПРИМЕЧАНИЕ

Добавлять к типу информацию сериализации нужно только с помощью одного из перегруженных методов `AddValue`. Для полей, тип которых реализует интерфейс `ISerializable`, нельзя вызывать метод `GetObjectData`. Чтобы добавить поле, используйте метод `AddValue`. Метод `GetObjectData` модуль форматирования вызовет уже самостоятельно. Если же вы решите вызвать этот метод для описанного ранее поля, при десериализации потока ввода-вывода модуль форматирования создаст новый объект.

На этом этапе модуль форматирования берет все добавленные к объекту `SerializationInfo` значения и сериализует их в поток ввода-вывода. Обратите внимание, что методу `GetObjectData` передается еще один параметр: ссылка на объект `System.Runtime.Serialization.StreamingContext`. Этот параметр игнорируется методом `GetObjectData` большинства типов, поэтому здесь мы на нем останавливаться не станем, а рассмотрим его отдельно ближе к концу главы.

Итак, вы уже знаете, как задать всю необходимую для сериализации информацию. Пришла пора рассмотреть процедуру десериализации. Извлекая объект из потока ввода-вывода, модуль форматирования выделяет для него место в памяти (вызывая статический метод `GetUninitializedObject` типа `System.Runtime.Serialization.FormatterServices`). Изначально всем полям объекта присваивается значение `0` или `null`. Затем модуль форматирования проверяет, реализует ли

тип интерфейс `ISerializable`. В случае положительного результата проверки модуль пытается вызвать специальный конструктор, параметры которого идентичны параметрам метода `GetObjectData`.

Для классов, помеченных модификатором `sealed`, этот конструктор рекомендуется объявлять закрытым. Это предохранит код от случайного вызова с превышением полномочий. Также конструктор можно пометить модификатором `protected`, предоставив доступ к нему только производным классам. Впрочем, модули форматирования могут вызывать его вне зависимости от того, каким именно способом был объявлен конструктор.

Конструктор получает ссылку на объект `SerializationInfo`, содержащий все значения, добавленные на этапе сериализации. Он может вызывать любые методы `GetBoolean`, `GetChar`, `GetByte`, `GetSByte`, `GetInt16`, `GetUInt16`, `GetInt32`, `GetUInt32`, `GetInt64`, `GetUInt64`, `GetSingle`, `GetDouble`, `GetDecimal`, `GetDateTime`, `GetString` и `GetValue`, передавая им строку с именем, которое использовалось при сериализации значения. Значения, возвращаемые этими методами, затем инициализируют поля нового объекта.

При десериализации полей объекта нужно вызвать метод `Get` с тем же самым типом значения, которое было передано методу `AddValue` в процессе сериализации. Другими словами, если метод `GetObjectData` передал в метод `AddValue` значение типа `Int32`, метод `GetInt32` должен вызываться для значения этого же типа. Если значение в потоке ввода-вывода отличается от того, которое вы пытаетесь получить, модуль форматирования попытается воспользоваться объектом `IFormatterConverter` для «приведения» к нужному типу значения в потоке ввода-вывода.

Как уже упоминалось, при конструировании объекта `SerializationInfo` ему передается объект типа, реализующего интерфейс `IFormatterConverter`. Так как за конструирование отвечает модуль форматирования, он выбирает нужный тип `IFormatterConverter`. Модули `BinaryFormatter` и `SoapFormatter` разработки Microsoft всегда конструируют экземпляр типа `System.Runtime.Serialization.FormatterConverter`. Изменить этот тип вручную вы не можете.

Тип `FormatterConverter` использует статические методы класса `System.Convert` для преобразования значений между базовыми типами, например из `Int32` в `Int64`. Но при необходимости преобразования между произвольными типами `FormatterConverter` вызывает метод `ChangeType` класса `Convert` для приведения сериализованного (или исходного) типа к интерфейсу `IConvertible`. После этого вызывается подходящий интерфейсный метод. Следовательно, если объекты сериализуемого типа требуется десериализовать как другой тип, нужно, чтобы выбранный тип реализовывал интерфейс `IConvertible`. Обратите внимание, что объект `FormatterConverter` используется только при десериализации объектов и при вызове метода `Get`, тип которого не совпадает с типом значения в потоке ввода-вывода.

Вместо одного из многочисленных методов `Get` специальный конструктор может вызвать метод `GetEnumerator`, возвращающий объект `System.Runtime.`

`Serialization.SerializationInfoEnumerator`, при помощи которого можно по очереди просмотреть все значения внутри объекта `SerializationInfo`. Каждое из перечисленных значений представляет собой объект `System.Runtime.Serialization.SerializationEntry`.

Разумеется, никто не запрещает вам создавать собственные типы, производные от типа, реализующего метод `GetObjectData` класса `ISerializable`, а также специальный конструктор. Если ваш тип реализует также интерфейс `ISerializable`, для корректной сериализации и десериализации объекта ваши реализации метода `GetObjectData` и специального конструктора должны вызывать аналогичные функции в базовом классе. В следующем разделе мы поговорим о том, как правильно определить тип `ISerializable`, базовый тип которого не реализует данный интерфейс.

Если ваш производный тип не имеет дополнительных полей и, следовательно, не нуждается в особых сериализации и десериализации, реализовывать интерфейс `ISerializable` не нужно. Подобно другим членам интерфейса метод `GetObjectData` является виртуальным и вызывается для корректной сериализации объекта. Кроме того, модуль форматирования считает специальный конструктор «виртуализированным». То есть во время десериализации модуль форматирования проверяет тип, экземпляр которого он пытается создать. При отсутствии у этого типа специального конструктора модуль форматирования начинает сканировать базовые классы, пока не найдет класс, в котором реализован нужный ему конструктор.

ВНИМАНИЕ

Код специального конструктора обычно извлекает поля из переданного ему объекта `SerializationInfo`. Однако извлечение полей не гарантирует полной десериализации объекта, поэтому код специального конструктора не должен модифицировать объекты, которые он извлекает.

Если вашему типу нужен доступ к членам извлеченного объекта (например, вызов методов), рекомендуем снабдить тип методом с атрибутом `OnDeserialized` или заставить реализовывать метод `OnDeserialization` интерфейса `IDeserializationCallback` (как показано в примере `Dictionary`). Вызов этого метода задает значения полей всех объектов. Но порядок вызова методов `OnDeserialized` и `OnDeserialization` для набора объектов заранее не известен. Так что, несмотря на инициализацию полей, если объект, на который вы ссылаетесь, снабжен методом `OnDeserialized` или реализует интерфейс `IDeserializationCallback`, нельзя быть уверенным в его полной десериализации.

Определение типа, реализующего интерфейс `ISerializable`, не реализуемый базовым классом

Как уже упоминалось, интерфейс `ISerializable` является крайне мощным инструментом, позволяющим типу полностью управлять сериализацией и де-

сериализацией своих экземпляров. Однако за эту мощь приходится платить ответственностью типа за сериализацию еще и всех полей базового типа. Если базовый тип реализует также интерфейс `ISerializable`, нет никаких проблем. Достаточно вызвать метод `GetObjectData`.

Однако иногда требуется определить тип, управляющий сериализацией, при условии, что его базовый тип не реализует интерфейс `ISerializable`. В этом случае производный класс должен вручную сериализовать поля базового типа, добавив их значения в коллекцию `SerializationInfo`. Затем в специальном конструкторе нужно будет извлечь эти значения и задать поля базового класса. В случае когда поля базового класса помечены модификатором `public` или `protected`, это несложно, а вот для закрытых полей задача может стать даже вообще неразрешимой.

Следующий код — это пример корректной реализации метода `GetObjectData` интерфейса `ISerializable` и его конструктора, обеспечивающий сериализацию полей базового типа:

```
[Serializable]
internal class Base {
    protected String m_name = "Jeff";
    public Base() { /* Наделяем тип способностью создавать экземпляры */ }
}

[Serializable]
internal class Derived : Base, ISerializable {
    private DateTime m_date = DateTime.Now;
    public Derived() { /* Наделяем тип способностью создавать экземпляры */ }

    // Если конструктор не существует, исключение SerializationException
    // Если класс не изолирован, конструктор должен быть защищен
    [SecurityPermissionAttribute(
        SecurityAction.Demand, SerializationFormatter = true)]
    private Derived(SerializationInfo info, StreamingContext context) {
        // Получение набора сериализуемых членов для нашего и базовых классов
        Type baseType = this.GetType().BaseType;
        MemberInfo[] mi = FormatterServices.GetSerializableMembers(
            baseType, context);

        // Десериализация полей базового класса из объекта данных
        for (Int32 i = 0; i < mi.Length; i++) {
            // Получение поля и присвоение ему десериализованного значения
            FieldInfo fi = (FieldInfo)mi[i];
            fi.SetValue(this, info.GetValue(
                baseType.FullName + "+" + fi.Name, fi.FieldType));
        }
    }
}
```

```

// Десериализация значений, сериализованных для этого класса
m_date = info.GetDateTime("Date");
}

[SecurityPermissionAttribute(
    SecurityAction.Demand, SerializationFormatter = true)]
public virtual void GetObjectData(
    SerializationInfo info, StreamingContext context) {
    // Сериализация нужных значений для этого класса
    info.AddValue("Date", m_date);

    // Получение набора сериализуемых членов для нашего и базовых классов
    Type baseType = this.GetType().BaseType;
    MemberInfo[] mi = FormatterServices.GetSerializableMembers(
        baseType, context);

    // Сериализация полей базового класса в объект данных
    for (Int32 i = 0; i < mi.Length; i++) {
        // Полное имя базового типа ставим в префикс имени поля
        info.AddValue(baseType.FullName + "+" + mi[i].Name,
            ((FieldInfo)mi[i]).GetValue(this));
    }
}

public override String ToString() {
    return String.Format("Name={0}, Date={1}", m_name, m_date);
}
}

```

В этом коде присутствует базовый класс `Base`, помеченный только настраиваемым атрибутом `SerializableAttribute`. Производным от него является класс `Derived`, также помеченный этим атрибутом и реализующий интерфейс `ISerializable`. Ситуацию усугубляет тот факт, что оба класса определяют поле типа `String` с именем `m_name`. При вызове метода `AddValue` класса `SerializationInfo` нельзя добавлять значения с одним и тем же именем. Это ограничение обходится путем присвоения каждому полю нового имени с префиксом из имени класса. К примеру, когда метод `GetObjectData` вызывает метод `AddValue` для сериализации поля `m_name` класса `Base`, имя значения записывается как `"Base+m_name"`.

Контексты потока ввода-вывода

Как уже упоминалось, сериализовать объект можно куда угодно: в тот же самый процесс, в другой процесс на этой же машине, в другой процесс на другой машине и т. п. Бывают ситуации, когда объект нужно заранее уведомить, куда он будет десериализован, так как это влияет на его состояние. Например, объект, являющийся оболочкой для `Windows-семафора`, может инициировать

сериализацию обработчика ядра при условии, что десериализация произойдет в тот же самый процесс. Ведь обработчики ядра действительны только в пределах одного процесса. При этом если десериализация будет произведена в другой процесс на этой же машине, объект сможет сериализовать строковое имя семафора. Если же вдруг окажется, что десериализация ожидается на другую машину, появится исключение, так как семафоры действительны только в пределах одной машины.

Ряд упомянутых в данной главе методов в качестве параметра принимают структуру `StreamingContext`. Это простая структура значимого типа, имеющая всего два открытых, предназначенных только для чтения свойства (табл. 24.1).

Таблица 24.1. Свойства перечисления `StreamingContext`

Имя члена	Тип члена	Описание
State	Streaming-Context-States	Набор битовых флагов, указывающих источник или приемник сериализуемых/десериализуемых данных
Context	Object	Ссылка на объект, содержащий нужный пользователю контекст

Получивший структуру `StreamingContext` метод может исследовать битовые флаги свойства `State` и определить источник или приемник сериализуемых/десериализуемых объектов. Возможные значения флагов перечислены в табл. 24.2.

Таблица 24.2. Флаги перечисления `StreamingContextStates`

Имя флага	Значение флага	Описание
CrossProcess	0x0001	Источником или приемником является другой процесс на той же машине
CrossMachines	0x0002	Источник или приемник находится на другой машине
File	0x0004	Источником или приемником является файл. При этом не стоит предполагать, что десериализовать данные будет тот же самый процесс
Persistence	0x0008	Исходный или целевой контекст является хранилищем баз данных или файлов. При этом не стоит предполагать, что десериализовать данные будет тот же самый процесс
Remoting	0x0010	Данные являются удаленными по отношению к контексту в неизвестном расположении. Это может быть как та же самая, так и другая машина

продолжение ➤

Таблица 24.2 (продолжение)

Имя флага	Значение флага	Описание
Other	0x0020	Источник или приемник неизвестны
Clone	0x0040	Точное копирование графа объекта. Код сериализации может предполагать десериализацию данных тем же процессом, а значит, безопасным для доступа обработчикам и другим неуправляемым ресурсам
CrossAppDomain	0x0080	Источником или приемником является другой домен приложений
All	0x00FF	Возможна передача или получение сериализованных данных из любых упомянутых контекстов. Используется по умолчанию

Теперь, когда вы знаете, как получить информацию, поговорим о том, как ее задать. В интерфейсе `IFormatter` (реализуемом как типом `BinaryFormatter`, так и типом `SoapFormatter`) определено доступное для чтения и записи свойство типа `StreamingContext` с именем `Context`. В процессе конструирования модуль форматирования инициализирует свойство `Context`, присваивая `StreamingContextStates` значение `All`, а ссылке на дополнительный объект состояния — значение `null`.

При наличии модуля форматирования можно создать структуру `StreamingContext`, используя любые битовые флаги `StreamingContextStates`, а также при желании передав ссылку на объект, содержащий дополнительную контекстную информацию. После чего остается присвоить свойству `Context` новый объект `StreamingContext` до вызова метода `Serialize` или `Deserialize`. Показанный ранее метод `DeepClone` демонстрирует, как объяснить модулю форматирования, что вы сериализуете/десериализуете граф объекта исключительно с целью его копирования.

Сериализация в другой тип и десериализация в другой объект

Богатая инфраструктура сериализации в `.NET Framework` позволяет разработчикам создавать типы, допускающие сериализацию и десериализацию в другой тип или объект. Вот зачем это может быть нужно:

- ❑ Некоторые типы, например `System.DBNull` и `System.Reflection.Missing`, допускают существование в домене приложений только одного экземпляра. Их часто называют одноэлементными. Сериализуя и десериализуя, например, объект `DBNull`, нужно избегать создания в домене нового объекта.

Возвращаемая после десериализации ссылка должна указывать на уже существующий в домене объект `DBNull`.

- ❑ Некоторые типы (например, `System.Type`, `System.Reflection.Assembly` и другие связанные с отражениями типы, такие как `MemberInfo`) допускают существование всего одного экземпляра на тип, сборку, член и т. п. Представьте массив ссылок на объекты `MemberInfo`. Допустима ситуация, когда пять ссылок указывают на один объект. Это положение дел должно сохраняться и после сериализации и десериализации. Более того, элементы должны ссылаться на объект `MemberInfo`, существующий в домене для определенного члена. Такой подход полезен также для последовательного опроса объектов соединения базы данных и любых других типов объектов.
- ❑ Для объектов, контролируемых удаленно, CLR сериализует информацию о серверном объекте таким образом, что при десериализации на клиенте CLR создает объект, являющийся *представителем* (проху) сервера на стороне клиента. Тип представителя отличается от типа серверного объекта, но для клиентского кода эта ситуация прозрачна. Если клиент вызывает для объекта-представителя экземплярные методы, код представителя переправляет вызов на сервер, который в действительности и обрабатывает запрос.

Вот код, демонстрирующий корректные сериализацию и десериализацию одноэлементного типа:

```
// Допустим один экземпляр типа на домен
[Serializable]
public sealed class Singleton : ISerializable {
    // Единственный экземпляр этого типа
    private static readonly Singleton theOneObject = new Singleton();

    // Поля экземпляра
    public String Name = "Jeff";
    public DateTime Date = DateTime.Now;

    // Закрытый конструктор для создания однокомпонентного типа
    private Singleton() { }

    // Метод, возвращающий ссылку на однокомпонентный тип
    public static Singleton GetSingleton() { return theOneObject; }

    // Метод, вызываемый при сериализации объекта Singleton
    // Рекомендую явно реализованный здесь интерфейсный метод
    [SecurityPermissionAttribute(
        SecurityAction.Demand, SerializationFormatter = true)]
    void ISerializable.GetObjectData(
```

продолжение ➤

```

        SerializationInfo info, StreamingContext context) {
    info.SetType(typeof(SingletonSerializationHelper));
    // Добавлять значений не нужно
}

[Serializable]
private sealed class SingletonSerializationHelper : IObjectReference {
    // Метод, вызываемый после десериализации этого объекта (без полей)
    public Object GetRealObject(StreamingContext context) {
        return Singleton.GetSingleton();
    }
}

// ПРИМЕЧАНИЕ. Специальный конструктор так и НЕ вызывается
}

```

Класс `Singleton` представляет тип, для которого в домене приложений может существовать только один экземпляр. Показанный далее код тестирует процедуры сериализации и десериализации этого типа, проверяя выполнение данного условия:

```

private static void SingletonSerializationTest() {
    // Создание массива с несколькими ссылками на один объект Singleton
    Singleton[] a1 = { Singleton.GetSingleton(), Singleton.GetSingleton() };
    Console.WriteLine("Do both elements refer to the same object? "
        + (a1[0] == a1[1])); // "True"

    using (var stream = new MemoryStream()) {
        BinaryFormatter formatter = new BinaryFormatter();

        // Сериализация и десериализация элементов массива
        formatter.Serialize(stream, a1);
        stream.Position = 0;
        Singleton[] a2 = (Singleton[])formatter.Deserialize(stream);

        // Проверяем, что все работает, как нужно:
        Console.WriteLine("Do both elements refer to the same object? "
            + (a2[0] == a2[1])); // "True"
        Console.WriteLine("Do all elements refer to the same object? "
            + (a1[0] == a2[0])); // "True"
    }
}

```

Попытаемся понять, что же происходит. Для загруженного в домен типа `Singleton` CLR вызывает статический конструктор, создающий объект `Singleton` и сохраняющий ссылку на него в статическом поле `s_theOneObject`. Класс `Singleton` не имеет открытых конструкторов, что не дает стороннему коду создавать его экземпляры.

В методе `SingletonSerializationTest` создается массив из двух элементов, каждый из которых ссылается на объект `Singleton`. Для инициализации этих элементов вызывается статический метод `GetSingleton` класса `Singleton`. Он возвращает ссылку на один объект `Singleton`. Первый вызов метода `WriteLine` выводит "True", указывая, что оба элемента массива ссылаются на один объект.

Далее метод `SingletonSerializationTest` вызывает метод `Serialize` модуля форматирования для сериализации массива и его элементов. Обнаружив, что тип `Singleton` реализует интерфейс `ISerializable`, модуль форматирования вызывает метод `GetObjectData`, который передает в метод `SetType` тип `SingletonSerializationHelper`. В результате модуль форматирования сериализует объект `Singleton` в качестве объекта `SingletonSerializationHelper`. Так как метод `AddValue` в данном случае не вызывается, в поток ввода-вывода не записывается никакой дополнительной информации. Кроме того, модуль форматирования, обнаружив, что оба элемента массива ссылаются на один и тот же объект, сериализует только один из них.

После сериализации массива метод `SingletonSerializationTest` вызывает метод `Deserialize` модуля форматирования. При работе с потоком ввода-вывода модуль форматирования пытается десериализовать объект `SingletonSerializationHelper`, который с его точки зрения был им ранее сериализован (на самом деле, именно поэтому класс `Singleton` не имеет специального конструктора, обычно требующего реализации интерфейса `ISerializable`). Сконструировав объект `SingletonSerializationHelper`, модуль форматирования обнаруживает, что данный тип реализует интерфейс `System.Runtime.Serialization.IObjectReference`. В FCL этот интерфейс определяется следующим образом:

```
public interface IObjectReference {  
    Object GetRealObject(StreamingContext context);  
}
```

Для реализующих такой интерфейс типов модуль форматирования вызывает метод `GetRealObject`, возвращающий ссылку на объект, на который вы действительно хотите сослаться после завершения десериализации. В моем примере для типа `SingletonSerializationHelper` метод `GetRealObject` возвращает ссылку на уже существующий в домене объект `Singleton`. После возвращения управления методом `Deserialize` массив `a2` содержит два элемента, каждый из которых ссылается на объект `Singleton` домена. Вспомогательный объект `SingletonSerializationHelper`, использовавшийся при десериализации, в данный момент недоступен и будет уничтожен при следующей сборке мусора.

Второй вызов метода `WriteLine` выводит значение "True", подтверждая, что оба элемента массива `a2` указывают на один и тот же объект. Аналогичный результат дает третий и последний вызов этого метода, так как на один и тот же объект и в самом деле указывают элементы обоих массивов.

Суррогаты сериализации

До этого момента мы обсуждали способы изменения реализации типов, позволяющие управлять сериализацией и десериализацией их экземпляров. Однако существует возможность переопределить поведение этих процессов при помощи кода, не относящегося к реализации типа. Вот зачем это нужно:

- ❑ разработчик может сериализовать типы, изначально для этой процедуры не предназначенные;
- ❑ разработчик может наложить одну версию типа на другую.

Чтобы заставить работать данный механизм, нужно определить «суррогатный тип», который возьмет на себя работу по сериализации и десериализации существующего типа. Затем следует зарегистрировать экземпляр суррогатного типа, объяснив модулю форматирования, за действия какого существующего типа он будет отвечать. В результате при попытке сериализовать или десериализовать экземпляр существующего типа модуль форматирования будет вызывать методы, определенные суррогатным объектом. Сконструируем пример, демонстрирующий, как все это работает.

Тип суррогата сериализации должен реализовывать интерфейс `System.Runtime.Serialization.ISerializationSurrogate`, определяемый в FCL следующий образом:

```
public interface ISerializationSurrogate {  
    void GetObjectData(Object obj, SerializationInfo info,  
        StreamingContext context);  
    Object SetObjectData(Object obj, SerializationInfo info,  
        StreamingContext context, ISurrogateSelector selector);  
}
```

Теперь посмотрим на пример использования этого интерфейса. Предположим, программа содержит объекты `DateTime`, значения которых привязаны к компьютеру пользователя. Каким образом сериализовать эти объекты в поток ввода-вывода, указав их значения, как всемирное время? Ведь только при таком подходе вы можете перенаправить поток на машину, расположенную на другом конце планеты, сохранив корректные значения даты и времени. Так как вы не можете менять тип `DateTime`, представленный в FCL, остается определить собственный суррогатный класс, управляющий сериализацией и десериализацией объектов `DateTime`. Вот как он выглядит:

```
internal sealed class UniversalToLocalTimeSerializationSurrogate :  
    ISerializationSurrogate {  
    public void GetObjectData(  
        Object obj, SerializationInfo info, StreamingContext context) {  
        // Переход от локального к мировому времени  
        info.AddValue("Date", ((DateTime)obj).ToUniversalTime().ToString("u"));  
    }  
}
```

```
public Object SetObjectData(Object obj, SerializationInfo info,
    StreamingContext context, ISurrogateSelector selector) {
    // Переход от мирового времени к локальному
    return DateTime.ParseExact(
        info.GetString("Date"), "u", null).ToLocalTime();
}
```

Здесь метод `GetObjectData` работает почти как одноименный метод интерфейса `ISerializable`. Отличается он всего одним параметром: ссылкой на «реальный» объект, который требуется сериализовать. В показанном варианте метода `GetObjectData` данный объект приводится к типу `DateTime`, его значение преобразуется из локального в универсальное время, а полученная в итоге строка (отформатированная с использованием универсального эталона полной даты/времени) добавляется в коллекцию `SerializationInfo`.

Для десериализации объекта `DateTime` вызывается метод `SetObjectData`. Ему передается ссылка на объект `SerializationInfo`. Метод извлекает из коллекции строковые данные, анализирует их как строку в формате универсальной полной даты/времени и преобразует полученный объект `DateTime` в формат локального машинного времени.

Первый параметр метода `SetObjectData`, объект `Object`, выглядит немножко странно. Непосредственно перед вызовом метода модуль форматирования выделяет место (через статический метод `GetUninitializedObject` класса `FormatterServices`) под экземпляр типа, для которого предназначается суррогат. Все поля этого экземпляра имеют значение `0` или `null`, и для объекта не вызывается никаких конструкторов. Метод `SetObjectData` может просто инициализировать его поля, используя значения из переданного методу объекта `SerializationInfo`, а затем вернуть значение `null`. В качестве альтернативы метод `SetObjectData` может создать совсем другой объект или даже другой объектный тип и вернуть ссылку на него. В этом случае модуль форматирования проигнорирует любые изменения, которые могли произойти с объектом, переданным им в метод `SetObjectData`.

В моем примере класс `UniversalToLocalTimeSerializationSurrogate` действует как суррогат для значимого типа `DateTime`. И поэтому параметр `obj` ссылается на упакованный экземпляр типа `DateTime`. Менять поля в большинстве значимых типов нельзя (так как они предполагаются неизменными), поэтому мой метод `SetObjectData` игнорирует параметр `obj` и возвращает новый объект `DateTime` с нужным значением.

Можно спросить, откуда при сериализации/десериализации объекта `DateTime` модуль форматирования узнает, как использовать тип `ISerializationSurrogate`. Следующий код тестирует класс `UniversalToLocalTimeSerializationSurrogate`:

```
private static void SerializationSurrogateDemo() {
    using (var stream = new MemoryStream()) {
```

```

// 1. Создание желаемого модуля форматирования
IFormatter formatter = new SoapFormatter();

// 2. Создание объекта SurrogateSelector
SurrogateSelector ss = new SurrogateSelector();

// 3. Селектор выбирает наш суррогат для объекта DateTime
ss.AddSurrogate(typeof(DateTime), formatter.Context,
    new UniversalToLocalTimeSerializationSurrogate());

// ПРИМЕЧАНИЕ. AddSurrogate можно вызывать более одного раза
// и зарегистрировать несколько суррогатов

// 4. Модуль форматирования использует наш селектор
formatter.SurrogateSelector = ss;

// Создание объекта DateTime с локальным временем машины
// и его сериализация
DateTime localTimeBeforeSerialize = DateTime.Now;
formatter.Serialize(stream, localTimeBeforeSerialize);

// Поток выводит универсальное время в виде строки,
// проверяя, что все работает
stream.Position = 0;
Console.WriteLine(new StreamReader(stream).ReadToEnd());

// Десериализация универсального времени и преобразование
// объекта DateTime в локальное время
stream.Position = 0;
DateTime localTimeAfterDeserialize =
    (DateTime)formatter.Deserialize(stream);

// Проверка корректности работы
Console.WriteLine(
    "LocalTimeBeforeSerialize={0}", localTimeBeforeSerialize);
Console.WriteLine(
    "LocalTimeAfterDeserialize={0}", localTimeAfterDeserialize);
}
}

```

После выполнения четвертого шага модуль форматирования готов к работе с суррогатными типами. При вызове метода `Serialize` тип каждого объекта ищется в наборе, управляемом объектом `SurrogateSelector`. При обнаружении соответствия вызывается метод `GetObjectData` класса `ISerializationSurrogate`, чтобы получить информацию для записи в поток ввода-вывода.

При вызове метода `Deserialize` тип подлежащего десериализации объекта также ищется в объекте `SurrogateSelector`, и при обнаружении соответствия

вызывается метод `SetObjectData` класса `ISerializationSurrogate`, задающий поля десериализуемого объекта.

Объект `SurrogateSelector` управляет закрытой хэш-таблицей. При вызове метода `AddSurrogate` методы `Type` и `StreamingContext` создают ключ, а объект `ISerializationSurrogate` является его значением. Если ключ для рассматриваемых методов `Type` и `StreamingContext` уже существует, методом `AddSurrogate` вбрасывается исключение `ArgumentException`. Включив в ключ объект `StreamingContext`, вы регистрируете два объекта суррогатного типа: один умеет сериализовать в файл объект `DateTime`, второй — сериализовать/десериализовать объект `DateTime` в другой процесс.

ПРИМЕЧАНИЕ

В классе `BinaryFormatter` существует ошибка, из-за которой суррогат не может сериализовать ссылающиеся друг на друга объекты. Для ее устранения следует передать ссылку на объект `ISerializationSurrogate` статическому методу `GetSurrogateForCyclicalReference` класса `FormatterServices`. Этот метод возвращает объект `ISerializationSurrogate`, который затем можно передать методу `AddSurrogate` класса `SurrogateSelector`. Однако при работе с методом `GetSurrogateForCyclicalReference` метод `SetObjectData` вашего суррогата должен менять значение внутри объекта, на который ссылается параметр `obj`, и в конце концов возвращать вызывающему методу значение `null` или `obj`. В прилагаемом к книге коде (он доступен для загрузки) демонстрируется, как отредактировать класс `UniversalToLocalTimeSerializationSurrogate` и метод `SerializationSurrogateDemo`, чтобы обеспечить поддержку циклических ссылок.

Цепочка селекторов суррогатов

Несколько объектов `SurrogateSelector` можно связать в цепочку. К примеру, у вас может быть объект `SurrogateSelector` с набором суррогатов сериализации, предназначенных для сериализации типов в представителе с целью обеспечения удаленных вызовов или вызовов между доменами приложений. Может существовать и специальный объект `SurrogateSelector` с набором суррогатов сериализации, предназначенных для преобразования типов версии 1 в типы версии 2.

Чтобы модуль форматирования смог использовать все эти объекты, их нужно соединить в цепочку. Тип `SurrogateSelector` реализует интерфейс `ISurrogateSelector`, определяющий три метода. Все они связаны с созданием цепочек. Вот определение интерфейса `ISurrogateSelector`:

```
public interface ISurrogateSelector {  
    void ChainSelector(ISurrogateSelector selector);  
    ISurrogateSelector GetNextSelector();  
}
```

продолжение ➤


```
ISerializationSurrogate GetSurrogate(  
    Type type, StreamingContext context, out ISurrogateSelector selector);  
}
```

Метод `ChainSelector` вставляет объект `ISurrogateSelector` непосредственно после того объекта `ISurrogateSelector`, с которым ведется работа (на него указывает ключевое слово `this`). Метод `GetNextSelector` возвращает ссылку на следующий объект `ISurrogateSelector` в цепочке, или значение `null`, если цепочка закончилась.

Метод `GetSurrogate` ищет пару `Type/StreamingContext` в объекте `ISurrogateSelector`, идентифицируемом ключевым словом `this`. Если пара не обнаруживается, рассматривается следующий объект `ISurrogateSelector` и т. д. В случае же обнаружения пары метод `GetSurrogate` возвращает объект `ISerializationSurrogate`, выполняющий сериализацию/десериализацию просматриваемого типа. Кроме того, возвращается содержащий пару объект `ISurrogateSelector`; но необходимости в этом обычно нет, и данный результат работы метода игнорируется. Если ни один из объектов `ISurrogateSelector` в цепочке не имеет совпадения для пары `Type/StreamingContext`, метод `GetSurrogate` возвращает значение `null`.

ПРИМЕЧАНИЕ

В FCL определен интерфейс `ISurrogateSelector` и реализующий его тип `SurrogateSelector`. При этом практически никто и никогда не определяет собственных типов, реализующих этот интерфейс. Ведь единственной причиной его создания может быть разве что повышенная гибкость при наложении одного типа на другой, например, если возникает необходимость сериализовать все типы, наследующие от определенного базового класса определенным образом. Превосходным образчиком такого класса является `System.Runtime.Remoting.Messaging.RemotingSurrogateSelector`. Сериализуя объекты для удаленной передачи, CLR форматирует их методом `RemotingSurrogateSelector`. Этот суррогатный селектор сериализует все производные от `System.MarshalByRefObject` объекты таким образом, что десериализация приводит к созданию объектов-представителей на стороне клиента.

Переопределение сборки и/или типа при десериализации объекта

В процессе сериализации объекта модули форматирования извлекают полное имя типа и полное имя определяющей этот тип сборки. При десериализации эта информация позволяет им точно узнать тип конструируемого и инициализируемого объекта. При обсуждении интерфейса `ISerializationSurrogate` я продемонстрировал механизм, позволяющий производить сериализацию и десериализа-

цию определенного типа. Тип, реализующий интерфейс `ISerializationSurrogate`, привязан к определенному типу в определенной сборке.

Однако бывают ситуации, когда указанный механизм оказывается недостаточно гибким. Вот ситуации, в которых может оказаться полезным десериализация объекта в другой тип:

- ❑ Перемещение реализации типа из одной сборки в другую. Например, номер версии сборки меняется, и новая сборка начинает отличаться от исходной.
- ❑ Объект с сервера сериализуется в поток, отправляемый на сторону клиента. При обработке потока клиент может десериализовать объект в совершенно другой тип, код которого «знает», как удаленным методом обратиться к объектам на сервере.
- ❑ Разработчик создает новую версию типа и именно в нее требуется десериализовать все ранее сериализованные объекты.

Десериализация объектов в другой тип легко выполняется при помощи класса `System.Runtime.Serialization.SerializationBinder`. Достаточно определить тип, производный от абстрактного типа `SerializationBinder`. В показанном далее коде предполагается, что версия 1.0.0.0 сборки определяет класс, вызывающий объект `Ver1`. И эта новая версия определяет класс `Ver1ToVer2SerializationBinder`, а также класс, вызывающий объект `Ver2`:

```
internal sealed class Ver1ToVer2SerializationBinder : SerializationBinder {
    public override Type BindToType(String assemblyName, String typeName) {
        // Десериализация объекта Ver1 из версии 1.0.0.0 в объект Ver2

        // Вычисление имени сборки, определяющей тип Ver1
        AssemblyName assemVer1 = Assembly.GetExecutingAssembly().GetName();
        assemVer1.Version = new Version(1, 0, 0, 0);

        // При десериализации объекта Ver1 версии v1.0.0.0 превращаем его в Ver2
        if (assemblyName == assemVer1.ToString() && typeName == "Ver1")
            return typeof(Ver2);

        // В противном случае возвращаем запрошенный тип
        return Type.GetType(String.Format("{0}. {1}", typeName, assemblyName));
    }
}
```

После создания модуля форматирования нужно создать экземпляра `Ver1ToVer2SerializationBinder` и присвоить открытому для чтения и записи свойству `Binder` ссылку на объект привязки. А потом можно вызывать метод `Deserialize`. В процессе десериализации модуль форматирования обнаружит привязку и для каждого обрабатываемого объекта вызовет метод `BindToType`, передавая ему имя сборки и тип, которые требуется десериализовать. Именно метод `BindToType` определяет, какой тип следует сконструировать.

ПРИМЕЧАНИЕ

Класс `SerializationBinder` позволяет также в процессе сериализации менять информацию о сборке/типе путем переопределения метода `BindToName`. Данный метод выглядит следующим образом:

```
public virtual void BindToName(Type serializedType,  
    out string assemblyName, out string typeName)
```

Во время сериализации модуль форматирования вызывает данный метод, передавая тип, который он собирается сериализовать. После этого вы можете передать (при помощи двух параметров `out`) сборку и тип, которые хотите сериализовать вы. Если же вы передаете в качестве параметров значения `null` и `null` (именно это происходит в заданной по умолчанию реализации), тип и сборка остаются без изменений.

ЧАСТЬ V

Многопоточность

Глава 25.	Потоки исполнения	744
Глава 26.	Асинхронные вычислительные операции	771
Глава 27.	Асинхронные операции ввода-вывода	813
Глава 28.	Простейшие конструкции синхронизации потоков	852
Глава 29.	Гибридные конструкции синхронизации потоков	889

Глава 25. Потоки исполнения

В этой главе вы познакомитесь с *потоками исполнения*, или просто *потоки* (threads)¹, и узнаете, что разработчики думают о самих потоках и об их применении. Мы поговорим о том, почему в Microsoft Windows появились потоки, о тенденциях развития процессоров, о взаимоотношениях потоков общесистемной исполняющей среды (CLR-потоков) и Windows-потоков, о большой ресурсоемкости потоков, об управлении потоками в Windows, о классах .NET Framework, предоставляющих доступ к свойствам потоков, и о многом другом.

В главах пятой части книги объясняется, каким образом в Windows и CLR организована архитектура потоков. Надеюсь, после прочтения этих глав вы получите фундаментальные знания, позволяющие эффективно применять потоки и создавать быстро реагирующие, надежные и расширяемые приложения и компоненты.

Зачем потоки в Windows?

На заре компьютерной эры операционные системы не поддерживали концепцию потоков. Точнее, существовал всего один поток исполнения, обслуживающий как код операционной системы, так и код приложений. В результате задание, выполнение которого требовало времени, не давало быстро перейти к выполнению других заданий. К примеру, во времена 16-разрядной системы Windows обычной была ситуация, когда распечатывающее документ приложение приостанавливало работу всей машины. Операционная система и остальные приложения просто «зависали». А если вдруг в приложении возникала ошибка, которая приводила к бесконечному циклу, она вообще порождала массу проблем.

Пользователю оставалось только перезагрузить компьютер, нажав кнопку Reset или выключатель питания. Разумеется, пользователи ненавидели такие ситуации (и продолжают ненавидеть до сих пор), потому что приходилось закрывать все запущенные приложения; более того, обрабатываемые этими приложениями данные стирались из памяти. В Microsoft понимали, что 16-разрядная платформа Windows — не столь хороша, чтобы удержать компанию на плаву в ходе дальнейшего развития компьютерной индустрии, поэтому было решено создать новую операционную систему, удовлетворяющую нуждам как корпораций, так и отдельных пользователей. Она должна была быть устойчивой,

¹ Не путать с потоками ввода-вывода (streams). — *Примеч. ред.*

надежной, расширяемой, безопасной и избавленной от большинства недостатков своей предшественницы. Ядро этой операционной системы изначально было ориентировано на Microsoft Windows NT. С годами это ядро претерпело множество обновлений и приобрело дополнительные возможности. Последняя версия ядра поставляется с последними версиями операционных систем Windows для клиента и сервера.

При разработке нового ядра операционной системы было решено запускать каждый экземпляр приложения в отдельном *процессе* (process). Процессом называется набор ресурсов, используемый отдельным экземпляром приложения. Каждому процессу выделяется виртуальное адресное пространство, гарантируя тем самым, что код и данные одного процесса будут недоступны для другого. Это делает приложения отказоустойчивыми, поскольку при таком подходе один процесс не может повредить код или данные другого. Код и данные ядра также недоступны для процессов; а значит, код приложений не в состоянии повредить код или данные операционной системы. Это упрощает работу конечных пользователей. Система становится также более безопасной, потому что код произвольного приложения не имеет доступа к именам пользователей, паролям, информации кредитной карты или иным конфиденциальным данным, с которыми работают другие приложения или сама операционная система.

А что с центральным процессором? Что если приложение войдет в бесконечный цикл? Если процессор всего один, приложение будет выполнять этот бесконечный цикл и не сможет уделять внимание другим операциям. Несмотря на очевидные преимущества (неповрежденные данные и более высокая степень безопасности), система, как и ее предшественницы, не сможет реагировать на действия конечного пользователя. Для решения этой проблемы и были придуманы потоки. Именно *поток* стал той концепцией, которая предназначена для виртуализации процессора в Windows. Каждому Windows-процессу выделяется собственный поток исполнения (функционирующий аналогично процессору), и при попадании кода приложения в бесконечный цикл замораживается только связанный с этим кодом процесс, а остальные процессы (исполняющиеся в собственных потоках) продолжают функционировать!

Ресурсоемкость потоков

Потоки — замечательное изобретение, ведь именно благодаря им Windows отвечает нам, даже несмотря на то, что отдельные приложения могут быть заняты исполнением длительных заданий. Кроме того, с помощью одного приложения (например, диспетчера задач) вы можете принудительно прекратить работу другого приложения, если оно перестает отвечать на запросы. Однако как и любые механизмы виртуализации, потоки потребляют дополнительные ресурсы, требуя пространства (памяти) и времени (снижая производительность

среды исполнения). Рассмотрим эти проблемы более детально. Каждый поток состоит из нескольких частей.

- ❑ **Ядро потока (thread kernel).** Для каждого созданного в ней потока операционная система выделяет и инициализирует одну из структур данных. Набор свойств этой структуры (о них мы поговорим чуть позже) описывает поток. Структура содержит также так называемый контекст потока, то есть блок памяти с набором регистров процессора. На машине с процессором x86 контекст потока занимает около 700 байт памяти, в то время как для процессоров x64 и IA64 нужно уже 1240 и 2500 байт соответственно.
- ❑ **Блок окружения потока (Thread Environment Block, ТЕВ).** Это место в памяти, выделенное и инициализированное в пользовательском режиме (адресное пространство, к которому имеет быстрый доступ код приложений). Этот блок занимает одну страницу памяти (4 Кбайт для процессоров x86 и x64, 8 Кбайт для IA64). Он содержит заголовок цепочки обработки исключений. В этот заголовок вставляет узел каждый блок try, в который входит поток. Если поток существует внутри блока try, узел из цепочки удаляется. Также ТЕВ содержит локальное хранилище данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL.
- ❑ **Стек пользовательского режима (user-mode stack).** Применяется для передаваемых в методы локальных переменных и аргументов. Также он содержит адрес, показывающий, откуда начнет исполнение поток после того, как текущий метод возвратит управление. По умолчанию на каждый стек пользовательского режима Windows выделяет 1 Мбайт памяти¹.
- ❑ **Стек режима ядра (kernel-mode stack).** Используется, когда код приложения передает аргументы в функцию операционной системы, находящуюся в режиме ядра. Для безопасности Windows копирует все аргументы, передаваемые в ядро кодом в пользовательском режиме, из стека потока пользовательского режима в стек режима ядра. После копирования ядро проверяет значения аргументов. Так как код приложения не имеет доступа к стеку режима ядра, приложение не в состоянии изменить уже проверенные аргументы, и с ними начинает работать код ядра операционной системы. Кроме того, ядро вызывает собственные методы и использует стек режима ядра для передачи локальных аргументов, а также для сохранения локальных пере-

¹ Для собственных приложений Windows резервирует 1 Мбайт адресного пространства и иногда, если этого требует поток из-за увеличения стека, осуществляет физическое сохранение. Однако при создании потока управляемым приложением CLR заставляет Windows зарезервировать и немедленно зафиксировать стек, поэтому для каждого созданного потока выделяется физическое хранилище размером 1 Мбайт. Это сделано для повышения надежности управляемого кода в ситуациях, когда системе не хватает памяти. В результате CLR и управляемые приложения не имеют проблем с восстановлением, столкнувшись с недостатком памяти при попытке увеличить стек потока. Для Microsoft SQL Server при выполнении процедур, реализованных в виде управляемого кода, это крайне важно.

менных функции и обратного адреса. В 32-разрядной версии Windows стек режима ядра занимает 12 Кбайт, а в 64-разрядной — 24 Кбайт.

- **Уведомления о создании и завершении потоков.** Политика Windows такова, что если в процессе создается поток, то для всех загруженных в этот процесс DLL-библиотек вызывается метод `DllMain` и в него передается флаг `DLL_THREAD_ATTACH`. Соответственно, при завершении потока этому методу передается уже флаг `DLL_THREAD_DETACH`. Получая уведомления об этих событиях, некоторые DLL-библиотеки выполняют специальные операции инициализации или очистки для каждого созданного/завершенного в процессе потока. К примеру, DLL-библиотека C-Runtime выделяет место под хранилище локальных состояний потока, которое используется при наличии в потоке функций из указанной библиотеки.

На заре развития Windows в процесс загружалось 5 или 6 библиотек, в то время как в наши дни некоторые процессы включают в себя несколько сотен библиотек. Скажем, в адресное пространство приложения Microsoft Office Outlook на моем компьютере загружено около 250 DLL-библиотек! Это означает, что созданный в данном приложении новый поток получит возможность приступить к своей работе только после вызова 250 функций из DLL. По завершении потока все эти функции будут вызваны снова. Это не может не влиять на производительность создания и завершения потоков в процессе¹.

Теперь вы видите, каких затрат времени и памяти стоит создание потока, его поддержание в системе и завершение. Но на самом деле ситуация еще хуже из-за необходимости *переключения контекста* (context switching). Компьютер с одним процессором может одновременно выполнять только что-то одно. Следовательно, операционная система должна распределять физический процессор между всеми своими потоками (логическими процессорами).

В произвольный момент времени Windows передает процессору на исполнение один поток. Этот поток исполняется в течение некоторого временного интервала, иногда называемого *тактом* (quantum). После завершения этого интервала контекст Windows переключается на другой поток. При этом обязательно происходит следующее:

1. Значения регистров процессора для исполняющегося в данный момент потока сохраняются в структуре контекста, которая располагается в ядре потока.
2. Из набора имеющихся потоков выделяется тот, которому будет передано управление. Если выбранный поток принадлежит другому процессу, Windows переключает для процессора виртуальное адресное пространство.

¹ Библиотеки для C# и большинства других управляемых языков программирования не имеют метода `DllMain`, поэтому управляемые DLL-библиотеки не получают уведомлений `DLL_THREAD_ATTACH` и `DLL_THREAD_DETACH`. Что же касается неуправляемых библиотек, то при помощи Win32-функции `DisableThreadLibraryCalls` они могут отключать режим получения уведомлений. К сожалению, многие разработчики неуправляемого кода не используют эту функцию просто потому, что не знают о ней.

Только после этого возможно выполнение какого-либо кода или доступ к каким-либо данным.

3. Значения из выбранной структуры контекста потока загружаются в регистры процессора.

После переключения контекста процессор исполняет выбранный поток, пока не истечет выделенное потоку время, после этого снова происходит переключение контекста. Windows делает это примерно каждые 30 мс. Никакого выигрыша в производительности или потреблении памяти переключение контекстов не дает. Оно требуется только для того, чтобы конечные пользователи имели надежную и быстро реагирующую на их действия операционную систему.

Если поток какого-то приложения входит в бесконечный цикл, Windows его периодически выгружает и передает процессору другой поток для исполнения. К примеру, это может быть поток диспетчера задач, позволяющего завершить процесс, в котором исполняется зависший в бесконечном цикле поток. Процесс в результате прекращает свою работу, теряя несохраненные данные, но остальные процессы в системе продолжают функционировать, как ни в чем не бывало. Пользователю не приходится перезагружать компьютер. Как видите, переключение контекстов приводит к повышению отказоустойчивости, хотя за это приходится платить снижением производительности.

Издержки в данном случае выше, чем вы можете предположить. При переключении контекста на другой поток производительность серьезно падает. Пока работа ведется с одним потоком, его код и данные находятся в кэше процессора, а значит, обращения процессора к оперативной памяти, замедляющие работу, случаются не столь часто. Однако новый поток, скорее всего, исполняет совсем другой код и имеет доступ к другим данным, которых еще нет в кэше процессора. Значит, прежде чем вернуться к прежней скорости работы, процессор вынужден некоторое время обращаться к оперативной памяти, наполняя кэш. А примерно через 30 мс происходит очередное переключение контекста.

Время переключения контекста зависит от архитектуры процессора и его быстродействия. А время заполнения кэша зависит от запущенных в системе приложений, размера самого кэша и ряда других факторов. Поэтому оценить, с какими временными затратами связано каждое переключение контекста, невозможно. Достаточно просто запомнить, что при разработке высокопроизводительных приложений и компонентов переключения контекста нужно по возможности избегать.

ВНИМАНИЕ

Если в конце временного промежутка Windows решает продолжить исполнение уже исполняемого потока (а не переходить к другому), переключения контекста не происходит. Это значительно повышает производительность.

ВНИМАНИЕ

Поток может самопроизвольно завершиться до конца такта, что происходит довольно часто, например, если поток ожидает ввода-вывода. Так, поток приложения Notepad обычно ничего не делает, ожидая ввода данных. При нажатии пользователем клавиши Windows пробуждает этот поток, чтобы тот обработал данное действие. Обработка занимает всего 5 мс, после чего вызывается Win32-функция, сообщающая Windows о готовности к обработке следующего события ввода. Если события ввода отсутствуют, поток переводится в состояние ожидания. В результате поток игнорируется процессором до следующего события ввода. Такой подход повышает производительность системы.

В ходе процедуры сборки мусора CLR приостанавливает все потоки, просматривает их стеки в поисках корней, помечает объекты в куче, снова просматривает стеки (обновляя корни объектов, перемещенных в процессе сжатия) и возобновляет исполнение всех потоков. Так что, избавившись от потоков, вы повысите производительность сборки мусора. В процессе отладки Windows приостанавливает все потоки приложения в каждой точке останова и снова запускает их при переходе к следующему шагу или при запуске приложения. Соответственно, чем больше потоков, тем медленнее будет происходить отладка.

Из сказанного можно сделать заключение, что использования потоков нужно по возможности избегать, так как они потребляют память и требуют времени для своего создания, управления и завершения. При переключении контекста и сборке мусора время также тратится впустую. Тем не менее следует понимать, что без потоков тоже не обойтись, так как именно они обеспечивают в Windows приемлемые показатели надежности и времени реакции.

Не стоит забывать и о том, что компьютер с несколькими процессорами может исполнять несколько потоков одновременно, делая больше работы за меньшее время. Каждому ядру процессора назначается свой поток, и это ядро организует собственное переключение контекстов. Операционная система следит за тем, чтобы один поток не распределялся одновременно между несколькими ядрами, так как это привело бы к хаосу. В настоящее время повсеместно встречаются компьютеры с несколькими процессорами, гиперпараллельными процессорами или многоядерными процессорами. Но на заре создания Windows работать приходилось на машинах с одним процессором, и именно поэтому для повышения надежности операционной системы были введены потоки. В настоящее время потоки позволяют повысить производительность на машинах с несколькими ядрами.

Оставшиеся главы этой книги посвящены механизмам Windows и CLR, позволяющим эффективно ответить на вопрос, как при минимальном количестве потоков сохранить работоспособность кода и каким образом масштабировать код для исполнения на машине с многоядерным процессором.

Так дальше не пойдет!

Если во главу угла поставить производительность, оптимальное число потоков на машине должно быть равно числу установленных на ней процессоров. То есть на компьютере с одним процессором допустим всего один поток, на компьютере с двумя процессорами — два и т. д. Причины в данном случае очевидны: если количество потоков превышает количество процессоров, начинается переключение контекста, и производительность падает. А при наличии одного потока на процессор контекстное переключение не требуется, и все потоки исполняются на полной скорости.

Тем не менее при разработке Windows специалисты Microsoft отдали предпочтение надежности и скорости реакции на действия пользователя, а не скорости расчетов и производительности выполнения приложений. И с моей точки зрения это правильно. Не думаю, что кто-либо пользовался бы Windows или .NET Framework, если бы проблемы приложений по-прежнему решались исключительно путем перезагрузки операционной системы. Именно поэтому в Windows каждому процессу выделяется собственный поток, что повышает надежность системы и быстроту реакции. К примеру, на рис. 25.1 показано окно диспетчера задач, открытое на вкладке Performance (Быстродействие).

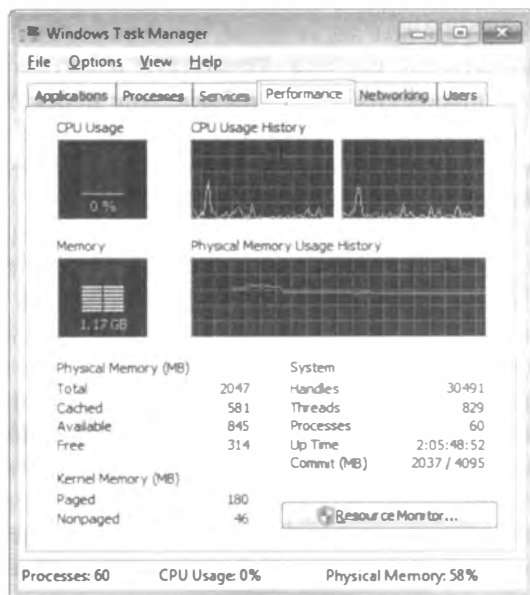


Рис. 25.1. Диспетчер задач, демонстрирующий производительность системы

Как видите, на момент получения снимка экрана на моем компьютере было запущено 60 процессов, а значит, по меньшей мере, 60 потоков. Ведь для каждого процесса существует хотя бы один поток. Однако как легко увидеть, по-

токов на самом деле 829! Это означает, что только на стеки потоков выделено 829 Мбайт памяти, и это при том, что ее полный объем на моем компьютере составляет 2 Гбайт. Что же касается соотношения количества процессов и потоков, то в среднем на один процесс приходится 13,8 потока.

Тем не менее расположенная в левом верхнем углу диаграмма загрузки процессора показывает, что в настоящий момент загрузка нулевая. То есть 100 % времени эти 829 потоков в буквальном смысле ничего не делают — они просто занимают память, которая не используется, пока потоки не начинают исполняться. Резонно спросить, нужны ли приложениям все эти ничего не делающие потоки? Разумеется, нет. Чтобы посмотреть, какой из процессов является самым бесполезным, перейдите на вкладку **Processes** (Процессы), добавьте столбец **Threads** (Счетчик потоков)¹ и отсортируйте его по убыванию, как показано на рис. 25.2.

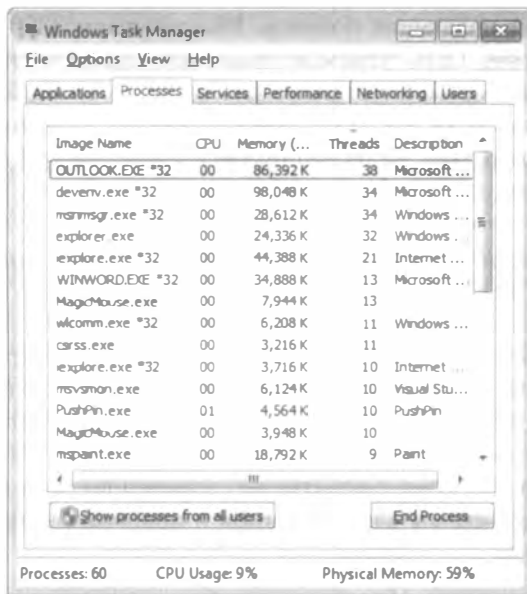


Рис. 25.2. Процессы в окне диспетчера задач

Как видите, приложение Outlook создало 38 потоков, но использует 0 % мощности процессора, приложение Microsoft Visual Studio (Devenv.exe) создало 34 потока, опять же, используя 0 % мощности процессора, та же самая картина с приложением Windows Live Messenger (Msnmsgr.exe), создавшим 34 потока и т. п. Что же происходит?

Знакомясь с Windows, разработчики узнают, что создание процессов в этой операционной системе — дорогостоящая процедура. Создание процесса занимает несколько секунд, требует выделения изрядного объема памяти, эту память

¹ Для этого нужно выбрать в меню View (Вид) команду Select Columns (Выбрать столбцы).

требуется инициализировать, нужно загрузить с диска EXE- и DLL-файлы и т. п. По сравнению с этим создать поток достаточно просто. Поэтому разработчики вместо процессов предпочитают создавать потоки, множество которых мы и видим перед собой. Но в сравнении с большинством других системных ресурсов создание потока — не такая уж дешевая процедура. И применять их следует расчетливо и только там, где они действительно уместны.

Однако, как видите, упомянутые приложения используют потоки не эффективно. Непонятно, зачем они вообще существуют в системе. Одно дело — выделить ресурсы для приложения, и совсем другое — выделить их и не использовать. Ведь выделение памяти под стеки потоков означает, что ее останется меньше для более важных данных, например документов пользователя¹.

А теперь представьте, что процесс запущен в сеансе удаленного рабочего стола одного пользователя, но у машины на самом деле 100 пользователей. То есть запускается 100 экземпляров приложения Outlook, каждый из которых создает 38 ничего не делающих потоков. Мы получаем 3800 потоков, каждый с собственным ядром, ТЕВ, стеком режима пользователя, стеком режима ядра и т. п. Огромное количество впустую потраченных ресурсов. Эту практику пора прекращать, особенно если Microsoft хочет, чтобы пользователи успешно работали с Windows на нетбуках, на большинстве которых всего 1 Гбит оперативной памяти. Именно тому, как наиболее эффективно разработать приложение с минимальным количеством потоков, и посвящены последние главы этой книги.

В настоящее время большинство потоков в системе создается машинным кодом. То есть стек режима пользователя на самом деле всего лишь резервирует адресное пространство, которое обычно полностью не расходуется. Однако по мере того, как все большее количество приложений становятся управляемыми или получают управляемые компоненты (которые поддерживает Outlook), тем более полно стеки расходуют предоставляемое им пространство в 1 Мбайт. Тем не менее все потоки обладают ядром, стеком режима ядра и прочими выделенными им ресурсами. Поэтому с практикой создания потоков пора заканчивать; потоки являются затратным удовольствием и прибегать к ним следует только там, где они на самом деле нужны.

Тенденции развития процессоров

В прошлом имел место постоянный рост быстродействия процессоров, в результате даже медленно работающие приложения при переходе на более

¹ Хотелось бы еще раз продемонстрировать, насколько удручающе обстоят дела. Откройте приложение Notepad.exe и посмотрите с помощью Диспетчера задач количество связанных с ним потоков. Выберите в меню File (Файл) приложения команду Open (Открыть), и как только откроется диалоговое окно File Open (Открытие файла), посмотрите на количество новых потоков. На моей машине результатом открытия этого окна стало создание 22 дополнительных потоков! Аналогично обстоят дела с любым другим приложением, открывающим диалоговое окно открытия или сохранения файла. И большинство этих потоков не завершается даже после закрытия окна!

новую машину начинали работать быстрее. Однако бесконечно наращивать быстродействие невозможно. Кроме того, процессор, работающий с большой скоростью, выделяет тепло, которое нужно рассеивать. Несколько лет назад я приобрел компьютер новой модели от уважаемого производителя. Однако из-за проблем со встроенным программным обеспечением скорость вращения вентилятора оказалась недостаточной; и через некоторое время процессор и материнская плата просто расплавились. Производитель заменил мне компьютер и «улучшил» встроенное программное обеспечение, просто заставив вентилятор вращаться быстрее. Но из-за этого стали намного быстрее садиться батарейки, ведь вентилятор потреблял много энергии.

С подобными проблемами в наши дни приходится сталкиваться всем производителям аппаратного обеспечения. Из-за отсутствия возможности бесконечно наращивать скорость процессоров, они пытаются уменьшить транзисторы, чтобы на одной микросхеме можно было разместить их больше. Уже существуют кремниевые микросхемы, содержащие два и более процессорных ядра. А значит, скорость функционирования программного обеспечения повысится только при условии, что оно умеет работать с несколькими ядрами. Как этого добиться? *Корректно* используя потоки.

В настоящее время существуют три вида многопроцессорных технологий:

- ❑ **Набор процессоров.** Некоторые компьютеры просто оснащают несколькими процессорами. То есть на материнской плате находятся несколько гнезд, в каждом из которых располагается процессор. Это приводит к увеличению размеров материнской платы, а значит, и корпуса. В некоторых случаях приходится ставить также дополнительные источники питания из-за повышенного потребления энергии. Такие компьютеры использовались в течение нескольких десятилетий, но постепенно их популярность сходит на нет из-за большого размера и высокой стоимости.
- ❑ **Гиперпотоковые микросхемы.** Эта технология (от Intel) позволяет одной микросхеме функционировать как две. Микросхема содержит два набора архитектурных состояний, таких как регистры процессора, при этом имеется всего один набор механизмов исполнения. Для Windows это выглядит, как наличие в машине двух процессоров, и операционная система одновременно планирует поведение двух потоков. Однако при этом выполняется только один из них. Как только он прерывается из-за недостатка размера кэша, ошибочного прогнозирования ветви или зависимости по данным, микросхема переключается на другой поток. Все это происходит на аппаратном уровне, и Windows об этом не «знает». С точки зрения операционной системы оба потока выполняются одновременно. При наличии на одной машине нескольких гиперпотоковых процессоров операционная система сначала назначит по одному потоку на каждый процессор, в результате чего они действительно будут выполняться одновременно. Все же остальные потоки будут распределяться по уже занятым процессорам. По утверждениям Intel, такой подход повышает производительность на 10–30 %.

- **Многоядерные микросхемы.** Несколько лет назад появились микросхемы, содержащие более одного процессорного ядра. На момент написания этой книги были доступны микросхемы с двумя, тремя и четырьмя ядрами. Два ядра имеет даже процессор моего ноутбука. Я уверен, что эта технология скоро распространится даже на мобильные телефоны. Компания Intel работала даже над прототипом процессора с 80 ядрами. Представляете, насколько мощным является такой компьютер! Кроме того, в Intel имеются гиперпоточковые микросхемы с несколькими ядрами.

Неравномерный доступ к памяти

Привлекательные, на первый взгляд, многоядерные процессоры становятся причиной новых проблем. Поскольку все ядра одновременно имеют доступ к ресурсам системы, возникает проблема производительности. Например, при одновременном доступе двух ядер к оперативной памяти полоса пропускания ограничивает общую производительность таким образом, что повышение производительности в системе с двумя ядрами составляет всего от 30 до 70 %. Для решения этой проблемы была разработана так называемая архитектура с *неравномерным доступом к памяти* (Non-Uniform Memory Access, NUMA).

Как показано на рис. 25.3, NUMA-компьютер состоит из четырех узлов. Каждый узел содержит четыре процессора, северный мост, южный мост и локальную память (RAM). К некоторым узлам могут быть подсоединены локальные устройства. Доступ к памяти осуществляется из любого узла; но время доступа не одинаково. К примеру, любой процессор первого узла может быстро обратиться к памяти этого узла. Кроме того, он может обратиться к памяти узлов 2 и 4, но это происходит намного медленней. Доступ же к памяти узла 3 для процессоров из первого узла происходит крайне медленно, потому что узлы 1 и 3 напрямую не связаны. Несмотря на распределение 16 процессоров по четырем разным узлам, аппаратное обеспечение гарантирует согласованность их кэшей друг с другом.

Программный интерфейс Win32 предлагает разработчикам неуправляемого кода множество функций, позволяющих их приложениям выделять память отдельно для каждого NUMA-узла и направлять в него поток. Среда CLR в настоящее время не приспособлена для работы с NUMA-системами. В будущем хотелось бы, чтобы в CLR появились, к примеру, кучи со сборкой мусора для каждого NUMA-узла или чтобы приложения научились определять, в каком узле следует выделить память для того или иного объекта. Возможно, CLR станет перемещать объекты из одного узла в другой в зависимости от того, какой узел имеет самый лучший доступ к объекту.

В начале 1990-х было сложно даже вообразить, что однажды появятся компьютеры с 32 процессорами. Соответственно, 32-разрядная платформа Windows разрабатывалась с ориентацией на машины, количество процессоров в которых достигает 32. А появившаяся позже 64-разрядная платформа Windows ориен-

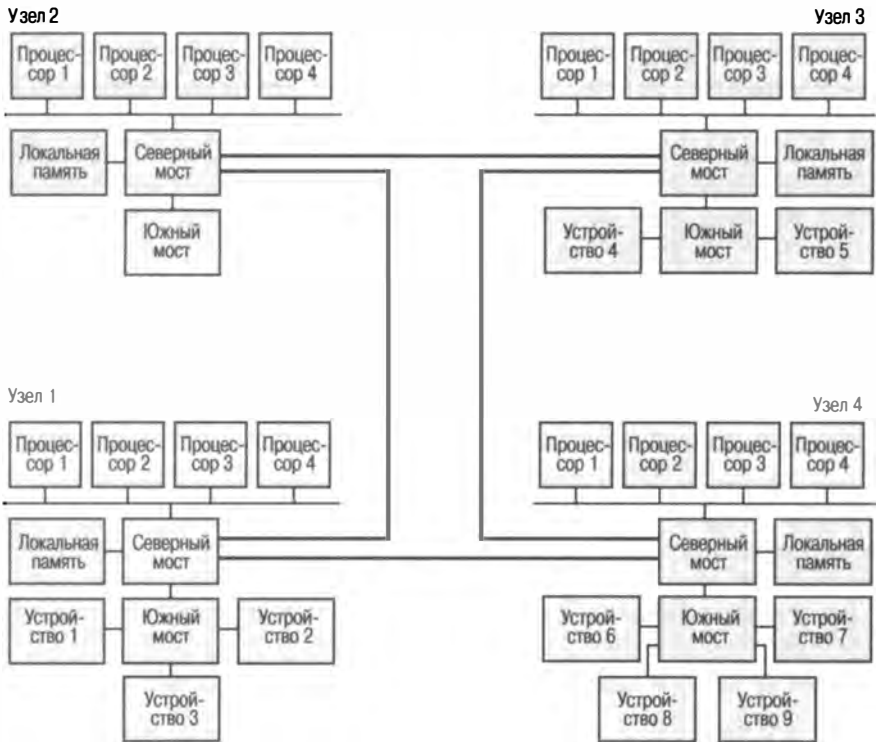


Рис. 25.3. Компьютер с архитектурой NUMA

тирована на машины с 64 процессорами. И хотя сейчас кажется, что 64 процессора — это более чем достаточно, все идет к тому, что в недалеком будущем количество процессоров еще больше увеличится.

Начиная с Windows Server 2008 R2, в Microsoft разрабатывалась операционная система, поддерживающая компьютеры, количество логических процессоров на которых достигало 256. Схему их поддержки иллюстрирует рис. 25.4. Вот как она интерпретируется:

- ❑ На одном компьютере имеется одна или более групп процессоров, каждая из которых содержит от 1 до 64 логических процессоров.
- ❑ Группа процессоров имеет один или несколько NUMA-узлов. Каждый узел содержит несколько логических процессоров, кэш-память и локальную память (все рядом друг с другом).
- ❑ Каждый NUMA-узел имеет одно или несколько гнезд для силиконовых микросхем.
- ❑ Микросхема в каждом гнезде содержит одно или несколько процессорных ядер.
- ❑ Каждое ядро содержит один или несколько логических процессоров (ЛП). Последнее касается гиперпоточковых микросхем.

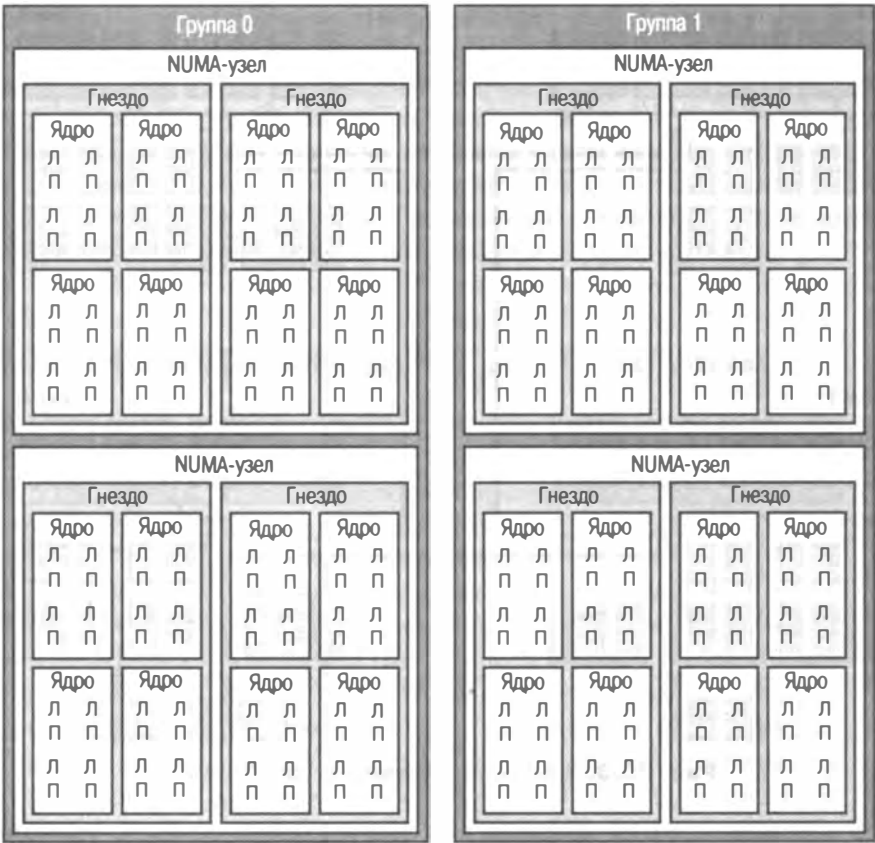


Рис. 25.4. В Windows группы процессоров поддерживают машины, количество логических процессоров в которых достигает 256

В настоящее время CLR не использует преимущества, которые предоставляют группы процессоров, так что все создаваемые этой средой потоки исполняются в нулевой (назначаемой по умолчанию) группе. Поэтому применяться могут максимум 64 ядра при условии работы в 64-разрядной версии Windows. 32-разрядные версии этой операционной системы опять же поддерживают только нулевую группу процессоров, а, значит, запускаемым управляемым приложениям доступно всего 32 процессорных ядра.

CLR- и Windows-потоки

В настоящее время CLR использует способность Windows работать с потоками, поэтому пятая часть данной книги посвящена рассмотрению возможностей, которые открываются перед разработчиками, создающими код с помощью CLR. Мы поговорим о том, как исполняются потоки в Windows и как на их пове-

дение влияет CLR. Для получения дополнительной информации о потоках исполнения рекомендую мои предыдущие книги, в частности пятое издание *Windows via C/C++* (Microsoft Press, 2007).

В настоящее время CLR-потоки аналогичны Windows-потокам, но разработчики CLR не исключают, что со временем они начнут различаться. Вполне может появиться собственная концепция логического потока, не совпадающая с физическими потоками в Windows. К примеру, велись разговоры о создании логических потоков, потребляющих меньше ресурсов, и о том, что множество таких потоков можно запускать поверх небольшого числа физических потоков. Скажем, среда CLR могла бы определять потоки, находящиеся в состоянии ожидания, и назначать им другую задачу. Это позволило бы упростить написание кода, уменьшить количество потребляемых ресурсов и потенциально повысить производительность. К сожалению, достижение этой цели требует изрядных трудозатрат от команды разработчиков CLR, так что вряд ли подобные нововведения можно ожидать в ближайшем будущем.

Для вас сказанное означает, что при управлении потоками следует делать как можно меньше допущений. К примеру, следует избегать механизма *P/Invoke* для вызова машинных функций Windows, так как эти функции ничего не «знают» о CLR-потоках¹. Ограничиваясь по мере возможности типами из *FCL*, вы гарантируете, что написанный код в будущем сможет использовать преимущества новых потоков, как только те появятся.

Потоки для асинхронных вычислительных операций

В этом разделе мы поговорим о том, как создать поток и заставить его исполнить асинхронную вычислительную операцию. При этом я не рекомендую пользоваться приемами, описываемыми в этом разделе. По возможности для этой цели лучше прибегать к доступному в CLR *пулу потоков* (thread pool). О нем мы поговорим в следующей главе.

Возможны ситуации, когда требуется явно создать поток, исполняющий конкретную вычислительную операцию. Обычно такая необходимость возникает при выполнении кода, приводящего поток в состояние, отличное от обычного состояния потока из пула. К примеру:

- ❑ Поток требуется запустить с нестандартным приоритетом (все потоки пула выполняются с обычным приоритетом). Хотя изменить приоритет можно,

¹ Если вам требуется воспользоваться механизмом *P/Invoke* для обращения к машинному коду и важно, чтобы при выполнении кода применялся текущий физический поток операционной системы, воспользуйтесь статическим методом *BeginThreadAffinity* класса *System.Threading.Thread*. После того как надобность в указанном потоке отпадет, CLR можно оповестить при помощи статического метода *EndThreadAffinity* класса *Thread*.

но делать это не рекомендуется, кроме того, изменение приоритета не сохраняется в операциях пула потоков.

- ❑ Чтобы приложение не закрылось до завершения потоком задания, требуется, чтобы поток исполнялся в фоновом режиме. Эта тема подробно рассмотрена в разделе «Фоновые и активные потоки» далее в этой главе. Потоки пула всегда являются фоновыми, и существует риск, что они не успеют выполнить задание из-за того, что CLR решит завершить процесс.
- ❑ Задания, связанные с вычислениями, обычно выполняются крайне долго; для подобных заданий я не стал бы отдавать решение о необходимости создания нового потока на откуп логике пула потоков.
- ❑ Возможно возникнет необходимость преждевременно завершить исполняющийся поток методом `Abort` класса `Thread`, который был подробно рассмотрен в главе 22.

Для создания выделенного потока вам потребуется экземпляр класса `System.Threading.Thread`, чтобы получить который следует передать имя метода в конструктор. Вот прототип такого конструктора:

```
public sealed class Thread : CriticalFinalizerObject, ... {
    public Thread(ParameterizedThreadStart start);
    // Здесь не показаны редко используемые конструкторы
}
```

Параметр `start` задает метод, который будет выполнять выделенный поток. Сигнатура этого метода должна совпадать с сигнатурой делегата `ParameterizedThreadStart`:¹

```
delegate void ParameterizedThreadStart(Object obj);
```

Создание объекта `Thread` является достаточно простой операцией, так как при этом физический поток в операционной системе не появляется. Для создания физического потока, призванного исполнить метод обратного вызова, следует воспользоваться методом `Start` класса `Thread`, передав в него объект (состояние), которое вы хотите сделать аргументом метода обратного вызова. Вот код, демонстрирующий процедуру создания выделенного потока, который затем асинхронно вызывает метод:

```
using System;
using System.Threading;

public static class Program {
    public static void Main() {
```

¹ Для записи класс `Thread` также предлагает конструктор, принимающий делегат `ThreadStart`, который не имеет аргументов и не возвращает значений. Но лично я не рекомендую их использовать из-за многочисленных ограничений. Если метод вашего потока принимает класс `Object` и возвращает значение типа `void`, вызовите его при помощи выделенного потока или пула потока, как показано в главе 26.

```

Console.WriteLine("Main thread: starting a dedicated thread " +
    "to do an asynchronous operation");
Thread dedicatedThread = new Thread(ComputeBoundOp);
dedicatedThread.Start(5);

Console.WriteLine("Main thread: Doing other work here...");
Thread.Sleep(10000);    // Имитирует другую работу (10 секунд)

dedicatedThread.Join(); // Ожидание завершения потока
Console.WriteLine("Hit <Enter> to end this program...");
Console.ReadLine();
}

// Сигнатура метода должна совпадать
// с сигнатурой делегата ParameterizedThreadStart
private static void ComputeBoundOp(Object state) {
    // Метод, выполняемый выделенным потоком
    Console.WriteLine("In ComputeBoundOp: state={0}", state);
    Thread.Sleep(1000);    // Имитирует другую работу (1 секунда)

    // После возвращения методом управления выделенный поток завершается
}
}

```

После компиляции и запуска такого кода получаем:

```

Main thread: starting a dedicated thread to do an asynchronous operation
Main thread: Doing other work here...
In ComputeBoundOp: state=5

```

Так как мы не можем контролировать очередность исполнения потоков в Windows, возможен и другой результат:

```

Main thread: starting a dedicated thread to do an asynchronous operation
In ComputeBoundOp: state=5
Main thread: Doing other work here...

```

Обратите внимание, что метод `Main` вызывает метод `Join`. Последний заставляет вызывающий поток остановить выполнение любого кода до момента, пока поток, определенный при помощи `dedicatedThread`, не завершится сам или не будет завершен.

Мотивы использования потоков

Потоки необходимы по трем причинам:

- ❑ **Изоляция одного кода от другого.** Изоляция повышает надежность ваших приложений. Собственно, именно поэтому в операционной системе Windows

и появилась концепция потоков. Ведь ваши приложения являются для нее сторонними компонентами, и Microsoft не может контролировать их качество. Предварительное тестирование приложений — ваша задача. Предполагается, что вы убедились в их надежности и высоком качестве. В результате отказоустойчивость нужна не столько приложению, сколько операционной системе. Значит, приложению не нужно большого количества потоков. Исключением являются приложения, загружающие компоненты сторонних производителей. Им требуется большая надежность, а значит, и больше потоков.

- **Потоки упрощают написание кода.** Иногда код проще писать, если предположить, что он будет исполняться в собственном потоке. Однако при этом, разумеется, потребуются дополнительные ресурсы, что снизит эффективность кода. Тем не менее мне кажется, что упрощение кодирования стоит потраченных ресурсов. Если бы я придерживался другой точки зрения, то до сих пор писал бы машинный код, а не стал бы разработчиком на C#. Хотя иногда приходится сталкиваться с людьми, которые, думая, что выбрали самую простую методику программирования, на самом деле значительно усложнили себе жизнь (и свой код). Обычно вместе с потоками создается координирующий код, который может потребоваться для получения синхронизирующими конструкциями информации о завершении других потоков. А решение проблемы с координацией требует дополнительных ресурсов и усложняет код. Поэтому перед началом работы с потоками следует хорошо подумать, насколько они действительно вам нужны.
- **Потоки обеспечивают параллельное выполнение.** Повысить производительность можно тогда и только тогда, когда приложение будет выполняться на машине с несколькими процессорами. Впрочем, такие компьютеры в наше время уже достаточно распространены. Вопросы создания приложений, предназначенных для работы в многопроцессорной конфигурации, рассматриваются в главах 26 и 27.

А теперь я хотел бы поделиться с вами своей теорией. Итак, каждый компьютер снабжен таким мощным инструментом, как процессор. И если вы покупаете компьютер, он должен работать все время. Другими словами, я считаю, что все процессоры в машине должны использоваться на 100 %. Впрочем, тут нужно сделать две оговорки. Во-первых, при питании от батареек 100-процентное использование процессора сократит время работы с машиной. Во-вторых, в некоторых центрах обработки данных предпочитают иметь десять компьютеров с процессорами, работающими на половинной мощности, вместо пяти, процессоры которых загружены на 100 %. Дело в том, что полностью загруженный процессор выделяет тепло, а значит, требует системы охлаждения. Однако питание такой системы может оказаться более затратным делом, чем питание большего количества компьютеров, работающих на меньшей мощности. Впрочем, наличие большого количества компьютеров тоже значительно повышает издержки, ведь каждый из них требует периодического обновления аппаратного и программного обеспечения.

Теперь, если вы согласны с моей теорией, нужно определиться с тем, какие задачи должен решать процессор. Но сначала — небольшое вступление. В прошлом как разработчики, так и конечные пользователи считали, что мощность компьютеров недостаточна. И поэтому код не выполнялся, пока конечный пользователь не давал на это разрешения при помощи таких элементов интерфейса, как пункты меню, кнопки и флажки, явно не показывая, что согласен предоставить приложению необходимые ресурсы процессора.

Однако сейчас все изменилось. Современные компьютеры достаточно мощны и в ближайшем будущем могут стать еще более мощными. Как я уже упоминал в этой главе, часто в окне Диспетчера задач можно видеть, что процессор занят 0 % времени. Если бы ядер было не два, а четыре, такая ситуация возникала бы еще чаще. Когда появится 80-ядерный процессор, вообще получится, что практически все время компьютер ничего не делает. С точки зрения потребителя получается, что за большие деньги машина выполняет меньше работы!

Именно поэтому производители аппаратного обеспечения с трудом продают пользователям многоядерные компьютеры. Программное обеспечение не может полноценно воспользоваться предоставляемым преимуществом, а значит, пользователь не получает выгоды от покупки машины с дополнительным процессором. То есть в настоящее время мы имеем избыток компьютерных мощностей, поэтому разработчики могут себе позволить их активное потребление. Раньше даже помыслить было нельзя о том, чтобы приложение занималось дополнительными вычислениями, если не было полной уверенности, что конечному пользователю понадобится результат этих вычислений. Но теперь, при наличии дополнительных мощностей, мы можем думать об этом.

Например, по завершении набора текста в редакторе Visual Studio это приложение автоматически вызывает компилятор и обрабатывает введенный код. Такой подход повышает продуктивность труда разработчиков, так как они сразу видят ошибки вводимого кода и немедленно могут их исправить. Фактически в настоящее время из последовательности редактирование-сборка-отладка пропал центральный член, так как сборка (компиляция) кода осуществляется непрерывно. Конечные пользователи этого даже не замечают благодаря мощному процессору. Ведь частый запуск компилятора никак не отражается на решении других задач. Я думаю, что в будущих версиях Visual Studio из меню исчезнет пункт **Build (Сборка)**, так как компиляция станет полностью автоматической. Не только упрощается пользовательский интерфейс, но и само приложение дает «ответы» на нужды конечного пользователя, повышая продуктивность его работы.

В результате удаления отдельных пунктов меню пользоваться приложением становится проще. Остается меньше вариантов и меньше концепций, которые следует прочитать и запомнить. Именно многоядерная конфигурация позволяет упростить пользование компьютером настолько, что в один прекрасный день с ним сможет работать даже моя бабушка. Для разработчиков удаление элементов пользовательского интерфейса означает меньший объем тестиро-

вания и упрощение основы кода. Кроме того, ослабляется острота проблемы локализации интерфейса и сопроводительной документации. Все это дает возможность экономить время и деньги.

Вот еще несколько примеров активного потребления ресурсов процессора: проверка орфографии и грамматики в документах, пересчет электронных таблиц, индексирование файлов на диске для ускорения процедуры поиска и дефрагментация жесткого диска для повышения производительности ввода-вывода.

Мне нравится мир, в котором пользовательские интерфейсы минимизируются и упрощаются, оставляя больше места для визуализации данных, а приложения сами предлагают информацию, помогающую быстро решать насущные задачи. Пришло время творчески использовать программное обеспечение.

Порядок исполнения и приоритеты потоков

Операционные системы с вытесняющей многозадачностью должны использовать некий алгоритм, определяющий порядок и продолжительность исполнения потоков. В этом разделе рассмотрен алгоритм, применяемый в Windows. Я уже упоминал о наличии контекстной структуры в каждом ядре потока. Эта структура отражает состояние регистров процессора потока во время его исполнения. После каждого такта Windows просматривает все существующие ядра потоков в поисках потоков, которые не находятся в режиме ожидания, выбирает один из них и переключает на него контекст. При этом фиксируется, сколько раз каждый из потоков потребовал переключения контекста. Эту информацию можно увидеть в показанном на рис. 25.5 окне приложения Microsoft Spy++. В нем представлены свойства всех потоков. Обратите внимание, что выбранный поток запускался 31 768 раз¹.

Итак, каждый поток исполняет код и манипулирует данными в адресном пространстве процесса. Через такт Windows переключает контекст. Переключения контекста продолжаются с момента загрузки операционной системы и до завершения ее работы.

Windows называют многопоточной операционной системой с вытесняющей многозадачностью, потому что каждый поток может быть остановлен в произвольный момент времени и вместо него выбран для исполнения другой. Как вы увидите, этим процессом в какой-то степени можно управлять. Просто нужно

¹ Можно также заметить, что поток находится в системе более 25 часов, но использовался менее чем одну секунду процессорного времени. То есть речь идет о непродуктивном расходовании ресурсов.

помнить, что нельзя гарантировать постоянное исполнение потока, не прерываемое никаким другим потоком.

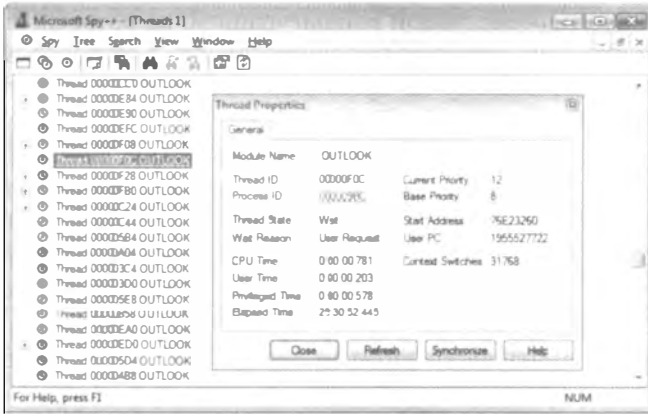


Рис. 25.5. Свойства потоков в приложении Spy++

ПРИМЕЧАНИЕ

Разработчики часто задают вопрос: каким образом можно гарантированно запустить поток через определенное время после какого-то события? Например, как запустить определенный поток через 1 мс после прохождения данных через последовательный порт? Я отвечаю просто: это невозможно.

Такие вещи возможны в операционных системах реального времени, но Windows к ним не относится. Операционные системы реального времени требуют хорошего знакомства с аппаратным обеспечением, на базе которого они работают. То есть вам должны быть известны задержки контроллеров жесткого диска, клавиатуры и других компонентов. Платформа Windows создавалась в Microsoft для работы с самым разным аппаратным обеспечением: различными процессорами, драйверами, сетями и т. п. Именно поэтому она не является операционной системой реального времени. Следует добавить, что из-за CLR управляемый код еще хуже приспособлен для работы в реальном времени. Причин этому много, в том числе динамическая загрузка библиотек, JIT-компиляция кода и сборка мусора, начало выполнения которой невозможно спрогнозировать.

Каждому потоку назначается уровень приоритета с нулевого (самого низкого) до 31 (самого высокого). При выборе потока, который будет передан процессору, сначала рассматриваются потоки с самым высоким приоритетом и ставятся в очередь в цикле. При обнаружении потока с приоритетом 31 он передается процессору. После завершения такта ищется следующий поток с аналогичным приоритетом, чтобы переключить на него контекст.

При наличии в очереди потоков с приоритетом 31 система никогда не передаст процессору поток с меньшим приоритетом. Это условие называется

зависанием (starvation), а возникает оно в случае, когда потоки с высоким приоритетом потребляют практически все время процессора и не дают исполняться потокам более низкого приоритета. Зависание намного реже возникает на машинах с многопроцессорной конфигурацией, на которых потоки с приоритетами 31 и 30 могут исполняться одновременно. Система всегда старается загрузить процессор, поэтому он простаивает только при отсутствии готовых к исполнению потоков.

Потоки с высоким приоритетом всегда исполняются перед потоками с низким приоритетом вне зависимости от того, какие задания выполняют последние. Если в ходе исполнения какого-либо потока появляется поток с более высоким приоритетом, исполнение немедленно приостанавливается (даже если поток находится в середине такта) и процессору передается новый поток.

В момент загрузки система создает *поток обнуления страниц* (zero page thread), которому назначается нулевой приоритет. Это единственный поток в системе с таким приоритетом. Его задача состоит в обнулении свободных страниц и исполняется он только при отсутствии других потоков.

Ясное дело, что с точки зрения разработчика сложно придумать рациональное объяснение назначению потокам приоритетов. Почему в Microsoft одному потоку присвоили приоритет 10, а другому — 23? Для решения этого вопроса Windows предлагает некий абстрактный уровень.

При разработке приложения следует решить, должно ли оно реагировать быстрее или медленнее, чем другие запущенные на этой же машине приложения. В соответствии с этим решением выбирается класс приоритета для процесса. В Windows поддерживаются шесть классов приоритетов: Idle (холостого хода), Below Normal (ниже обычного), Normal (обычный), Above Normal (выше обычного), High (высокий) и Realtime (реального времени). По умолчанию выбирается приоритет Normal, он же является самым распространенным.

Приоритет холостого хода подходит для приложений, которые запускаются в системе, где больше ничего не происходит (это такие приложения, как хранители экрана). Даже не используемый в интерактивном режиме компьютер может быть занят (к примеру, функционируя как файловый сервер) и не должен конкурировать за процессорное время с хранителем экрана. Отслеживающие статистику приложения, периодически обновляющие некоторое состояние, обычно тоже не должны становиться препятствием для более важных заданий.

Высокий приоритет следует использовать только там, где это действительно необходимо. А приоритета реального времени вообще лучше по возможности избегать. Его выбор может помешать выполнению таких системных заданий, как дисковый ввод-вывод или передача данных по сети. Поток с приоритетом реального времени может помешать обработке данных, вводимых с клавиатуры или при помощи мыши, создавая у пользователя впечатление, что система перестала работать. По большому счету для выбора такого приоритета нужно иметь веские основания, например необходимость с минимальной задержкой

отвечать на события аппаратного уровня или выполнять какие-то кратковременные задания.

ПРИМЕЧАНИЕ

Чтобы система работала без сбоев, процесс невозможно запустить с приоритетом реального времени при отсутствии прав на увеличение приоритета выполнения. Эта привилегия по умолчанию имеется только у администраторов и пользователей с расширенными правами.

Выбрав класс приоритета, не нужно думать о том, как ваше приложение соотносится со всеми остальными приложениями, достаточно сосредоточиться на потоках своего приложения. В Windows поддерживаются семь относительных приоритетов потоков: Idle (холостого хода), Lowest (самый низкий), Below Normal (ниже обычного), Normal (обычный), Above Normal (выше обычного), Highest (самый высокий) и Time-Critical (требующий немедленной обработки). Эти приоритеты соотносятся с классами приоритетов процесса. По умолчанию для потоков используется обычный приоритет, соответственно, он применяется чаще всего.

Подводя итог, скажем, что процесс является членом класса приоритета и внутри него потокам назначаются связанные друг с другом приоритеты. Если вы заметили, я ничего не говорил об уровнях приоритета с нулевого по 31. Разработчики приложений никогда не имеют с ними дела напрямую. За них это делает система. Соотношение между классом приоритета процесса, относительным приоритетом потока и итоговым уровнем приоритета иллюстрирует табл. 25.1.

Таблица 25.1. Определение уровня приоритета на основе класса приоритета процесса и относительного приоритета потока

Относительный приоритет потока	Класс приоритета процесса					
	Idle	Below Normal	Normal	Above Normal	High	Realtime
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above Normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below Normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

К примеру, если поток с приоритетом Normal принадлежит процессу с приоритетом Normal, ему назначается уровень приоритета 8. Так как приоритет Normal

по умолчанию используется как для классов, так и для потоков, большинство потоков в системе имеют уровень приоритета 8.

Для потока с приоритетом `Normal` в высокоприоритетном процессе уровень приоритета равен 13. Если поменять класс приоритета на `Idle`, уровень приоритета потока снизится до 4. Помните, что приоритеты потоков связаны с классом приоритета процесса. При изменении последнего относительный приоритет потока остается без изменений, а вот уровень приоритета меняется.

Обратите внимание, что в таблице не фигурирует сочетание, при котором поток получает нулевой уровень приоритета. Как уже упоминалось, этот приоритет зарезервирован для потока обнуления страниц, поэтому система не позволяет присвоить его какому-то другому потоку. Недоступны также следующие уровни приоритета: 17, 18, 19, 20, 21, 27, 28, 29 и 30. Они зарезервированы под драйверы устройств, работающие в режиме ядра, а потому не присваиваются пользовательским приложениям. Обратите внимание, что поток в классе приоритета `Realtime` не может иметь уровень приоритета ниже 16. В то же время потоки в остальных классах приоритета не могут получить уровень выше 15.

ПРИМЕЧАНИЕ

Концепция классов приоритета процесса может навести на мысль, что Windows каким-то образом управляет очередностью процессов. Но очередность операционная система определяет только для потоков. Класс приоритета процесса является абстрактным понятием, помогающим логически сравнить одно запущенное приложение с остальными.

ВНИМАНИЕ

Лучше снизить приоритет одного потока, чем повысить приоритет другого. Обычно понижение приоритета требуется, если поток выполняет длительные вычисления, например компилирует код, проверяет орфографию, пересчитывает электронные таблицы и т. п. Повышать приоритет имеет смысл, если поток должен быстро отреагировать на какое-то событие, запуститься на короткий промежуток времени и вернуться в состояние ожидания. Потоки с высоким приоритетом большую часть своего существования находятся в режиме ожидания, не влияя на быстродействие всей системы. В качестве примера потока с высоким приоритетом можно упомянуть поток Проводника Windows (`Windows Explorer`), отслеживающий нажатие клавиши Windows пользователем. Проводник приостанавливает потоки с более низким приоритетом и немедленно выводит на экран меню. В процессе навигации поток Проводника Windows быстро отвечает на нажатия клавиш, обновляет меню и приостанавливается до следующего нажатия клавиши пользователем.

Обычно процесс получает класс приоритета в зависимости от того, каким процессом он был запущен. Большинство процессов иницируются Проводником Windows, присваивающим всем своим потомкам класс приоритета `Normal`. Управляемые приложения не могут владеть своими процесса-

ми, они запускаются в домене. Именно поэтому они не могут менять класс приоритета процесса, ведь это окажет влияние на весь запущенный в процессе код. К примеру, многие приложения ASP.NET выполняются в одном процессе, хотя каждое из них работает в собственном домене приложений. То же самое можно сказать о приложениях Silverlight, запускаемых в процессе интернет-браузера, или управляемых хранимых процедурах, запускаемых внутри процесса Microsoft SQL Server.

В то же время приложение может менять относительный приоритет своих потоков при помощи свойства `Priority` класса `Thread`, которому присваивается одно из пяти значений (`Lowest`, `BelowNormal`, `Normal`, `AboveNormal` или `Highest`), определенных в перечислении `ThreadPriority`. При этом точно так же, как Windows резервирует для себя нулевой уровень и уровень реального времени, CLR резервирует уровни приоритета `Idle` и `Time-Critical`. В настоящее время в CLR отсутствуют потоки с уровнем приоритета `Idle`, но в будущем ситуация может поменяться. При этом поток финализации, о котором шла речь в главе 21, выполняется на уровне приоритета `Time-Critical`. Соответственно, для разработчиков управляемых приложений остаются пять приоритетов потока: в табл. 25.1 это строки со второй (`Highest`) по шестую (`Lowest`).

ВНИМАНИЕ

В настоящее время большинство разработчиков приложений предпочитает игнорировать преимущества, предоставляемые концепцией приоритетов потоков. Тем не менее хотелось бы надеяться, что в будущем, когда процессоры будут загружены на 100 %, непрерывно выполняя полезную работу, именно приоритеты потоков позволят обеспечивать быстроедействие системы. К сожалению, сейчас конечные пользователи воспринимают высокую загрузку процессора как сигнал, что приложение вышло из-под контроля. В будущем мне хотелось бы, чтобы этот фактор воспринимался положительно, как знак того, что компьютер активно обрабатывает информацию для пользователя. Проблема в том, что если занять процессор обработкой потоков с уровнем приоритета 8, приложения могут начать недостаточно быстро реагировать на ввод данных пользователем. Надеюсь, что в будущей версии Диспетчера задач в отчете о загрузке процессора будет фигурировать также информация об уровнях приоритета потоков. Это гораздо лучше поможет в диагностике проблем.

Следует упомянуть о наличии в пространстве имен `System.Diagnostics` классов `Process` и `ProcessThread`. Они обеспечивают Windows просмотр процесса и потока соответственно. Эти классы предназначены для разработчиков, желающих написать сервисное приложение на управляемом коде или пытающихся оснастить свой код инструментами, помогающими в отладке. Именно поэтому данные классы попали в пространство имен `System.Diagnostics`. Для доступа к данным классам приложениям необходимы специальные права системы безопасности. Вы не сможете применить эти классы, к примеру, в приложениях Silverlight или ASP.NET.

Тем не менее приложения могут воспользоваться классами `AppDomain` и `Thread`, обеспечивающими просмотр средой CLR доменов и потоков. Для работы с этими классами по большей части не требуется специальных прав системы безопасности, хотя некоторые операции доступны только при наличии определенных привилегий.

Фоновые и активные потоки

В CLR все потоки делятся на активные (`foreground`) и фоновые (`background`). При завершении активных потоков в процессе CLR принудительно завершает также все запущенные на этот момент фоновые потоки. При этом завершение фоновых потоков происходит немедленно и без вбрасывания исключений.

Следовательно, активные потоки имеет смысл использовать для исполнения заданий, которые обязательно требуется завершить, например, для перемещения на диск данных из буфера обмена. Фоновые же потоки можно оставить для таких некритичных вещей, как пересчет ячеек электронных таблиц или индексирование записей. Ведь эта работа может быть продолжена и после перезагрузки приложения, а значит, нет необходимости насильно оставлять приложение работать, когда пользователь пытается его закрыть.

Ввести в CLR концепцию активных и фоновых потоков потребовалось для лучшей поддержки доменов приложений. Как вы знаете, в каждом домене может быть запущено отдельное приложение, при этом каждое такое приложение может иметь собственный фоновый поток. Даже если одно из приложений завершается, заставляя завершиться свой фоновый поток, среда CLR все равно должна функционировать, поддерживая остальные приложения. И только после того как все приложения со всеми своими фоновыми процессами будут завершены, можно будет уничтожить весь процесс.

Следующий код демонстрирует разницу между фоновым и активным потоками:

```
using System;
```

```
using System.Threading;
```

```
public static class Program {  
    public static void Main() {  
        // Создание нового потока (по умолчанию активного)  
        Thread t = new Thread(Worker);  
  
        // Превращение потока в фоновый  
        t.IsBackground = true;  
  
        t.Start(); // Старт потока  
        // В случае активного потока приложение будет работать около 10 секунд
```

```
// В случае фонового потока приложение немедленно прекратит работу
Console.WriteLine("Returning from Main");
}

private static void Worker() {
    Thread.Sleep(10000); // Имитация 10 секунд работы

    // Следующая строка выводится только для кода,
    // исполняемого активным потоком
    Console.WriteLine("Returning from Worker");
}
}
```

Поток можно превращать из активного в фоновый и обратно. Основной поток приложения и все потоки, в явном виде созданные путем конструирования объекта `Thread`, по умолчанию являются активными. А вот потоки пула по умолчанию являются фоновыми. Также потоки, создаваемые машинным кодом и попадающие в управляемую среду исполнения, помечаются как фоновые.

ВНИМАНИЕ

По возможности старайтесь избегать активных потоков. Однажды меня попросили определить, почему приложение никак не может завершить свою работу. Провозившись несколько часов, я понял, что причиной был компонент пользовательского интерфейса, в явном виде создающий активный поток. После того как компонент заставили использовать поток из пула, проблема была решена. А заодно повысилась и общая эффективность работы приложения.

Что дальше?

В этой главе мы рассмотрели основы работы с потоками. Надеюсь, вы усвоили, что поток исполнения является довольно дорогим ресурсом и использовать его следует крайне аккуратно. Лучше всего задействовать пул потоков среды CLR, который создает и уничтожает потоки автоматически. Пул предлагает набор потоков для решения различных задач, и некоторые из этих потоков вполне справятся с решением задач вашего приложения.

В главе 26 мы поговорим о том, каким образом пул потоков позволяет выполнять вычислительные операции. Глава 27 посвящена обсуждению того, как с помощью комбинации пула потоков и асинхронной модели программирования CLR выполнять операции ввода-вывода. Во многих сценариях для этих операций синхронизация потоков вам вообще не потребуется. Тем не менее остаются сценарии, в которых без синхронизации не обойтись. Конструкции, применяемые для синхронизации, и разница между ними рассматриваются в главах 28 и 29.

В заключение упомяну, что я интенсивно использовал потоки, начиная с первой бета-версии Windows NT 3.1, появившейся примерно в 1992 году. После выхода бета-версии .NET я начал создавать библиотеку классов, позволяющую упростить асинхронное программирование и синхронизацию потоков. Эту библиотеку (она называется Wintellect Power Threading Library) можно загрузить бесплатно. Существуют ее версии для обычной среды CLR, а также для Silverlight CLR и Compact Framework. Найти библиотеку, документацию и примеры кода можно по адресу <http://Wintellect.com/PowerThreading.aspx>. Там же находятся ссылки на форум поддержки и на видеоролики с примерами использования различных компонентов библиотеки.

Глава 26. Асинхронные вычислительные операции

В этой главе рассказывается о различных способах асинхронного выполнения операций. Для их реализации требуются дополнительные потоки. К вычислительным операциям, в частности, относятся компиляция кода, проверка орфографии, проверка грамматики, пересчет электронных таблиц, перекодирование аудио- и видеоданных, создание миниатюр изображений. Как видите, такие операции встречаются в финансовых и технических приложениях повсеместно.

Большинство приложений не так уж много времени уделяет обработке находящихся в памяти данных или вычислениям. Это легко проверить, открыв Диспетчер задач на вкладке Performance (Быстродействие). Загрузка процессора менее 100 % (а именно такая картина наблюдается в большинстве случаев), означает, что запущенные процессы не используют на полную мощность резервы всех ядер. Также это означает, что некоторые (если не все) потоки в процессах вообще не исполняются. Они ждут операции ввода или вывода, например срабатывания таймера, чтения данных из базы или записи данных в нее, нажатия клавиши на клавиатуре, перемещения указателя или нажатия кнопки мыши. При операциях ввода-вывода драйверы Microsoft Windows иницируют работу устройств, а сам процессор в это время не исполняет потоки, запущенные в системе. Именно поэтому диспетчер задач показывает низкую загрузку процессора.

Однако даже приложения, предназначенные для операций ввода-вывода, обрабатывают получаемые данные, поэтому распараллеливание вычислений может значительно повысить их пропускную способность. В этой главе рассказывается о пуле потоков общезыковой исполняющей среды и основных приемах его использования. Это крайне важная информация, так как пул потоков является ключевой технологией, обеспечивающей разработку и реализацию масштабируемых, быстро реагирующих и надежных приложений и компонентов. Также в этой главе рассказывается о механизмах, позволяющих выполнять вычислительные операции посредством пула потоков. Эти операции происходят в асинхронном режиме, что позволяет, во-первых, обеспечить быструю реакцию на действия пользователей приложений с графическим интерфейсом, во-вторых, распределить занимающие много времени вычисления между различными процессорами.

Пул потоков в CLR

Как было отмечено в предыдущей главе, создание и уничтожение потока занимает изрядное время. Кроме того, при наличии множества потоков впус-тую

расходуется память и снижается производительность, ведь операционной системе приходится планировать исполнение потоков и выполнять переключения контекста. К счастью, среда CLR способна управлять собственным пулом потоков. Пул можно представить в виде набора доступных приложениям потоков. Для каждого процесса существует свой пул, используемый всеми доменами приложений в CLR. Если в один процесс загружаются несколько копий CLR, для каждой из них формируется собственный пул.

При инициализации CLR пул потоков пуст. Внутренне он должен поддерживать очередь запросов на обработку. Для выполнения приложением асинхронной операции вызывается метод, размещающий соответствующий запрос в очереди пула потоков. Код пула извлекает записи из очереди и распределяет их среди потоков из пула. Если пул пуст, создается новый поток. Как уже отмечалось, создание потока отрицательно сказывается на производительности. Однако по завершении исполнения своего задания поток не уничтожается, а возвращается в пул и ожидает следующего запроса. Поскольку поток не уничтожается, производительность не страдает.

Когда приложение отправляет пулу много запросов, он пытается обслужить их все с помощью одного потока. Однако если приложение создает очередь запросов быстрее, чем поток из пула их обслуживает, создаются дополнительные потоки. Такой подход позволяет обойтись при обработке запросов небольшим количеством потоков.

Когда приложение прекращает отправлять запросы в пул, появляются незанятые потоки, впустую занимающие память. Поэтому через некоторое время бездействия (различное для разных версий CLR) поток пробуждается и самоуничтожается, освобождая ресурсы. Это опять отрицательно сказывается на производительности, но в данном случае это уже не столь важно, поскольку уничтожаемый поток все равно простаивал, а значит, приложение в данный момент не было особо загружено работой.

Пул потоков позволяет найти золотую середину в ситуации, когда малое количество потоков экономит ресурсы, а большое позволяет воспользоваться преимуществами многопроцессорных систем, а также многоядерных и гиперпотоковых процессоров. Пул потоков действует по эвристическому алгоритму. Если приложение должно выполнить множество заданий и при этом имеются доступные процессоры, пул создает больше потоков. При снижении загрузки приложения потоки из пула самоуничтожаются.

В пуле различают два типа потоков: *рабочие потоки* (worker thread) и *потоки ввода-вывода* (I/O thread). Первые используются, когда приложение требует от пула выполнения асинхронной вычислительной операции (которая, в частности, может инициировать процедуры ввода-вывода). Вторые же служат для уведомления кода о завершении асинхронной операции ввода-вывода. Это значит, что для выполнения запросов, например обращения к файлу, сетевому серверу, базе данных, веб-службе или аппаратному устройству, должна исполь-

зоваться асинхронная модель программирования (Asynchronous Programming Model, APM).

Простые вычислительные операции

Для добавления в очередь пула потоков асинхронных вычислительных операций обычно вызывают один из следующих методов класса `ThreadPool`:

```
static Boolean QueueUserWorkItem(WaitCallback callBack);  
static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);
```

Эти методы ставят «рабочий элемент» вместе с дополнительными данными состояния в очередь пула потоков и сразу возвращают управление приложению. Рабочим элементом называется указанный в параметре `callback` метод, который будет вызван потоком из пула. Этому методу можно передать один параметр через аргумент `state` (данные состояния). Без этого параметра версия метода `QueueUserWorkItem` передает методу обратного вызова значение `null`. Все заканчивается тем, что один из потоков пула обработает рабочий элемент, приводя к вызову указанного метода. Создаваемый метод обратного вызова должен соответствовать делегату `System.Threading.WaitCallback`, который определяется так: `delegate void WaitCallback(Object state);`

ПРИМЕЧАНИЕ

Сигнатуры делегатов `WaitCallback` и `TimerCallback` (о них мы поговорим в этой главе), а также делегата `ParameterizedThreadStart` (он упоминался в главе 25) совпадают. Если вы определяете метод, совпадающий с этой сигнатурой, он может быть вызван через метод `ThreadPool.QueueUserWorkItem` при помощи объекта `System.Threading.Timer` или `System.Threading.Thread`.

Вот пример процедуры асинхронного вызова метода потоком из пула:

```
using System;  
using System.Threading;  
  
public static class Program {  
    public static void Main() {  
        Console.WriteLine("Main thread: queuing an asynchronous operation");  
        ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);  
        Console.WriteLine("Main thread: Doing other work here...");  
        Thread.Sleep(10000); // Имитация другой работы (10 секунд)  
        Console.WriteLine("Hit <Enter> to end this program...");  
        Console.ReadLine();  
    }  
}
```

продолжение ➤

```
// Сигнатура метода совпадает с сигнатурой делегата WaitCallback
private static void ComputeBoundOp(Object state) {
    // Метод выполняется потоком из пула
    Console.WriteLine("In ComputeBoundOp: state={0}", state);
    Thread.Sleep(1000); // Имитация другой работы (1 секунда)

    // После возвращения управления методом поток
    // возвращается в пул и ожидает следующего задания
}
}
```

Результат компиляции и запуска этого кода:

```
Main thread: queuing an asynchronous operation
Main thread: Doing other work here...
In ComputeBoundOp: state=5
```

Впрочем, возможен и такой результат:

```
Main thread: queuing an asynchronous operation
In ComputeBoundOp: state=5
Main thread: Doing other work here...
```

Разный порядок следования строк в данном случае объясняется асинхронным выполнением методов. Порядок исполнения потоков определяет планировщик Windows. При запуске приложения на многопроцессорном компьютере он допускает их одновременное выполнение.

ПРИМЕЧАНИЕ

Если метод обратного вызова вбрасывает необработанное исключение, CLR завершает процесс (если это не противоречит политике хоста). Необработанные исключения обсуждались в главе 20.

Контексты исполнения

С каждым потоком связан определенный контекст исполнения. Он включает в себя параметры безопасности (сжатый стек, свойство `Principal` объекта `Thread` и идентификационные данные пользователя Windows), параметры хоста (`System.Threading.HostExecutionContextManager`) и контекстные данные логического вызова (см. методы `LogicalSetData` и `LogicalGetData` класса `System.Runtime.Remoting.Messaging.CallContext`). Когда поток исполняет код, значения параметров контекста исполнения оказывают влияние на некоторые операции. В идеале всякий раз при использовании для выполнения заданий вспомогательного потока в этот вспомогательный поток должен копироваться контекст исполнения первого потока. Это гарантирует одинаковые параметры безопасности и хоста в обоих потоках, а также доступ вспомогательного потока к данным, сохраненным в контексте логического вызова исходного потока.

По умолчанию CLR автоматически копирует контекст исполнения самого первого потока во все вспомогательные потоки. Это гарантирует безопасность, но в ущерб производительности, потому что в контексте исполнения содержится много информации. Сбор всей информации и ее копирование во вспомогательные потоки занимает немало времени. Вспомогательный поток может, в свою очередь, использовать вспомогательные потоки, при этом создаются и инициализируются дополнительные структуры данных.

Класс `ExecutionContext` в пространстве имен `System.Threading` позволяет управлять копированием контекста исполнения потока. Вот как он выглядит:

```
public sealed class ExecutionContext : IDisposable, ISerializable {
    [SecurityCritical] public static AsyncFlowControl SuppressFlow();
    public static void RestoreFlow();
    public static Boolean IsFlowSuppressed();
    // Не показаны редко применяемые методы
}
```

С помощью этого класса можно запретить копирование контекста исполнения, повысив производительность приложения. Для серверных приложений рост производительности в этом случае оказывается весьма значительным. Для клиентских приложений особой выгоды нет, кроме того, метод `SuppressFlow` помечается атрибутом `[SecurityCritical]`, в результате становится невозможным вызов некоторых клиентских приложений (например, `Silverlight`). Разумеется, запрещать копирование контекста исполнения можно, только если вспомогательному потоку не требуется содержащаяся там информация. Когда иницируемый контекст исполнения не копируется во вспомогательный поток, тот обслуживается в рамках последнего сохраненного в нем контекста исполнения. Поэтому при отключенном копировании контекста поток не должен исполнять код на основе состояния текущего контекста исполнения (например, идентификационных данных пользователя `Windows`).

Вот пример, демонстрирующий, как запрет на копирование контекста исполнения влияет на данные в контексте логического вызова потока при постановке рабочего элемента в очередь в CLR-пуле¹:

```
public static void Main() {
    // Помещаем данные в контекст логического вызова потока метода Main
    CallContext.LogicalSetData("Name", "Jeffrey");

    // Заставляем поток из пула работать
    // Поток из пула имеет доступ к данным контекста логического вызова
    ThreadPool.QueueUserWorkItem(
```

продолжение ➤

¹ Добавляемые к контексту логического вызова элементы должны быть сериализуемыми (см. главу 24). Копирование контекста исполнения, содержащего данные контекста логического вызова, крайне отрицательно сказывается на производительности, так как требует сериализации и десериализации всех элементов данных.

```

state => Console.WriteLine("Name={0}",
    CallContext.LogicalGetData("Name")));

// Запрещаем копирование контекста исполнения потока метода Main
ExecutionContext.SuppressFlow();
// Заставляем поток из пула работать
// Поток из пула НЕ имеет доступа к данным контекста логического вызова
ThreadPool.QueueUserWorkItem(
    state => Console.WriteLine(
        "Name={0}", CallContext.LogicalGetData("Name")));

// Восстанавливаем копирование контекста исполнения потока метода Main
// на случай будущей работы с другими потоками из пула
ExecutionContext.RestoreFlow();
...
}

```

Результат компиляции и запуска этого кода:

```

Name=Jeffrey
Name=

```

Пока мы обсуждаем только запрет копирования контекста исполнения при вызове метода `ThreadPool.QueueUserWorkItem`, но этот прием используется как при работе с объектами `Task` (см. раздел «Задания» данной главы), так и при иницировании асинхронных операций ввода-вывода (о них речь идет в главе 27).

Скоординированная отмена

Платформа .NET предлагает стандартный эталон операций отмены. Этот эталон является *скоординированным* (cooperative), что указывает на необходимость явной поддержки отмены операций. Другими словами, как код, выполняющий отменяемую операцию, так и код, пытающийся реализовать отмену, должны относиться к типам, о которых рассказывается в этом разделе. Так как необходимость отмены занимающих много времени вычислительных операций не вызывает сомнения, к вашим вычислительным операциям имеет смысл добавить возможность отмены.

О том, как это сделать, мы и поговорим в этом разделе. Но начать следует с описания двух основных типов из библиотеки FCL, входящих в состав стандартного эталона скоординированной отмены.

Для начала вам потребуется объект `System.Threading.CancellationTokenSource`. Вот как выглядит данный класс:

```

public sealed class CancellationTokenSource : IDisposable { // Ссылочный тип
    public CancellationTokenSource();
    public void Dispose(); // Освобождает ресурсы (как WaitHandle)
}

```

```

public Boolean IsCancellationRequested { get; }
public CancellationToken Token { get; }

public void Cancel(); // Методу Cancel передается false
public void Cancel(Boolean throwOnFirstException);
...
}

```

Этот объект содержит все состояния, необходимые для управляемой отмены. После создания объекта `CancellationTokenSource` (ссылочный тип) получить один или несколько экземпляров `CancellationToken` (значимый тип) можно из свойства `Token`. Затем они передаются операциям, которые могут быть отменены. Вот наиболее полезные члены значимого типа `CancellationToken`:

```

public struct CancellationToken { // Значимый тип
    // IsCancellationRequested вызван операциями, не связанными с заданием
    public Boolean IsCancellationRequested { get; }

    public void ThrowIfCancellationRequested(); // Вызван операциями,
                                                // связанными с заданием

    // WaitHandle получает сигнал об отмене объекта CancellationTokenSource
    public WaitHandle WaitHandle { get; }
    // Члены GetHashCode, Equals, == и != не показаны

    public static CancellationToken None { get; }
    public Boolean CanBeCanceled { get; } // Редко используется

    public CancellationTokenRegistration Register(
        Action<Object> callback, Object state,
        Boolean useSynchronizationContext); // Более простые варианты
                                                // перегрузки не показаны
}

```

Экземпляр `CancellationToken` относится к упрощенному значимому типу, так как содержит всего одно закрытое поле: ссылку на свой объект `CancellationTokenSource`. Цикл вычислительной операции может периодически обращаться к свойству `IsCancellationRequested` объекта `CancellationToken`, чтобы узнать, не требуется ли раннее завершение его работы, то есть прерывание операции. Процессор перестает совершать операции, в результате которых вы не заинтересованы. Рассмотрим пример кода:

```

internal static class CancellationDemo {
    public static void Go() {
        CancellationTokenSource cts = new CancellationTokenSource();

        // Передаем операции CancellationToken и число
        ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));
    }
}

```

```
Console.WriteLine("Press <Enter> to cancel the operation.");
Console.ReadLine();
cts.Cancel(); // Если метод Count уже вернул управления,
              // Cancel не оказывает никакого эффекта
// Cancel немедленно возвращает управление, и метод продолжает работу...
}

private static void Count(CancellationToken token, Int32 countTo) {
    for (Int32 count = 0; count < countTo; count++) {
        if (token.IsCancellationRequested) {
            Console.WriteLine("Count is cancelled");
            break; // Выход из цикла для остановки операции
        }

        Console.WriteLine(count);
        Thread.Sleep(200); // Для демонстрационных целей просто ждем
    }
    Console.WriteLine("Count is done");
}
}
```

ПРИМЕЧАНИЕ

Чтобы предотвратить отмену операции, ей можно передать экземпляр `CancellationToken`, возвращенный статическим свойством `None` структуры `CancellationToken`. Этот экземпляр не связан ни с каким объектом `CancellationTokenSource` (его закрытое поле имеет значение `null`). При отсутствии объекта `CancellationTokenSource` отсутствует и код, который может вызвать метод `Cancel`. А значит, запрос к свойству `IsCancellationRequested` упомянутого экземпляра `CancellationToken` всегда будет получать в ответ значение `false`. Аналогичная ситуация с запросом к свойству `CanBeCanceled`. Значение `true` возвращается только для экземпляров `CancellationToken`, полученных через свойство `Token` перечисления `CancellationTokenSource`.

При желании можно зарегистрировать один или несколько методов таким образом, чтобы они вызывались при отмене объекта `CancellationTokenSource`. Но для регистрации каждого метода обратного вызова вам потребуется метод `Register` структуры `CancellationToken`. Этому методу передается делегат `Action<Object>`, состояние, которое вы предполагаете передать через делегат в метод обратного вызова, и значение типа `Boolean`, указывающее, вызывать ли делегат при помощи параметра `SynchronizationContext` вызываемого потока. Если передать параметру `useSynchronizationContext` значение `false`, поток, вызывающий метод `Cancel`, последовательно запустит все зарегистрированные методы. При передаче же значения `true` обратные вызовы отсылаются фиксированному объекту `SynchronizationContext`, который выбирает, какой из потоков активи-

зирует тот или иной обратный вызов. Подробно класс `SynchronizationContext` рассматривается в главе 27.

ПРИМЕЧАНИЕ

Если вы регистрируете метод обратного вызова, используя уже отмененный объект `CancellationTokenSource`, поток, вызывающий метод `Register`, активизирует обратный вызов (если параметру `useSynchronizationContext` было передано значение `true`, это, вероятно, произойдет путем вызова данного параметра потока).

Многократный вызов метода `Register` приводит к многократной же активизации методов обратного вызова. Причем последние могут вбрасывать необработанное исключение. Если передать методу `Cancel` объекта `CancellationTokenSource` значение `true`, первый же метод обратного вызова, ставший источником необработанного исключения, остановит выполнение остальных методов обратного вызова, а исключение будет также брошено методом `Cancel`. Если же передать этому методу значение `false`, будут вызваны все зарегистрированные методы обратного вызова. Все появляющиеся при этом необработанные исключения, добавляются в коллекцию. Если после завершения всех методов обратного вызова обнаруживается наличие необработанных исключений метод `Cancel` вбрасывает исключение `AggregateException`, свойству `InnerExceptions` которого присваивается коллекция брошенных объектов исключений. При отсутствии необработанных исключений метод `Cancel` просто возвращает управление.

ВНИМАНИЕ

Не существует способа определить, с какой операцией связан тот или иной объект из коллекции `InnerExceptions` исключения `AggregateException`. То есть вы фактически получаете только информацию о том, что некоторые операции выполнены не были, и по типу исключения можете определить, в чем была причина такого поведения. Чтобы выяснить местоположение ошибки, нужно исследовать свойство `StackTrace` объекта исключения и вручную проверить исходный код.

Метод `Register` объекта `CancellationToken` возвращает структуру `CancellationTokenRegistration`, которая выглядит следующим образом:

```
public struct CancellationTokenRegistration :  
    IEquatable<CancellationTokenRegistration>, IDisposable {  
    public void Dispose();  
    // Не показаны GetHashCode, Equals, операторы == и !=  
}
```

Метод `Dispose` позволяет удалить из объекта `CancellationTokenSource` зарегистрированный обратный вызов, с которым связан данный объект. В резуль-

тате при вызове метода `Cancel` этот обратный вызов игнорируется. Вот код, демонстрирующий регистрацию двух обратных вызовов с одним объектом `CancellationTokenSource`:

```
var cts = new CancellationTokenSource();
cts.Token.Register(() => Console.WriteLine("Canceled 1"));
cts.Token.Register(() => Console.WriteLine("Canceled 2"));

// Для проверки отменим его и выполним оба обратных вызова
cts.Cancel();
```

Вот результат работы такого кода, полученный сразу после вызова метода `Cancel`:

```
Canceled 2
Canceled 1
```

Наконец, можно создать новый объект `CancellationTokenSource`, связав друг с другом другие объекты `CancellationTokenSource`. Отмена этого нового объекта произойдет при отмене *любого* из входящих в его состав объектов. Вот демонстрирующий это код:

```
// Создание объекта CancellationTokenSource
var cts1 = new CancellationTokenSource();
cts1.Token.Register(() => Console.WriteLine("cts1 canceled"));

// Создание второго объекта CancellationTokenSource
var cts2 = new CancellationTokenSource();
cts2.Token.Register(() => Console.WriteLine("cts2 canceled"));

// Создание нового объекта CancellationTokenSource,
// отменяемого при отмене cts1 или cts2
var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(
    cts1.Token, cts2.Token);
linkedCts.Token.Register(() => Console.WriteLine("linkedCts canceled"));

// Отмена одного из объектов CancellationTokenSource (я выбрал cts2)
cts2.Cancel();

// Показываем, какой из объектов CancellationTokenSource был отменен
Console.WriteLine("cts1 canceled={0}, cts2 canceled={1}, linkedCts={2}",
    cts1.IsCancellationRequested, cts2.IsCancellationRequested,
    linkedCts.IsCancellationRequested);
```

Результат запуска этого кода:

```
linkedCts canceled
cts2 canceled
cts1 canceled=False, cts2 canceled=True, linkedCts=True
```

Задания

Вызвать метод `QueueUserWorkItem` класса `ThreadPool` для запуска асинхронных вычислительных операций очень просто. Однако этот подход имеет множество ограничений. Самой большой проблемой является отсутствие встроенного механизма, позволяющего узнать о завершении операции и получить возвращаемое значение. Для обхода этих и других ограничений специалисты Microsoft ввели понятие *заданий (tasks)*, выполнение которых осуществляется посредством типов из пространства имен `System.Threading.Tasks`.

Вот каким образом при помощи заданий совершить действие, аналогичное вызову метода `QueueUserWorkItem` класса `ThreadPool`:

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5); // Вызов QueueUserWorkItem
new Task(ComputeBoundOp, 5).Start();             // Аналог предыдущей строки
```

После создания нового объекта `Task` немедленно вызывается метод `Start` для запуска задания. Причем оба этих действия могут быть выполнены позже. Можно также представить код, передающий созданный им объект `Task` какому-то стороннему методу, который и будет определять момент вызова метода `Start`.

Для создания объекта `Task` требуется вызвать конструктор и передать в него делегат `Action` или `Action<Object>`, указывающий, какую операцию вы хотите выполнить. При передаче метода, ожидающего тип `Object`, в конструктор объекта `Task` следует передать также аргумент, который должен быть в итоге передан операции. При желании конструктору можно передать еще и структуру `CancellationToken`, позволяющую отменить объект `Task` до его выполнения (эта процедура подробно рассмотрена далее).

При желании конструктору можно передавать флаги из перечисления `TaskCreationOptions`, управляющие способами выполнения заданий. Данное перечисление может обрабатываться и как битовое поле. Оно определяется следующим образом:

```
[Flags, Serializable]
public enum TaskCreationOptions {
    None = 0x0000, // По умолчанию

    // Заставляет используемый по умолчанию планировщик заданий
    // поместить задание в глобальную очередь пула потока вместо
    // локальной очереди рабочего потока
    PreferFairness = 0x0001,

    // Это подсказка для планировщика заданий, показывающая,
    // как ее интерпретировать
    // В настоящее время планировщик заданий создает поток
    // для задания, а не помещает его в очередь в пуле,
```

продолжение ➤

```
// но в будущем это поведение может измениться
LongRunning = 0x0002,

// Всегда принимается на обработку: присоединяет задание к его родителю
AttachedToParent = 0x0004,
}
```

Большинство этих флагов являются подсказками, которые могут использоваться, а могут и игнорироваться объектом `TaskScheduler`, являющимся планировщиком заданий; всегда принимается к выполнению только флаг `AttachedToParent`, который никак не связан с самим объектом `TaskScheduler`. Более подробно про этот объект мы поговорим чуть позже.

Завершение задания и получение результата

Можно дождаться завершения задания и после этого получить результат его выполнения. Рассмотрим метод `Sum`, который при больших значениях переменной `n` требует большой вычислительной мощности:

```
private static Int32 Sum(Int32 n) {
    Int32 sum = 0;
    for (; n > 0; n--)
        checked { sum += n; } // При больших n появляется System.OverflowException
    return sum;
}
```

Можно создать объект `Task<TResult>` (производный от объекта `Task`) и в качестве универсального аргумента `TResult` передать тип результата, возвращаемого вычислительной операцией. Затем остается дождаться завершения выполняющегося задания и получить результат при помощи следующего кода:

```
// Создание задания Task (оно пока не выполняется)
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

// Можно начать выполнение задания через некоторое время
t.Start();

// Можно ожидать завершения задания в явном виде
t.Wait(); // ПРИМЕЧАНИЕ. Существует перегруженная версия,
// принимающая timeout/CancellationToken

// Получение результата (свойство Result вызывает метод Wait)
Console.WriteLine("The Sum is: " + t.Result); // Значение Int32
```

ВНИМАНИЕ

При вызове потоком метода `Wait` система проверяет, началось ли выполнение задания `Task`, которого ожидает поток. В случае положительного результата проверки поток, вызывающий метод `Wait`, блокируется до за-

вершения задания. Но если задание еще не начало выполняться, система может (в зависимости от объекта `TaskScheduler`) выполнить его при помощи потока, вызывающего метод `Wait`. В этом случае данный поток не блокируется. Он выполняет задание `Task` и немедленно возвращает управление. Это снижает затраты ресурсов (вам не приходится создавать поток взамен заблокированного), повышает производительность (на создание потока и переключение контекста не тратится время). Однако и это может быть не очень хорошо. Например, если перед вызовом метода `Wait` в рамках синхронизации потока происходит его записание, а затем задание пытается получить доступ к тем же запертым ресурсам, возникает так называемый намертво запертый поток (`deadlocked thread`)!

Если вычислительное задание вбрасывает необработанное исключение, оно «проглатывается», сохраняется в коллекции, а потоку пула разрешается вернуться в пул. Затем при вызове метода `Wait` или свойства `Result` эти члены вбрасывают исключение `System.AggregateException`.

Тип `AggregateException` служит для инкапсуляции коллекции исключений (которые вбрасываются, если родительское задание порождает многочисленные дочерние задания, приводящие к исключениям). Он содержит свойство `InnerExceptions`, возвращающее объект `ReadOnlyCollection<Exception>`. Не следует путать его со свойством `InnerException`, наследуемым классом `AggregateException` от своего базового класса `System.Exception`. Скажем, в показанном ранее примере элемент 0 свойства `InnerExceptions` класса `AggregateException` будет ссылаться на объект `System.OverflowException`, порождаемый вычислительным методом (`Sum`).

Для удобства класс `AggregateException` переопределяет метод `GetBaseException` класса `Exception`. Эта реализация возвращает самое глубоко спрятанное исключение, которое и считается источником проблемы (предполагается, что в коллекции всего одно самое глубоко спрятанное исключение). Класс `AggregateException` также предлагает метод `Flatten`, создающий новый экземпляр `AggregateException`, свойство `InnerExceptions` которого содержит список исключений, вброшенных после перебора внутренней иерархии исключений первоначального объекта. Ну и наконец, данный класс содержит метод `Handle`, вызывающий для каждого из исключений в составе `AggregateException` метод обратного вызова. Этот метод выбирает способ обработки исключения. Для обрабатываемых исключений он возвращает значение `true`, для необрабатываемых, соответственно, — `false`. Если после вызова метода `Handle` остается хотя бы одно необработанное исключение, создается новый объект `AggregateException`. Впрочем, с методами `Flatten` и `Handle` мы подробно познакомимся чуть позже.

ВНИМАНИЕ

Если вы ни разу не вызывали методы `Wait` или `Result` и не обращались к свойству `Exception` класса `Task`, код не «узнает» о появившихся исключениях. То есть вы не получите информации о том, что программа столкнулась с неожиданной проблемой. Для предотвращения подобной ситуации при

попадании объекта Task в сферу действия сборщика мусора метод Finalize объекта Task проверяет наличие исключений, и в случае их обнаружения вбрасывается исключение AggregateException. Вы не можете перехватить исключение, вброшенное потоком финализации сборщика мусора CLR, поэтому процесс немедленно завершается. Поэтому следует внести исправления в код, вызвав один из упомянутых членов и гарантировав, что код «увидит» исключение и сможет после восстановиться.

Для распознавания скрытых исключений можно зарегистрировать метод обратного вызова со статическим событием UnobservedTaskException класса TaskScheduler. При попадании задания со скрытым исключением в сферу действия сборщика мусора это событие активизируется потоком финализации сборщика мусора CLR. После этого обработчику события передается объект UnobservedTaskEventArgs, содержащий скрытое исключение AggregateException. Чтобы предотвратить завершение процесса, имеет смысл вызвать метод SetObserved объекта UnobservedTaskEventArgs, информирующий об обработке исключения. Впрочем, не следует вводить это в стандартную практику. Как отмечено в главе 20, процесс лучше завершить, чем позволить ему работать в поврежденном состоянии.

Можно ожидать завершения не только одного задания, но и массива объектов Task. Для этого в одноименном классе существует два статических метода. Метод WaitAny блокирует вызов потоков до завершения выполнения всех объектов в массиве Task. Этот метод возвращает индекс типа Int32 в массив, содержащий завершенные задания, заставляя поток продолжить исполнение. Если истекает время ожидания, метод возвращает значение -1. Отмена же метода посредством структуры CancellationToken приводит к исключению OperationCanceledException.

Второй аналогичный метод класса Task называется WaitAll. Он также блокирует вызывающий поток до завершения всех объектов Task в массиве. Метод возвращает значение true после завершения всех объектов и значение false, если истекает время ожидания. Отмена этого метода посредством структуры CancellationToken также приводит к исключению OperationCanceledException.

Отмена задания

Для отмены задания можно воспользоваться объектом CancellationTokenSource. Впрочем, сначала нам следует отредактировать метод Sum, дав ему возможность работать со структурой CancellationToken:

```
private static Int32 Sum(CancellationToken ct, Int32 n) {  
    Int32 sum = 0;  
    for (; n > 0; n--) {  
        // Следующая строка приводит к исключению OperationCanceledException  
        // при вызове метода Cancel для объекта CancellationTokenSource.
```

```
// на который ссылается маркер
ct.ThrowIfCancellationRequested();

checked { sum += n; } // при больших n вбрасывается
                      // исключение System.OverflowException
}
return sum;
}
```

В этом коде цикл вычислительной операции периодически вызывает метод `ThrowIfCancellationRequested` класса `CancellationToken`, чтобы проверить, не появился ли запрос на отмену операции. Этот метод аналогичен свойству `IsCancellationRequested` класса `CancellationToken`, рассмотренному ранее. Однако при отмене объекта `CancellationTokenSource` метод вбрасывает исключение `OperationCanceledException`. Причиной исключения становится тот факт, что в отличие от рабочих элементов, запущенных методом `QueueUserWorkItem` класса `ThreadPool`, задания имеют отметку о выполнении и даже могут возвращать значение. Следовательно, нужен способ, позволяющий отличить законченное задание от незаконченного. Именно для этого применяется исключение.

Создадим объекты `CancellationTokenSource` и `Task`:

```
CancellationTokenSource cts = new CancellationTokenSource();
Task<Int32> t = new Task<Int32>(() => Sum(cts.Token, 10000), cts.Token);

t.Start();

// Позднее отменим CancellationTokenSource, чтобы отменить Task
cts.Cancel(); // Это асинхронный запрос, задача уже может быть завершена

try {
    // В случае отмены задания метод Result вбрасывает
    // исключение AggregateException
    Console.WriteLine("The sum is: " + t.Result); // Значение Int32
}
catch (AggregateException x) {
    // Считаем обработанными все объекты OperationCanceledException
    // Все остальные исключения попадают в новый объект AggregateException,
    // состоящий только из необработанных исключений
    x.Handle(e => e is OperationCanceledException);

    // Строка выполняется, если все исключения уже обработаны
    Console.WriteLine("Sum was canceled");
}
```

Создаваемый объект `Task` можно связать с объектом `CancellationToken`, передав его конструктору `Task` (как показано ранее). Если отменить объект

CancellationToken до планирования задания, задание тоже будет отменено¹. Однако если задание уже начало выполняться (при помощи метода Start), его код должен в явном виде поддерживать отмену. К сожалению, несмотря на то что с объектом Task связан объект CancellationToken, у вас нет доступа к последнему. То есть вы должны каким-то образом поместить тот же самый объект CancellationToken, который использовался при создании объекта Task, в код задания. Проще всего при написании этого кода воспользоваться лямбда-выражением и «передать» объект CancellationToken в качестве переменной замыкания (что, собственно, и было сделано в предыдущем примере).

Автоматический запуск задания по завершении предыдущего

Для написания расширяемого программного обеспечения следует избегать блокировки потоков. Вызов метода Wait или запрос свойства Result при незавершенном задании приведет, скорее всего, к появлению в пуле нового потока, что увеличит расход ресурсов и отрицательно скажется на расширяемости. К счастью, существует способ узнать о завершении задания. Оно может просто инициировать выполнение следующего задания. Вот как следует переписать предыдущий код, чтобы избежать блокировки потоков:

```
// Создание объекта Task с отложенным запуском
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

// Задание можно запустить позднее
t.Start();

// Метод ContinueWith возвращает объект Task, но это не имеет значения
Task cwt = t.ContinueWith(task => Console.WriteLine(
    "The sum is: " + task.Result));
```

Теперь, как только задание, выполняющее метод Sum, завершится, оно иницирует выполнение следующего задания (также на основе потока из пула), которое выведет результат. Исполняющий этот код поток не блокируется, ожидая завершения каждого из указанных заданий; он может в это время исполнять какой-то другой код или, если это поток из пула, вернуться в пул для решения других задач. Обратите внимание, что выполняющее метод Sum задание может завершиться до вызова метода ContinueWith. Впрочем, это не проблема, так как метод ContinueWith заметит завершение задания Sum и немедленно начнет выполнение задания, отвечающего за вывод результата.

Также следует обратить внимание на то, что метод ContinueWith возвращает ссылку на новый объект Task (в моем коде она помещена в переменную cwt). При помощи этого объекта можно вызывать различные члены (например, метод

¹ Попытка отменить задание до начала его выполнения приводит к вбрасыванию исключения InvalidOperationException.

Wait, Result или даже ContinueWith), но обычно он просто игнорируется, а ссылка на него не сохраняется в переменной.

Следует также упомянуть, что внутренне объект Task представляет собой коллекцию ContinueWith. Это дает возможность несколько раз вызвать метод ContinueWith при помощи единственного объекта Task. Когда это задание завершится, все задания из коллекции ContinueWith окажутся в очереди в пуле потоков. Кроме того, при вызове метода ContinueWith можно установить флаги перечисления TaskContinuationOptions. Первые четыре флага — None, PreferFairness, LongRunning и AttachedToParent — аналогичны флагам показанного ранее перечисления TaskCreationOptions. Вот как выглядит тип TaskContinuationOptions:

```
[Flags, Serializable]
public enum TaskContinuationOptions {
    None = 0x0000, // По умолчанию

    // Заставляет планировщик заданий поместить задание в общую
    // очередь пула потоков, а не в локальную очередь рабочего потока
    PreferFairness = 0x0001,

    // Заставляет планировщик создать для задания поток,
    // а не помещать его в пул
    LongRunning = 0x0002,

    // Всегда выполняется: связывает объект Task с его родителем
    AttachedToParent = 0x0004,

    // Этот флаг устанавливают, когда требуется, чтобы поток,
    // выполняющий первое задание, выполнил и задание ContinueWith
    // Если первое задание уже завершено, поток, вызывающий ContinueWith,
    // выполняет задание ContinueWith
    ExecuteSynchronously = 0x80000,

    // Эти флаги указывают, когда запускать задание ContinueWith
    NotOnRanToCompletion = 0x10000,
    NotOnFaulted = 0x20000,
    NotOnCanceled = 0x40000,

    // Эти флаги являются комбинацией трех предыдущих
    OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted,
    OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,
    OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
}
```

При вызове метода ContinueWith флаг TaskContinuationOptions.OnlyOnCanceled показывает, что новое задание должно выполняться только в случае отмены предыдущего. Аналогично, флаг TaskContinuationOptions.OnlyOnFaulted дает понять, что выполнение нового задания должно начаться только после того, как первое

задание станет источником необработанного исключения. Ну а при помощи флага `TaskContinuationOptions.OnlyOnRanToCompletion` вы программируете запуск нового задания только при условии, что предыдущее задание не было отменено и не стало источником необработанного исключения, а было выполнено полностью. Без этих флагов новое задание запускается вне зависимости от того, как завершилось предыдущее. После завершения объекта `Task` автоматически отменяются все его вызовы с незапущенными заданиями. Вот пример, демонстрирующий сказанное:

```
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 10000);

// Задание можно запустить позже
t.Start();

// Каждый метод ContinueWith возвращает Task, но это не имеет значения
t.ContinueWith(task => Console.WriteLine("The sum is: " + task.Result),
    TaskContinuationOptions.OnlyOnRanToCompletion);

t.ContinueWith(task => Console.WriteLine("Sum threw: " + task.Exception),
    TaskContinuationOptions.OnlyOnFaulted);

t.ContinueWith(task => Console.WriteLine("Sum was canceled"),
    TaskContinuationOptions.OnlyOnCanceled);
```

Дочерние задания

Как демонстрирует данный код, задания поддерживают в числе прочего и отношения предок-потомок:

```
Task<Int32[]> parent = new Task<Int32[]>(() => {
    var results = new Int32[3]; // Создание массива для результатов

    // Это задание создает и запускает 3 дочерних задания
    new Task(() => results[0] = Sum(10000),
        TaskCreationOptions.AttachedToParent).Start();
    new Task(() => results[1] = Sum(20000),
        TaskCreationOptions.AttachedToParent).Start();
    new Task(() => results[2] = Sum(30000),
        TaskCreationOptions.AttachedToParent).Start();

    // Возвращается ссылка на массив
    // (элементы могут быть не инициализированы)
    return results;
});

// Вывод результатов после завершения родительского и дочерних заданий
var rcwt = parent.ContinueWith(
    parentTask => Array.ForEach(parentTask.Result, Console.WriteLine));
```

```
// Запуск родительского задания, которое запускает дочерние  
parent.Start();
```

Родительское задание создает и запускает три объекта `Task`. По умолчанию задания-потомки попадают на самый верхний уровень и не имеют отношения к своему предку. Однако при установке флага `TaskCreationOptions.AttachedToParent` родительское задание завершается только после завершения всех его потомков. Если при создании объекта `Task` методом `ContinueWith` установить флаг `TaskContinuationOptions.AttachedToParent`, то задание, запускаемое после завершения предыдущего, станет его потомком.

Структура задания

Каждый объект `Task` состоит из набора полей, определяющих состояние задания. Сюда относятся: идентификатор типа `Int32` (предназначенное только для чтения свойство `Id` объекта `Task`), значение типа `Int32`, представляющее состояние выполнения задания, ссылка на родительское задание, ссылка на объект `TaskScheduler`, показывающий время создания задания, ссылка на метод обратного вызова, ссылка на объект, который следует передать в метод обратного вызова (этот объект доступен через предназначенное только для чтения свойство `AsyncState` объекта `Task`), ссылка на класс `ExecutionContext` и ссылка на объект `ManualResetEventSlim`. Кроме того, каждый объект `Task` имеет ссылку на дополнительное состояние, создаваемое по требованию. Это дополнительное состояние включает в себя объект `CancellationToken`, коллекцию объектов `ContinueWithTask`, коллекцию объектов `Task` для дочерних заданий, ставших источником необработанных исключений, и прочее в том же духе. За все эти возможности приходится платить, так как для хранения каждого состояния требуется выделять место в памяти. Если дополнительные возможности вам не нужны, для более эффективного расходования ресурсов рекомендуем воспользоваться методом `ThreadPool.QueueUserWorkItem`.

Классы `Task` и `Task<TResult>` реализуют интерфейс `IDisposable`, что позволяет после завершения работы с объектом `Task` вызвать метод `Dispose`. Пока что этот метод всего лишь закрывает объект `ManualResetEventSlim`, но можно определить классы, производные от `Task` и `Task<TResult>`, которые будут выделять свои ресурсы, освобождаемые при помощи переопределенного метода `Dispose`. Разумеется, разработчики практически никогда не вызывают метод `Dispose` для объекта `Task`; они просто позволяют сборщику мусора удалить освободившиеся ресурсы.

Как легко заметить, у каждого объекта `Task` есть поле типа `Int32`, содержащее уникальный идентификатор задания. При создании объекта этому полю присваивается начальное значение ноль. При первом запросе к предназначенному только для чтения свойству `Id` этому свойству присваивается значение типа `Int32`, которое и возвращается в качестве результата запроса. Нумерация

идентификаторов начинается с единицы и увеличивается на единицу с каждым следующим присвоенным идентификатором. Для присваивания заданию идентификатора достаточно посмотреть на объект Task в отладчике приложения Microsoft Visual Studio.

Идентификаторы были введены, чтобы сопоставить каждому заданию уникальный номер. В Visual Studio их можно увидеть в окнах Parallel Tasks и Parallel Stacks. Но так как присвоение идентификаторов происходит автоматически, практически невозможно понять, какие значения к каким заданиям относятся. Тем не менее можно обратиться к статическому свойству CurrentId объекта Task, которое возвращает значение типа Int32, допускающего присвоение значений null (Int32?). Узнать идентификатор кода, отладка которого происходит в данный момент, можно также в окнах Watch и Immediate. После этого остается найти это задание в окне Parallel Tasks или Parallel Stacks. Если при запросе значения свойства CurrentId задание не выполнялось, возвращается null.

Узнать, на какой стадии своего жизненного цикла находится задание, можно при помощи предназначенного только для чтения свойства Status объекта Task. Оно возвращает значение TaskStatus, которое определяется следующим образом:

```
public enum TaskStatus {  
    // Флаги, обозначающие состояние задания:  
    Created,           // Задание создано в явном виде  
                      // и может быть запущено вручную  
    WaitingForActivation, // Задание создано неявно  
                      // и запускается автоматически  
  
    WaitingToRun,      // Задание запланировано, но еще не запущено  
    Running,           // Задание выполняется  
  
    // Задание ждет завершения дочерних заданий, чтобы завершиться  
    WaitingForChildrenToComplete,  
  
    // Возможные окончательные состояния задания:  
    RanToCompletion,  
    Canceled,  
    Faulted  
}
```

Только что созданный объект Task имеет статус Created. Позднее, когда задание ставится в очередь на выполнение, его статус меняется на WaitingToRun. Запущенному заданию в потоке присваивается статус Running. Приостановленному заданию, которое ожидает завершения дочерних заданий, соответствует статус WaitingForChildrenToComplete. Полностью завершенное задание имеет одно из трех возможных состояний: RanToCompletion, Canceled или Faulted. Узнать результат выполнения задания Task<TResult> можно через

ее свойство `Result`. Если выполнение задачи `Task` или `Task<TResult>` прерывается, узнать, какое именно необработанное исключение было вброшено, можно через свойство `Exception` объекта `Task`; оно всегда возвращает объект `AggregateException`, коллекция которого состоит из необработанных исключений.

Для удобства объект `Task` обладает набором предназначенных только для чтения свойств типа `Boolean`: `IsCanceled`, `IsFaulted` и `IsCompleted`. Последнее свойство возвращает значение `true`, если объект `Task` находится в состоянии `RanToCompleted`, `Canceled` или `Faulted`. Определить, успешно ли выполнено задание, проще всего при помощи вот такого кода:

```
if (task.Status == TaskStatus.RanToCompletion) ...
```

Объект `Task` оказывается в состоянии `WaitingForActivation`, если он создается при помощи одной из следующих функций: `ContinueWith`, `ContinueWhenAll`, `ContinueWhenAny` или `FromAsync`. Задание, созданное путем конструирования объекта `TaskCompletionSource<TResult>`, также оказывается в состоянии `WaitingForActivation`. Это состояние означает, что планирование задания управляется его собственной инфраструктурой. К примеру, невозможно явным образом запустить объект `Task`, созданный вызовом функции `ContinueWith`. Это задание запустится автоматически после завершения предыдущего.

Фабрики заданий

Иногда возникает необходимость получить набор объектов `Task`, находящихся в одном и том же состоянии. Для этого не нужно раз за разом передавать одни и те же параметры в конструктор каждого задания, достаточно создать *фабрику заданий* (*task factory*), инкапсулирующую нужное состояние. В пространстве имен `System.Threading.Tasks` определены типы `TaskFactory` и `TaskFactory<TResult>`. Оба этих типа являются производными от типа `System.Object`; то есть они являются равноправными.

Для создания группы заданий, не возвращающих значений, конструируется класс `TaskFactory`. Если же эти задания должны возвращать некое значение, потребуется класс `TaskFactory<TResult>`, в который в виде универсального аргумента `TResult` передается желаемый тип возвращаемого значения. При создании этих классов их конструкторам передаются параметры, которыми задания должны обладать по умолчанию. А точнее, передаются параметры `CancellationToken`, `TaskScheduler`, `TaskCreationOptions` и `TaskContinuationOptions`, наделяющие задания нужными свойствами.

Вот пример применения класса `TaskFactory`:

```
Task parent = new Task(() => {  
    varcts = new CancellationTokenSource();  
    var tf = new TaskFactory<Int32>(cts.Token,
```

продолжение ➤

```

    TaskCreationOptions.AttachedToParent,
    TaskContinuationOptions.ExecuteSynchronously,
    TaskScheduler.Default);

// Задание создает и запускает 3 дочерних задания
var childTasks = new[] {
    tf.StartNew(() => Sum(cts.Token, 10000)),
    tf.StartNew(() => Sum(cts.Token, 20000)),
    tf.StartNew(() => Sum(cts.Token, Int32.MaxValue)) // Исключение
                                                    // OverflowException
};

// Если дочернее задание становится источником исключения,
// отменяем все дочерние задания
for (Int32 task = 0; task < childTasks.Length; task++)
    childTasks[task].ContinueWith(
        t => cts.Cancel(), TaskContinuationOptions.OnlyOnFaulted);

// После завершения дочерних заданий получаем максимальное
// возвращенное значение и передаем его другому заданию
// для вывода
tf.ContinueWhenAll(
    childTasks,
    completedTasks => completedTasks.Where(
        t => !t.IsFaulted && !t.IsCanceled).Max(t => t.Result),
    CancellationToken.None)
    .ContinueWith(t => Console.WriteLine("The maximum is: " + t.Result),
    TaskContinuationOptions.ExecuteSynchronously);
});

// После завершения дочерних заданий выводим,
// в том числе, и необработанные исключения
parent.ContinueWith(p => {
    // Текст помещен в StringBuilder и однократно вызван
    // метод Console.WriteLine просто потому, что это задание
    // может выполняться параллельно с предыдущим,
    // и я не хочу путаницы в выводимом результате
    StringBuildersb = new StringBuilder(
        "The following exception(s) occurred:" + Environment.NewLine);

    foreach (var e in p.Exception.Flatten().InnerExceptions)
        sb.AppendLine(" " + e.GetType().ToString());
    Console.WriteLine(sb.ToString());
}, TaskContinuationOptions.OnlyOnFaulted);

// Запуск родительского задания, которое может запускать дочерние
parent.Start();

```

В этом коде создается объект `TaskFactory<Int32>`, при помощи которого потом создаются три объекта `Task`. При этом я хочу четырех вещей: чтобы все объекты `Task` обладали одним и тем же маркером `CancellationTokenSource`, чтобы все они имели одного родителя, чтобы для них всех использовался один и тот же заданный по умолчанию планировщик заданий и чтобы все они выполнялись одновременно.

Поэтому из трех объектов `Task`, созданных методом `StartNew` класса `TaskFactory`, формируется массив. Данный метод крайне удобен для создания и запуска дочерних заданий. В цикле каждое из дочерних заданий, ставшее источником необработанного исключения, отменяет все остальные запущенные в данный момент задания. Напоследок в классе `TaskFactory` вызывается метод `ContinueWhenAll`, создающий задание, выполняющееся после завершения всех дочерних заданий. Будучи создано в классе `TaskFactory`, это новое задание также считается дочерним и выполняется в синхронном режиме с помощью заданного по умолчанию планировщика. Но так как оно должно выполняться даже после отмены остальных дочерних заданий, его свойство `CancellationToken` переопределяется путем передачи ему значения `CancellationToken.None`. Это вообще исключает возможность отмены задания. Ну и после того, как обрабатывающее результаты задание завершает свою работу, создается еще одно задание, предназначенное для вывода максимального из возвращенных дочерними заданиями значения.

ПРИМЕЧАНИЕ

Вызов статических методов `ContinueWhenAll` и `ContinueWhenAny` классов `TaskFactory` или `TaskFactory<TResult>` делает недействительными следующие флаги `TaskContinuationOption`: `NotOnRanToCompletion`, `NotOnFaulted` и `NotOnCanceled`. Игнорируются и такие удобные флаги, как `OnlyOnCanceled`, `OnlyOnFaulted` и `OnlyOnRanToCompletion`. То есть методы `ContinueWhenAll` и `ContinueWhenAny` запускают следующее задание вне зависимости от того, каким оказывается результат выполнения предыдущего.

Планировщики заданий

Задания обладают очень гибкой инфраструктурой, причем не в последнюю очередь благодаря объектам `TaskScheduler`. Именно объект `TaskScheduler` отвечает за выполнение запланированных заданий и выводит информацию о них в отладчике Visual Studio. В FCL существует два производных от `TaskScheduler` типа: планировщик заданий в пуле потоков и планировщик заданий контекста синхронизации. По умолчанию все приложения используют первый из них, планирующий задания рабочих потоков в пуле (он подробно рассматривается чуть позже). Для получения ссылки на него используется статическое свойство `Default` класса `TaskScheduler`.

Планировщики заданий контекста синхронизации обычно применяются в приложениях Windows Forms, Windows Presentation Foundation (WPF) и Silverlight. Они планируют задания в потоке графического интерфейса приложения, обеспечивая оперативное обновление таких элементов интерфейса, как кнопки, пункты меню и т. п. Этот планировщик вообще никак не использует пул потоков. Получить на него ссылку можно с помощью статического метода `FromCurrentSynchronizationContext` класса `TaskScheduler`.

Вот простое приложение Windows Forms, демонстрирующее применение планировщика заданий контекста синхронизации:

```
internalsealed class MyForm : Form {
    public MyForm() {
        Text = "Synchronization Context Task Scheduler Demo";
        Visible = true; Width = 400; Height = 100;
    }

    // Получение ссылки на планировщик заданий
    private readonly TaskScheduler m_syncContextTaskScheduler =
        TaskScheduler.FromCurrentSynchronizationContext();

    private CancellationTokenSource m_cts;

    protected override void OnMouseClick(MouseEventArgs e) {
        if (m_cts != null) { // Операция начата, отменяем ее
            m_cts.Cancel();
            m_cts = null;
        } else { // Операция не начата, начинаем ее
            Text = "Operation running";
            m_cts = new CancellationTokenSource();

            // Это задание использует заданный по умолчанию планировщик
            // и выполняет поток из пула
            var t = new Task<Int32>(() => Sum(m_cts.Token, 20000), m_cts.Token);
            t.Start();

            // Эти задания используют планировщик контекста синхронизации
            // и выполняют поток графического интерфейса
            t.ContinueWith(task => Text = "Result: " + task.Result,
                CancellationToken.None,
                TaskContinuationOptions.OnlyOnRanToCompletion,
                m_syncContextTaskScheduler);

            t.ContinueWith(task => Text = "Operation canceled",
                CancellationToken.None, TaskContinuationOptions.OnlyOnCanceled,
                m_syncContextTaskScheduler);

            t.ContinueWith(task => Text = "Operation faulted",
                CancellationToken.None, TaskContinuationOptions.OnlyOnFaulted,
```

```
        m_syncContextTaskScheduler);  
    }  
    base.OnMouseClicked(e);  
}  
}
```

При щелчке на клиентской области данной формы в потоке пула начинается вычислительное задание. Это хорошо, так как означает, что GUI-поток не заблокирован и может реагировать на операции с пользовательским интерфейсом. При этом исполняемый потоком из пула код не должен пытаться обновлять элементы интерфейса. В противном случае будет выброшено исключение `InvalidOperationException`.

После завершения задания, связанного с вычислениями, начинает выполняться одно из трех следующих за ним заданий. Все эти задания обрабатываются планировщиком контекста синхронизации, причем этот планировщик ставит задания в очередь GUI-потока, позволяя коду этих заданий успешно обновлять элементы интерфейса. Обновление подписей на форме осуществляется через унаследованное свойство `Text`.

Так как вычислительное задание (метод `Sum`) запускается в потоке пула, пользователь может отменить операции при помощи элементов интерфейса. В моем примере для отмены операции достаточно щелкнуть на клиентской области формы.

Разумеется, при наличии специальных требований к планировщику можно определить собственный класс, производный от `TaskScheduler`. Microsoft предлагает множество фрагментов кода для заданий и различных планировщиков в пакете `Parallel Extensions Extras`, который можно загрузить по адресу <http://code.msdn.microsoft.com/ParExtSamples>. Там вы найдете, в частности, следующие планировщики:

- ❑ **`IOTaskScheduler`**. Ставит задания в очередь в потоках ввода-вывода пула, а не в рабочих потоках.
- ❑ **`LimitedConcurrencyLevelTaskScheduler`**. Позволяет одновременно выполняться не более чем *n* заданиям, где *n* — параметр конструктора.
- ❑ **`OrderedTaskScheduler`**. Разрешает выполнение только одного задания за раз. Данный класс является производным от `LimitedConcurrencyLevelTaskScheduler` и в качестве параметра *n* ему передается 1.
- ❑ **`PrioritizingTaskScheduler`**. Ставит задания в очередь в пуле потоков среды CLR. После этого можно вызвать метод `Prioritize` и указать, что задание должно быть обработано раньше всех остальных заданий (если это еще не сделано). Метод `Deprioritize`, соответственно, позволяет выполнить задание после всех прочих.
- ❑ **`ThreadPerTaskScheduler`**. Создает и запускает отдельный поток для каждого задания, при этом пул потоков не используется.

Методы For, ForEach и Invoke класса Parallel

Существуют известные программные конструкции, позволяющие воспользоваться повышением производительности, достигаемым посредством заданий. Для упрощения программирования эти сценарии инкапсулированы в статический класс `System.Threading.Tasks.Parallel`. Например:

```
// Один поток выполняет всю работу по очереди
for (Int32 i = 0; i < 1000; i++) DoWork(i);
```

Вместо обработки всех элементов этой коллекции можно воспользоваться методом `For` класса `Parallel` и распределить работу между несколькими потоками из пула:

```
// Потоки из пула выполняют работу параллельно
Parallel.For(0, 1000, i => DoWork(i));
```

Аналогично со следующей коллекцией:

```
// Один поток выполняет всю работу по очереди
foreach (var item in collection) DoWork(item);
```

Вместо этого кода можно написать:

```
// Потоки из пула выполняют работу параллельно
Parallel.ForEach(collection, item => DoWork(item));
```

Цикл `For` предпочтительней цикла `ForEach`, так как он работает быстрее.

Если вам нужно выполнить несколько методов, лучше запускать их последовательно. Например, вот так:

```
// Один поток выполняет методы по очереди
Method1();
Method2();
Method3();
```

Впрочем, их можно выполнить и параллельно:

```
// Потоки из пула выполняют методы одновременно
Parallel.Invoke(
    () => Method1(),
    () => Method2(),
    () => Method3());
```

Все методы класса `Parallel` заставляют вызывающий поток принимать участие в их обработке. Это хорошо с точки зрения расходования ресурсов, так как вызывающий поток не блокируется, ожидая выполнения работы потоками пула. Впрочем, если выполнение вызывающего потока будет закончено до завершения потоков из пула, вызывающий поток заблокирует сам себя, что тоже неплохо, так как обеспечивает семантику, аналогичную применению цикла `for`

или `foreach`. Выполнение вызывающего потока не возобновляется, пока не будет завершена вся работа. Если какая-либо операция станет источником необработанного исключения, вызванный вами метод `Parallel` вбросит исключение `AggregateException`.

Разумеется, не нужно заменять все циклы `for` в своем коде вызовами метода `Parallel.For`, а циклы `foreach` — вызовами метода `Parallel.ForEach`. Вызывая метод `Parallel`, вы предполагаете, что рабочие элементы без проблем смогут обслуживаться параллельно. Значит, для заданий, которые следует выполнять последовательно, вызов этого метода не имеет смысла. Следует также избегать рабочих элементов, вносящих изменения в любые совместно используемые данные, так как при одновременном управлении несколькими потоками эти данные могут оказаться поврежденными. Обычно эта проблема решается в рамках синхронизации потоков запирианием фрагментов кода, в которых реализуется доступ к данным. Однако так как после этого доступ к данным в каждый момент времени сможет получать только один поток, теряется преимущество одновременного обслуживания множества элементов.

Кроме того, методы класса `Parallel` потребляют много ресурсов — приходится выделять память под делегаты, которые вызываются по одному для каждого рабочего элемента. При наличии множества рабочих элементов, которые могут обслуживаться разными потоками, можно получить рост производительности. К тому же, если каждый элемент выполняет много работы, на снижение производительности из-за вызова делегатов можно не обращать внимания. Проблемы начинаются в случае, когда методы класса `Parallel` применяются к небольшому числу рабочих элементов или же к элементам, обслуживание которых происходит очень быстро.

Следует упомянуть, что для методов `For`, `ForEach` и `Invoke` класса `Parallel` существуют перегруженные версии, принимающие объект `ParallelOptions`. Вот как он выглядит:

```
public class ParallelOptions{
    public ParallelOptions();

    // Допускает отмену операций
    public CancellationTokenCancellationToken { get; set; }
    // Default=CancellationToken.None

    // Допускает указывать максимальное количество рабочих
    // элементов, обслуживаемых одновременно
    public Int32MaxDegreeOfParallelism { get; set; }
    // Default =-1 (число доступных процессоров)

    // Допускает выбор планировщика заданий
    public TaskSchedulerTaskScheduler { get; set; }
    // Default=TaskScheduler.Default
}
```

Существуют перегруженные версии и для методов `For` и `ForEach`, позволяющие передавать три делегата:

- ❑ Делегат локальной инициализации задания (`localInit`) для каждого выполняемого задания вызывается только один раз — перед получением команды на обслуживание рабочего элемента.
- ❑ Делегат `body` вызывается один раз для каждого элемента, обслуживаемого участвующими в процессе потоками.
- ❑ Делегат локального состояния каждого потока (`localFinally`) вызывается один раз для каждого задания после того, как оно обслужит все переданные ему рабочие элементы. Также он вызывается, если код делегата `body` становится источником необработанного исключения.

Вот пример кода, демонстрирующий использование этих трех делегатов для добавления во все файлы в папке дополнительных байтов:

```
private static Int64 DirectoryBytes(String path, String searchPattern,
    SearchOption searchOption) {
    var files = Directory.EnumerateFiles(path, searchPattern, searchOption);
    Int64 masterTotal = 0;

    ParallelLoopResult result = Parallel.ForEach<String, Int64>(
        files,

        () => { // localInit: вызывается в момент запуска задания
            // Устанавливает, что задача видит 0 байтов
            return 0; // Присваивает taskLocalTotal начальное значение 0
        },

        (file, loopState, index, taskLocalTotal) => { // body: Вызывается
                                                    // один раз для каждого элемента
            // Получает размер файла и добавляет его к общему размеру
            Int64 fileLength = 0;
            FileStream fs = null;
            try {
                fs = File.OpenRead(file);
                fileLength = fs.Length;
            }
            catch (IOException) { /* Игнорируем файлы, к которым нет доступа */ }
            finally { if (fs != null) fs.Dispose(); }
            return taskLocalTotal + fileLength;
        },

        taskLocalTotal => { // localFinally: Вызывается один раз в конце задания
            // Добавляем размер из задания к общему размеру
```

```
Interlocked.Add(ref masterTotal, taskLocalTotal);
});

return masterTotal;
}
```

Каждое задание управляет собственной промежуточной суммой (в переменной `taskLocalTotal`) для данных ей файлов. После того как все задания завершатся, в безопасном в отношении потоков режиме обновляется общая сумма. Для этого используется метод `Interlocked.Add` (подробно он рассматривается в главе 28). Так как промежуточная сумма для каждого задания своя, во время обработки элементов не требуется синхронизации потоков. И это хорошо, так как позволяет избежать проблем производительности. Они возникают только на последнем этапе при вызове метода `Interlocked.Add`. То есть снижение производительности происходит одновременно для задания, а не для рабочего элемента.

Думаю, вы обратили внимание, что делегат `body` был передан объекту `ParallelLoopState`:

```
public class ParallelLoopState{
    public void Stop();
    public BooleanIsStopped { get; }

    public void Break();
    public Int64? LowestBreakIteration{ get; }

    public BooleanIsExceptional { get; }
    public BooleanShouldExitCurrentIteration { get; }
}
```

Каждое принимающее участие в работе задание получает собственный объект `ParallelLoopState` и использует его для взаимодействия с другими работающими заданиями. Метод `Stop` останавливает цикл, и все будущие запросы к свойству `IsStopped` возвращают значение `true`. Метод `Break` заставляет цикл отказаться от обработки всех элементов, расположенных после выделенного. Предположим, что цикл `ForEach` должен обработать 100 элементов, но после обработки пятого элемента был вызван метод `Break`. В итоге цикл гарантированно обрабатывает первые пять элементов и возвращает управление. Впрочем, это не исключает обработки дополнительных элементов. Свойство `LowestBreakIteration` возвращает низшую итерацию цикла, на которой был вызван метод `Break`. Если этот метод вообще не вызвался, свойство возвращает значение `null`.

Свойство `IsException` возвращает значение `true`, если при обработке хотя бы одного элемента было выброшено необработанное исключение. Свойство `ShouldExitCurrentIteration` получает значение, указывающее, следует ли прервать текущую итерацию цикла. Оно используется в случаях, когда обработка элемента занимает много времени. Свойство возвращает значение `true`, если был

вызван метод `Stop` или `Break`, отменен объект `CancellationTokenSource` (ссылка на него дается через свойство `CancellationToken` класса `ParallelOption`) или же обработка элемента привела к необработанному исключению.

Методы `For` и `ForEach` класса `Parallel` возвращают экземпляр `ParallelLoopResult`, который выглядит так:

```
public struct ParallelLoopResult{  
    // Возвращает false в случае преждевременного завершения операции  
    public Boolean IsCompleted { get; }  
    public Int64? LowestBreakIteration { get; }  
}
```

Есть свойства, позволяющие выяснить результат работы цикла. Если свойство `IsCompleted` возвращает значение `true`, значит, цикл пройден полностью, и все элементы обработаны. Если свойство `IsCompleted` возвращает значение `false`, а свойство `LowestBreakIteration` — значение `null`, значит, каким-то из потоков был вызван метод `Stop`. Если же в последнем случае значение, возвращаемое свойством `LowestBreakIteration`, отлично от `null`, значит, каким-то из потоков был вызван метод `Break`. При этом возвращенное свойством `LowestBreakIteration` значение типа `Int64` указывает индекс последнего гарантированно обработанного элемента.

Встроенный язык параллельных запросов

Разработанный Microsoft встроенный язык запросов (Language Integrated Query, LINQ) предлагает удобный синтаксис запросов к данным. С его помощью элементы легко фильтровать и сортировать, возвращать спроецированные наборы элементов и делать многое другое. При работе с объектами все элементы в наборе данных последовательно обрабатываются одним потоком — это называется *последовательным запросом* (sequential query). Повысить производительность можно при помощи языка параллельных запросов (Parallel LINQ), позволяющего последовательный запрос превратить в *параллельный* (parallel query). Последний внутренне задействует задания (поставленные в очередь планировщиком, используемым по умолчанию), распределяя элементы коллекции среди нескольких процессоров для обработки. Как и в случае с методами класса `Parallel`, максимальный выигрыш получается, когда имеется множество элементов для обработки или когда обработка каждого элемента представляет собой длительную вычислительную операцию.

Вся функциональность языка параллельных запросов реализована в статическом классе `System.Linq.ParallelEnumerable` (он определен в библиотеке `System.Core.dll`). То есть в код следует импортировать пространство имен `System.Linq`. Данный класс, в частности, обладает параллельными версиями

стандартных LINQ-операторов, таких как Where, Select, SelectMany, GroupBy, Join, OrderBy, Skip, Take и т. п. Все эти методы являются методами расширения типа System.Linq.ParallelQuery<T>. Для вызова их параллельных версий следует преобразовать последовательный запрос (основанный на интерфейсе IEnumerable или IEnumerable<T>) в параллельный (основанный на классе ParallelQuery или ParallelQuery<T>), воспользовавшись методом расширения AsParallel класса ParallelEnumerable, который выглядит следующим образом¹:

```
public static ParallelQuery<TSource>AsParallel<TSource>(
    this IEnumerable<TSource> source)
public static ParallelQueryAsParallel(this IEnumerable source)
```

Вот пример преобразования последовательного запроса в параллельный. Он возвращает все устаревшие методы, определенные в сборке:

```
private static void ObsoleteMethods(Assembly assembly) {
    var query =
        from type in assembly.GetExportedTypes().AsParallel()

        from method in type.GetMethods(BindingFlags.Public |
            BindingFlags.Instance | BindingFlags.Static)

        let obsoleteAttrType = typeof(ObsoleteAttribute)

        where Attribute.IsDefined(method, obsoleteAttrType)

        orderby type.FullName

        let obsoleteAttrObj = (ObsoleteAttribute)
            Attribute.GetCustomAttribute(method, obsoleteAttrType)

        select String.Format("Type={0}\nMethod={1}\nMessage={2}\n",
            type.FullName, method.ToString(), obsoleteAttrObj.Message);

    // Вывод результатов
    foreach (var result in query) Console.WriteLine(result);
}
```

Хотя подобный подход и не распространен, существует возможность в ходе операций переключиться с параллельного режима на последовательный. Это делается при помощи метода AsSequential класса ParallelEnumerable:

```
public static IEnumerable<TSource> AsSequential<TSource>(
    this ParallelQuery<TSource> source)
```

Этот метод преобразует ParallelQuery<T> в интерфейс IEnumerable<T>, и все операции начинают выполняться всего одним потоком.

¹ Класс ParallelQuery<T> является производным от ParallelQuery.

Обычно результат LINQ-запроса вычисляется потоком, исполняющим инструкцию `foreach` (как было показано ранее). Это означает, что все результаты запроса просматриваются всего одним потоком. Параллельный режим обработки обеспечивает метод `ForAll` класса `ParallelEnumerable`:

```
static void ForAll<TSource>(  
    this ParallelQuery<TSource> source, Action<TSource> action)
```

Этот метод позволяет нескольким потокам одновременно обрабатывать результаты запросов. Мой приведенный ранее код с помощью этого метода можно переписать следующим образом:

```
// вывод результатов  
query.ForAll(Console.WriteLine);
```

Однако одновременный вызов метода `Console.WriteLine` несколькими потоками отрицательно сказывается на производительности, так как класс `Console` внутренне синхронизирует потоки, гарантируя, что к консоли в каждый момент времени имеет доступ только один поток. Именно это предотвращает смешение потоков, из-за которого может появиться непонятный результат. Используйте метод `ForAll` в случаях, когда требуется вычисление каждого из результатов.

Так как при параллельном LINQ-запросе элементы обрабатываются несколькими потоками одновременно, результаты возвращаются в произвольном порядке. Для сохранения очередности обработки элементов применяется метод `AsOrdered` класса `ParallelEnumerable`. С его помощью потоки разбивают элементы по группам, которые впоследствии сливаются друг с другом. Однако все это отрицательно сказывается на производительности. Вот операторы, предназначенные для выполнения неупорядоченных операций: `Distinct`, `Except`, `Intersect`, `Union`, `Join`, `GroupBy`, `GroupJoin` и `ToLookup`. После любого из этих операторов можно вызвать метод `AsOrdered`, чтобы упорядочить элементы.

А вот операторы для выполнения упорядоченных операций: `OrderBy`, `OrderByDescending`, `ThenBy` и `ThenByDescending`. Если вы хотите вернуться к неупорядоченным операциям, чтобы повысить производительность, после любого из этих операторов также можно вызвать метод `AsUnordered`.

Язык параллельных запросов предлагает также дополнительные методы класса `ParallelEnumerable`, позволяющие управлять обработкой запросов:

```
public static ParallelQuery<TSource> WithCancellation<TSource>(  
    this ParallelQuery<TSource> source, CancellationToken cancellationToken)
```

```
public static ParallelQuery<TSource> WithDegreeOfParallelism<TSource>(  
    this ParallelQuery<TSource> source, int degreeOfParallelism)
```

```
public static ParallelQuery<TSource> WithExecutionMode<TSource>(  
    this ParallelQuery<TSource> source, ParallelExecutionMode executionMode)
```

```
public static ParallelQuery<TSource> WithMergeOptions<TSource>(  
    this ParallelQuery<TSource> source, ParallelMergeOptions mergeOptions)
```

Очевидно, что методу `WithCancellation` можно передать объект `CancellationToken`, что дает возможность в любой момент остановить обработку запроса. Метод `WithDegreeOfParallelism` задает максимальное количество потоков, которые могут обрабатывать запрос; при этом, если количество реально необходимых потоков меньше указанного, новые потоки не создаются. Обычно этот метод не используется, и по умолчанию запрос исполняется одним потоком на одно ядро. Однако этот метод можно вызвать, указав число ядер, меньшее реально имеющегося, оставив часть ядер для решения других задач. Если запрос выполняет синхронную операцию ввода-вывода, можно указать число ядер, превышающее реально имеющееся, так как во время таких операций потоки блокируются. При таком подходе потоки расходуются впустую, зато вы быстрее получаете результат.

Это можно делать в клиентских приложениях, но я бы крайне не рекомендовал прибегать к синхронным операциям ввода-вывода в серверных приложениях.

В `Parallel LINQ` запрос анализируется и выбирается оптимальный способ его обработки. Иногда производительность может оказаться выше при последовательных запросах. Обычно это бывает при использовании следующих операций: `Concat`, `ElementAt(OrDefault)`, `First(OrDefault)`, `Last(OrDefault)`, `Skip(While)`, `Take(While)` или `Zip`. Кроме того, это верно для случаев использования перегруженных версий методов `Select(Many)` или `Where`, в которых селектору передается позиционный индекс или делегат, возвращающий логическое значение. При этом запрос можно принудительно обработать в параллельном режиме, передав методу `WithExecutionMode` один из флагов `ParallelExecutionMode`:

```
public enum ParallelExecutionMode {  
    Default = 0,           // Способ обработки запроса выбирается автоматически  
    ForceParallelism = 1 // Запрос обрабатывается в параллельном режиме  
}
```

Как уже упоминалось, в `Parallel LINQ` обработкой запросов занимается целая группа потоков, а значит, возникает необходимость соединения полученных результатов в один. Для управления буферизацией и слиянием элементов используется метод `WithMergeOptions`, которому передается один из флагов `ParallelMergeOptions`:

```
public enum ParallelMergeOptions {  
    Default = 0,           // Аналогично AutoBuffered (в будущем может измениться)  
    NotBuffered = 1,       // Результаты обрабатываются по мере готовности  
    AutoBuffered = 2,      // Поток буферизует некоторые результаты  
                          // перед обработкой  
    FullyBuffered = 3      // Поток буферизует все результаты перед обработкой  
}
```

Эти параметры позволяют выбрать желаемое соотношение скорости работы и потребления памяти. Флаг `NotBuffered` экономит память, но обработка элементов происходит медленнее. А вот флаг `FullyBuffered` увеличивает потребление

памяти, но результат вы получите быстрее. Компромиссом между этими вариантами является флаг `AutoBuffered`. Определить, какой именно вариант лучше всего подходит именно вам, проще всего экспериментальным путем. Можно также принять параметры, предлагаемые по умолчанию, что оптимально для большинства ситуаций. Дополнительную информацию о `Parallel LINQ` можно найти по следующим адресам:

- ❑ <http://blogs.msdn.com/pfxteam/archive/2009/05/28/9648672.aspx>;
- ❑ <http://blogs.msdn.com/pfxteam/archive/2009/06/13/9741072.aspx>.

Периодические вычислительные операции

В пространстве имен `System.Threading` определен класс `Timer`, который позволяет периодически вызывать методы из пула потоков. Создавая экземпляр этого класса, вы как бы информируете пул, что вам нужен метод, обратный вызов которого должен быть выполнен в заданное время. У класса `Timer` есть несколько очень похожих друг на друга конструкторов:

```
public sealed class Timer : MarshalByRefObject, IDisposable {  
    public Timer(TimerCallback callback, Object state,  
        Int32 dueTime, Int32 period);  
    public Timer(TimerCallback callback, Object state,  
        UInt32 dueTime, UInt32 period);  
    public Timer(TimerCallback callback, Object state,  
        Int64 dueTime, Int64 period);  
    public Timer(TimerCallback callback, Object state,  
        TimeSpan dueTime, TimeSpan period);  
}
```

Все эти конструкторы создают объект `Timer`. Параметр `callback` указывает имя метода, обратный вызов которого должен выполняться потоком из пула. Конечно, созданный метод обратного вызова должен соответствовать типу делегата `System.Threading.TimerCallback`, который определяется следующим образом:

```
delegate void TimerCallback(Object state);
```

Параметр `state` конструктора служит для передачи методу обратного вызова данных состояния или значения `null`, если эти данные отсутствуют. Параметр `dueTime` позволяет задать для CLR время ожидания (в миллисекундах) перед первым вызовом метода обратного вызова. Это время представляется 32-разрядным значением со знаком или без, 64-разрядным значением со знаком или значением `TimeSpan`. Для немедленной активизации метода обратного вызова следует присвоить параметру `dueTime` значение 0. Последний

параметр `period` указывает периодичность (в миллисекундах) последующих обращений к методу обратного вызова. Если ему передано значение `Timeout.Infinite(-1)`, поток из пула ограничится одним обращением к методу обратного вызова.

В пуле имеется всего один поток для всех объектов `Timer`. Именно он знает время активизации следующего таймера. В этот момент поток пробуждается и вызывает метод `QueueUserWorkItem` объекта `ThreadPool`, чтобы добавить в очередь пула потоков запись, активизирующую метод обратного вызова. Если выполнение этого метода занимает много времени, возможно повторное срабатывание таймера. В результате один метод будет выполняться несколькими потоками пула. Решить эту проблему можно при помощи таймера, параметру `period` которого присвоено значение `Timeout.Infinite`. Такой таймер срабатывает только один раз. Затем в рамках метода обратного вызова вызывается метод `Change` и указывается новое время задержки, а параметру `period` снова присваивается значение `Timeout.Infinite`. Вот как выглядят перегруженные версии метода `Change`:

```
public sealed class Timer : MarshalByRefObject, IDisposable {  
    public Boolean Change(Int32 dueTime, Int32 period);  
    public Boolean Change(UInt32 dueTime, UInt32 period);  
    public Boolean Change(Int64 dueTime, Int64 period);  
    public Boolean Change(TimeSpan dueTime, TimeSpan period);  
}
```

В классе `Timer` существует также метод `Dispose`, позволяющий вообще отключать таймер и при желании при помощи параметра `notifyObject` сообщать ядру о завершении всех ожидающих обратных вызовов. Вот как выглядят перегруженные версии метода `Dispose`:

```
public sealed class Timer : MarshalByRefObject, IDisposable {  
    public Boolean Dispose();  
    public Boolean Dispose(WaitHandle notifyObject);  
}
```

ВНИМАНИЕ

При утилизации объекта `Timer` сборщиком мусора поток пула останавливает таймер, чтобы он больше не срабатывал. Поэтому при работе с таймером следует проверять наличие переменной, поддерживающей его «на плаву», иначе обращения к методу обратного вызова прекратятся. Эта ситуация подробно обсуждалась в главе 21.

Следующий код демонстрирует поток из пула, вызывающий метод, который сначала выполняется немедленно, а затем через каждые две секунды.

```
internal static class TimerDemo {  
    private static Timer s_timer;
```

продолжение »

```

public static void Go() {
    Console.WriteLine("Main thread: starting a timer");
    using (s_timer = new Timer(ComputeBoundOp, 5, 0, Timeout.Infinite)) {
        Console.WriteLine("Main thread: Doing other work here...");
        Thread.Sleep(10000); // Имитация другой работы (10 секунд)
    } // Отмена таймера методом Dispose
}

// Сигнатура этого метода должна соответствовать
// сигнатуре делегата TimerCallback
private static void ComputeBoundOp(Object state) {
    // Этот метод выполняется потоком из пула
    Console.WriteLine("In ComputeBoundOp: state={0}", state);
    Thread.Sleep(1000); // Имитация другой работы (1 секунда)

    // Заставляем таймер снова вызвать метод через 2 секунды
    s_timer.Change(2000, Timeout.Infinite);

    // Когда метод возвращает управление, поток
    // возвращается в пул и ожидает следующего задания
}
}

```

Варианты таймеров

Библиотека FCL содержит различные таймеры, но большинство программистов даже не знают, чем они отличаются друг от друга.

- ❑ **Класс `Timer` из пространства имен `System.Threading`.** Этот класс рассматривался в предыдущем разделе. Он лучше других подходит для выполнения повторяющихся фоновых заданий в потоке пула.
- ❑ **Класс `Timer` из пространства имен `System.Windows.Forms`.** Создание экземпляра этого класса указывает Windows на необходимость связать таймер с вызывающим потоком (см. Win32-функцию `SetTimer`). При срабатывании таймера Windows добавляет в очередь сообщений потока сообщение таймера (`WM_TIMER`). Поток должен выполнить прокачку сообщений (`message pump`), чтобы извлечь эти сообщения и передать их нужному методу обратного вызова. Обратите внимание, что вся работа осуществляется одним потоком — устанавливает таймер тот же поток, который исполняет метод обратного вызова. Это предотвращает параллельный вызов метода таймера несколькими потоками.
- ❑ **Класс `DispatcherTimer` из пространства имен `System.Windows.Threading`.** Этот класс является эквивалентом класса `Timer` из пространства имен `System.Windows.Forms` для приложений Silverlight и WPF.

- ❑ **Класс `Timer` из пространства имен `System.Timers`.** Этот класс является, по сути, оболочкой для класса `Timer` из пространства имен `System.Threading`. Он заставляет CLR по срабатыванию таймера ставить события в очередь пула потоков. Поскольку класс `System.Timers.Timer` является производным от класса `Component` из пространства имен `System.ComponentModel`, таймеры можно размещать в конструкторе форм приложения `Microsoft Visual Studio`. Этот класс появился в FCL в те времена, когда у `Microsoft` еще отсутствовала четкая концепция потоков и таймеров. Вообще говоря, его стоило бы удалить, оставив его функции классу `System.Threading.Timer`. Я никогда не работаю с классом `System.Timers.Timer` и не советую этого вам. Исключением является ситуация, когда таймер нужно поместить в конструктора форм.

Как пул управляет потоками

В этом разделе я хотел бы остановиться на том, каким образом пул управляет рабочими потоками и потоками ввода-вывода. Глубоко погружаться в детали мы не будем, так как внутренняя реализация этого процесса менялась при переходе от одной версии CLR к другой и явно будет изменена в будущем. Поэтому представим пул потоков в виде черного ящика. Этот ящик является не лучшим вариантом для одиночных приложений, так как в его основе лежит технология распределения потоков, ориентированная на работу с большим количеством приложений. Для некоторых приложений она работает лучше, чем для других. Впрочем, на сегодняшний день она прекрасно справляется со своими задачами, и я рекомендую отнестись к ней с доверием. Вряд ли вы сможете самостоятельно написать пул потоков, который будет функционировать лучше, чем поставляемый в составе CLR. Так как с течением времени внутренняя система управления потоками у пула меняется, многие приложения начинают работать лучше.

Ограничение количества потоков в пуле

CLR позволяет указать максимально возможное количество потоков, создаваемых пулом. Однако возникает ощущение, что задавать верхний предел для пула не стоит, потому что это может привести к зависанию или мертвому запирианию (клинчу). Представьте очередь из 1000 рабочих элементов, заблокированную сигнальным событием элемента под номером 1001. Если верхний предел для количества потоков равен 1000, этот новый поток исполнен не будет, а значит, вся тысяча потоков навсегда окажется заблокированной. Конечному пользователю останется только завершить работу приложения, потеряв несохраненные данные. Разработчики обычно не накладывают искусственных ограничений на доступные для приложения ресурсы. Скажем, запуская приложение, никто не

оговаривает объем используемой им памяти или ширину полосы пропускания. Хотя при этом считается нормальным по каким-то причинам ограничивать максимальное количество потоков в пуле.

Из-за проблем потребителей, касающихся зависания и клинча, разработчики CLR постоянно увеличивают заданное по умолчанию максимально возможное количество потоков в пуле. В настоящее время предел составляет 1000 потоков, что для 32-разрядного процесса, имеющего не менее 2 Гбайт адресного пространства, может рассматриваться как отсутствие ограничений. После загрузки библиотек Win32 и библиотек CLR, а также выделения собственной и управляемой кучи, остается примерно 1,5 Гбайт адресного пространства. Так как каждый поток требует для стека в пользовательском режиме и блока окружения (TEB) более 1 Мбайт памяти, в 32-разрядном процессе допустимо максимум 1360 потоков. Попытки создать большее количество потоков приведут к исключению `OutOfMemoryException`. 64-разрядный процесс предлагает 8 Тбайт адресного пространства, так что теоретически вы можете создавать сотни тысяч потоков. Но это будет пустая трата ресурсов, особенно с учетом того факта, что идеальное количество потоков совпадает с количеством процессоров. По идее разработчикам CLR следует убрать ограничения, но в настоящий момент это невозможно, так как в результате прекратят свою работу приложения, разработанные в предположении об ограниченном количестве потоков в пуле.

Класс `System.Threading.ThreadPool` предлагает несколько статических методов для управления количеством потоков в пуле: `GetMaxThreads`, `SetMaxThreads`, `GetMinThreads`, `SetMinThreads` и `GetAvailableThreads`. Впрочем, я не рекомендую ими пользоваться. Попытки менять заданные по умолчанию ограничения обычно ухудшают работу приложений. Если вы считаете, что вашему приложению требуются сотни или даже тысячи потоков, скорее всего, что-то не так с архитектурой приложения или механизмом использования потоков. О том, как правильно применять потоки, мы поговорим в главе 27.

Управление рабочими потоками

На рис. 26.1 показаны различные структуры данных, делающие рабочие потоки частью пула. Метод `ThreadPool.QueueUserWorkItem` и класс `Timer` всегда помещают рабочие элементы в глобальную очередь. Рабочие потоки берут элементы для обработки из очереди по алгоритму «первым пришел — первым ушел». А так как при наличии нескольких потоков элементы из глобальной очереди могут удаляться одновременно, все рабочие потоки конкурируют за право на записание в рамках синхронизации потоков, которое гарантирует, что никакие два или более потока не смогут одновременно обрабатывать один и тот же элемент. В некоторых приложениях это право на записание становится узким местом, до некоторой степени ограничивая масштабируемость и производительность.

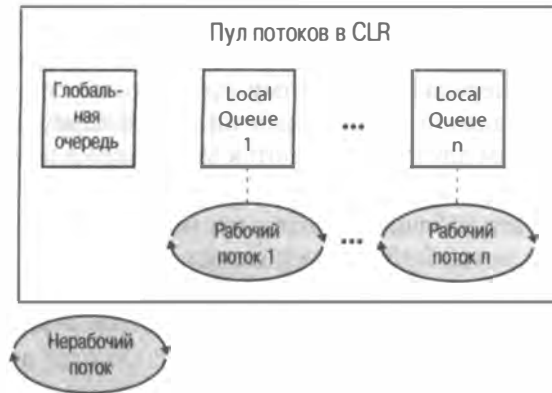


Рис. 26.1. Пул потоков в CLR

Рассмотрим процесс планирования заданий с помощью заданного по умолчанию планировщика (его можно получить через статическое свойство `Default` класса `TaskScheduler`)¹. При планировании задания для нерабочего потока объект `Task` добавляется в глобальную очередь. При этом каждый рабочий поток обладает собственной локальной очередью, в которую и добавляются планируемые задания.

Рабочий поток, готовый к обработке элементов, сначала проверяет наличие объектов `Task` в локальной очереди. Обнаружив такой объект, он изымает его из очереди и обрабатывает. Изъятие производится по алгоритму «последним пришел — первым ушел». Так как доступ к началу локальной очереди имеет только рабочий поток, запирание в рамках синхронизации потоков больше не требуется, а добавление заданий в очередь и изъятие их из нее происходят очень быстро. Побочным эффектом такого поведения является то, что выполнение заданий идет с конца очереди.

ВНИМАНИЕ

Пул потоков не гарантирует определенного порядка обработки элементов из очереди, особенно с учетом того факта, что наличие нескольких потоков делает возможной одновременную обработку нескольких элементов. Поэтому крайне желательно, чтобы для приложения порядок обслуживания элементов очереди не был принципиален.

Обнаружив пустую локальную очередь, рабочий поток пытается взять задание из локальной очереди другого рабочего потока. Задания, опять же, берутся с конца очереди, а значит, требуется запирание в рамках синхронизации потоков, что несколько снижает производительность. Остается надеяться на то,

¹ Поведение других объектов, производных от класса `TaskScheduler`, может отличаться от описываемого в данном разделе.

что записание будет случаться относительно редко. Если пустыми оказываются все локальные очереди, рабочий поток извлекает (прибегая к записанию) элемент из глобальной очереди по алгоритму «первым пришел — первым ушел». В случае пустой глобальной очереди рабочий поток переходит в режим ожидания. Если этот режим длится долго, поток просыпается и самоуничтожается, освобождая занятые ресурсы (ядро, стеки, ТEB).

Пул быстро создает рабочие потоки, а их количество определяется значением, переданным в метод `SetMinThreads` класса `ThreadPool`. Если вы не пользовались этим методом (а им и не стоит пользоваться), количество потоков по умолчанию совпадает с количеством процессоров, которые может задействовать процесс. Оно задается маской схожести процесса. Обычно допускается задействовать все процессоры¹, и пул создает рабочие потоки, количество которых быстро достигает числа процессоров. Затем пул начинает отслеживать частоту завершения рабочих элементов, и для тех из них, выполнение которых занимает много времени (с недокументированным значением), создает дополнительные потоки. При увеличении темпа завершения элементов рабочие потоки уничтожаются.

Строки кэша и ложное разделение

Для повышения производительности повторяющихся доступов к памяти современные процессоры оснащены встроенным кэшем. Доступ к кэшу — крайне быстрая процедура, особенно если вспомнить, какой была скорость доступа процессора к памяти материнской платы. При первом обращении потока к памяти процессор извлекает нужное значение и помещает его во встроенный кэш. Для еще большего повышения производительности вся память логически делится на *строки кэша* (*cache line*). Строка кэша для каждого из процессоров моего компьютера состоит из 64 байт, то есть процессор извлекает из оперативной памяти и сохраняет блоки именно такого размера². Если приложению нужно прочесть значение `Int32`, извлекаются содержащие его 64 байт. Извлечение большего количества байтов, чем требуется, обычно повышает производительность, так как большинство приложений для последующего доступа выбирает данные, расположенные рядом с теми, доступ к которым осуществляется в текущий момент. Заранее помещая их в кэш процессора, мы избегаем доступа к оперативной памяти.

Однако при доступе к байтам в одной строке кэша двух разных процессоров не обойтись без взаимодействия между ядрами и эффективной передачи строки

¹ В CLR версии 4 для 64-разрядного процесса чаще всего используются 64 процессора, а для 32-разрядного — 32.

² Задать количество байтов в строке кэша можно при помощи Win32-функции `GetProcessorInformation`. Моя библиотека `Power Threading` содержит управляемую оболочку этой функции, облегчая ее вызов из управляемого кода.

кэша из одного ядра в другое. Только в этом случае можно избежать одно-временного изменения соседних байтов разными ядрами. В противном случае производительность вычислительной операции может значительно снизиться. Рассмотрим это на примере:

```
internal static class FalseSharing {
    private class Data {
        // Два соседних поля, скорее всего, расположены в одной строке кэша
        public Int32 field1;
        public Int32 field2;
    }

    private const Int32 iterations = 100000000; // 100 миллионов
    private static Int32 s_operations = 2;
    private static Int64 s_startTime;

    public static void Main() {
        // Выделяем объект и записываем начальное время
        Data data = new Data();
        s_startTime = Stopwatch.GetTimestamp();

        // Два потока имеют доступ к своим полям внутри структуры
        ThreadPool.QueueUserWorkItem(o => AccessData(data, 0));
        ThreadPool.QueueUserWorkItem(o => AccessData(data, 1));

        // Для целей тестирования заблокируем поток Main
        Console.ReadLine();
    }

    private static void AccessData(Data data, Int32 field) {
        // Каждый поток имеет доступ к своим полям в объекте Data
        for (Int32 x = 0; x < iterations; x++)
            if (field == 0) data.field1++; else data.field2++;

        // Последний заверченный поток показывает время работы
        if (Interlocked.Decrement(ref s_operations) == 0)
            Console.WriteLine(
                "Access time: {0:N0}", Stopwatch.GetTimestamp() - s_startTime);
    }
}
```

В этом коде объект `Data` обладает двумя полями, которые, скорее всего, находятся в одной строке кэша. Два потока пула исполняют метод `AccessData`. Один поток добавляет к полю `field1` единицу 100 000 000 раз. Второй поток делает то же самое с полем `field2`. Каждый поток, завершив работу, начинает уменьшать на единицу значение в поле `s_operations`. Поток, получивший в итоге значение 0, показывает, сколько времени заняло выполнение работы обоими потоками. На своем компьютере я получил результат 15 856 074 мс.

Теперь внесем изменения в класс Data:

```
[StructLayout(LayoutKind.Explicit)]
private class Data {
    // Два поля больше не принадлежат одной строке кэша
    [FieldOffset(0)] public Int32 field1;
    [FieldOffset(64)] public Int32 field2;
}
```

То есть я поместил поля в разные строки кэша (64 байта). После запуска приложения я получил результат в 3 415 703 мс. То есть первая версия программы работала в четыре раза медленней из-за того, что процессору приходилось передавать байты из одного потока в другой и обратно! С точки зрения программы два потока манипулировали разными данными; а вот с точки зрения строки кэша процессора это были одни и те же данные. Такая ситуация называется *ложным разделением* (false sharing). В процессорах с неравномерным доступом к памяти (NUMA) возможно и более значительное снижение производительности. Во второй версии поля располагались в разных строках кэша, что позволило процессору работать с ними независимо.

Думаю, вы уже убедились, насколько серьезно могут повлиять на производительность строки кэша и ложное разделение в случае одновременного доступа разных потоков к близко расположенным данным. Распознав такую ситуацию, вы найдете способ ее обойти (например, воспользовавшись атрибутом FieldOffset).

К примеру, следует помнить, что информация о размере массива располагается в начале соответствующей области памяти, рядом с информацией о нескольких первых элементах. Поэтому при доступе к любому элементу массива CLR проверяет его индекс, обращаясь к информации о размере. А значит, чтобы избежать дополнительного ложного разделения, следует запретить одному потоку запись в первые элементы массива, если в это время происходит доступ других потоков к прочим элементам.

Глава 27. Асинхронные операции ввода-вывода

В предыдущей главе мы говорили об асинхронном выполнении вычислительных операций, когда пул потоков распределяет задания среди многочисленных ядер, обеспечивая параллельное исполнение потоков, что позволяет повысить производительность за счет более эффективного расходования ресурсов системы. В этой главе речь идет об асинхронном выполнении операций ввода-вывода, когда аппаратное обеспечение решает свои задачи вообще без участия потоков и процессора. Это, несомненно, оказывает влияние на эффективность расходования системных ресурсов, так как в этом случае эти ресурсы вообще не потребляются. Впрочем, пул потоков исполнения все равно играет важную роль, так как именно там обрабатываются результаты разнообразных операций ввода-вывода.

Операции ввода-вывода в Windows

Для начала рассмотрим, как в Microsoft Windows выполняются синхронные операции ввода-вывода. На рис. 27.1 показан компьютер с подсоединенным

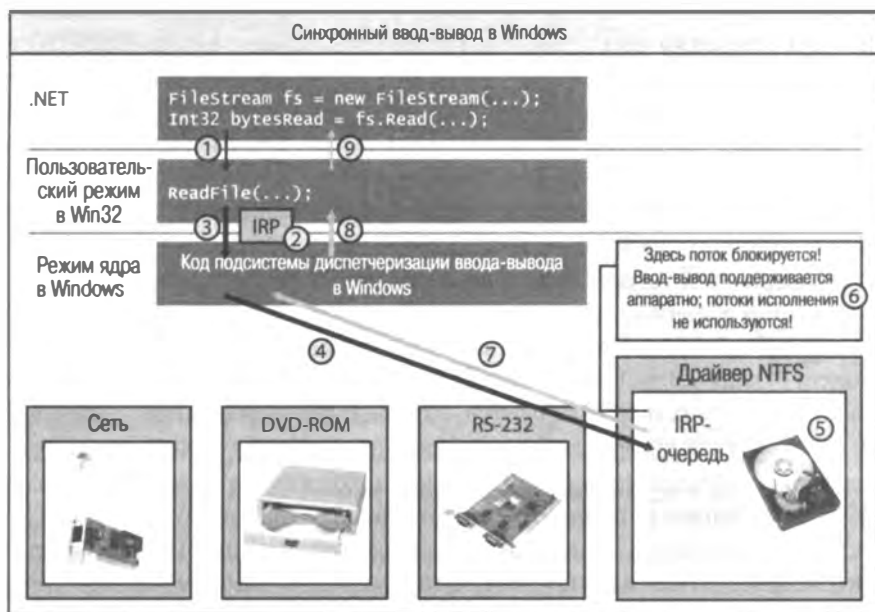


Рис. 27.1. Синхронные операции ввода-вывода в Windows

к нему периферийным оборудованием. Каждое из устройств снабжено собственной платой с микропроцессором специального назначения. К примеру, плата жесткого диска умеет вращать диск, устанавливать головку на нужную дорожку, читать данные с диска и записывать их на него, перемещать данные в память компьютера и обратно.

Открытие дисковых файлов в программах происходит путем создания объекта `FileStream`. Затем методом `Read` читаются данные из файла. Вызов метода `Read` объекта `FileStream` сопровождается переходом потока от управляемого кода в машинный код/код пользовательского режима, при этом вызывается Win32-функция `ReadFile` (1). Она выделяет память для небольшой структуры, называемой пакетом запросов ввода-вывода (`I/O Request Packet, IRP`) (2). Эта структура инициализирована дескриптором файла, смещением внутри файла, с которого начнется чтение байтов, адресом массива `Byte[]`, выделенного для считываемых байтов, количеством байтов, предназначенных для передачи и т. п.

Функция `ReadFile` обращается к ядру Windows, переводя поток из кода пользовательского режима в код в режиме ядра и передавая в ядро `IRP`-структуру (3). Из дескриптора ядро узнает, какое устройство предназначено для конкретной операции ввода-вывода, после чего пакет запросов ставится в `IRP`-очередь нужного драйвера устройства (4). Каждый драйвер устройства управляет собственной очередью запросов ввода-вывода от всех запущенных на машине процессов. При появлении `IRP`-пакетов драйвер устройства передает содержащуюся в них информацию соответствующему устройству, которое, собственно, и выполняет операцию ввода-вывода (5).

Но важно помнить про следующее обстоятельство. В процессе выполнения устройством операции ввода-вывода поток исполнения, передавший запрос, не имеет никаких заданий, поэтому Windows переводит его в спящее состояние, чтобы не расходовать процессорное время впустую (6). Однако при этом поток продолжает занимать место в памяти своим стеком пользовательского режима, стеком режима ядра, блоком переменных окружения потока (`Thread Environment Block, TEB`) и другими структурами данных, к которым в этот момент нет никакого доступа.

После завершения устройством операции ввода-вывода Windows пробуждает поток, ставит его в очередь процессора и позволяет ему вернуться из режима ядра сначала в пользовательский режим, а затем и в управляемый код (7, 8 и 9). Метод `Read` объекта `FileStream` при этом возвращает значение типа `Int32`, содержащее количество прочитанных из файла байтов. Это дает вам информацию о количестве байтов, оказавшихся в массиве `Byte[]`, ранее переданном методу `Read`.

Представим реализацию веб-приложения, в которой на каждый пришедший на ваш сервер клиентский запрос следует сделать запрос к базе данных. При клиентском запросе поток из пула потоков обращается к коду. При попытке синхронного запроса к базе данных этот поток окажется заблокированным на неопределенное время, необходимое для получения ответа из базы. Если в это время придет еще один клиентский запрос, пул создаст еще один поток, который снова

окажется заблокированным. В итоге можно оказаться с целым набором заблокированных потоков, ожидающих ответа из базы данных. То есть веб-сервер выделяет массу ресурсов (потоков и памяти для них), которые почти не используются!

Хуже всего то, что при получении результатов запросов из базы данных блокировка с потоков будет снята одновременно и все они начнут исполняться. В ситуации, когда количество потоков значительно превосходит количество ядер процессора, операционная система прибегнет к частым переключениям контекста, что значительно снизит производительность. Так что это не тот путь, который позволил бы реализовать масштабируемое приложение.

Теперь рассмотрим процедуру выполнения асинхронных операций ввода-вывода в Windows (рис. 27.2). Здесь отсутствуют все внешние устройства, кроме жесткого диска, кроме того, я добавил пул потоков среды CLR и слегка отредактировал код. Открытие файла по-прежнему выполняется путем создания объекта `FileStream`, но теперь ему передается флаг `FileOptions.Asynchronous`, который указывает Windows, что операции чтения из файла и записи в файл следует выполнять асинхронно.

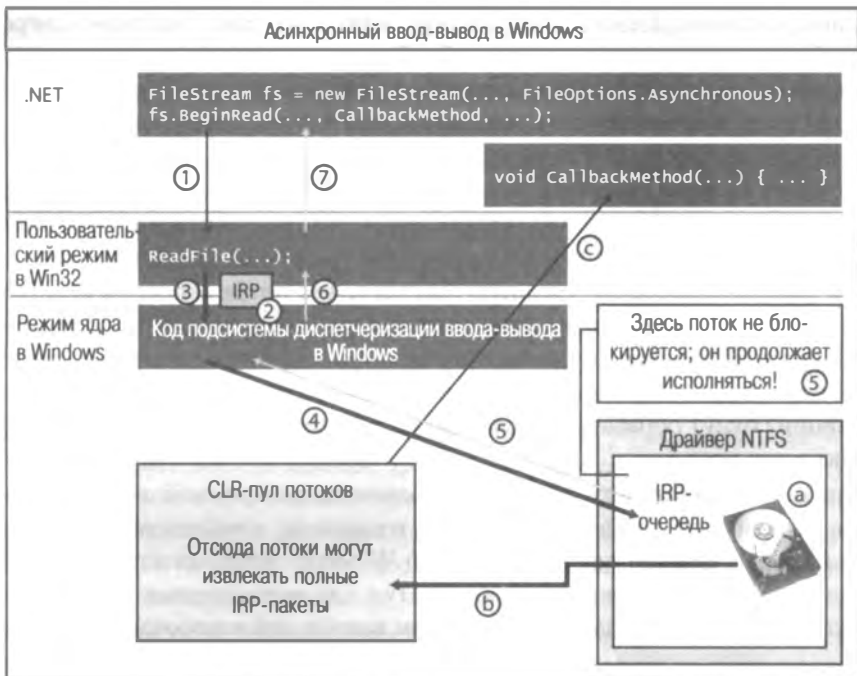


Рис. 27.2. Асинхронные операции ввода-вывода в Windows

Чтение данных из файла теперь выполняется методом `BeginRead`, а не методом `Read`. Впрочем, этот метод тоже вызывает Win32-функцию `ReadFile` (1), которая выделяет место под IRP-пакет, инициализирует его, как и в предыдущем сценарии (2), и передает в ядро Windows (3). Windows добавляет IRP-пакет

в IRP-очередь драйвера жесткого диска (4), но на этот раз поток не блокируется, а немедленно возвращает управление после вызовов метода `BeginRead` (5, 6 и 7). Теперь это может произойти еще до обработки IRP-пакета, поэтому у вас не будет кода, который после вызова метода `BeginRead` попытается получить доступ к байтам в переданном методу массиве `Byte[]`.

Может возникнуть вопрос, когда и каким образом обрабатываются считываемые данные? Методу `BeginRead` в качестве аргумента передается имя метода обратного вызова (в моем примере — `CallbackMethod`). Ссылающийся на метод обратного вызова делегат внутри IRP-пакета фактически передается драйверу устройства. Закончив обработку IRP-пакета (а), устройство помещает делегат в очередь CLR-пула потоков (б). В дальнейшем какой-то из потоков пула берет готовый IRP-пакет и активизирует метод обратного вызова (с)¹. В результате вы узнаете о завершении операции и внутри метода получите безопасный доступ к данным массива `Byte[]`.

Теперь, разобравшись с основами, посмотрим на открывающиеся перед нами перспективы. Предположим, в ответ на клиентский запрос сервер асинхронно запрашивает базу данных. При этом наш поток не блокируется, а возвращается в пул, получая возможность заняться обработкой других клиентских запросов. Таким образом, получается, что для обработки *всех* входящих запросов достаточно всего *одного* потока. Полученный от базы данных ответ также окажется в очереди пула потоков, то есть наш поток сможет тут же его обработать и отправить данные клиенту. Таким образом, единственный поток обрабатывает не только клиентские запросы, но и все ответы базы данных. В итоге сервер практически не потребляет системных ресурсов, но работает с максимальной возможной скоростью, так как переключения контекста не происходит!

Если элементы появляются в пуле быстрее, чем поток может их обработать, пул может создать дополнительные потоки. Пул быстро создаст по одному потоку на каждый процессор. Соответственно, на машине с четырьмя процессорами четыре клиентских запроса к базе данных и ответа базы данных (в любой комбинации) будут обрабатываться в четырех потоках без какого бы то ни было переключения контекста².

Однако при блокировке потока (выполнении синхронной операции ввода-вывода, вызове метода `Thread.Sleep` или ожидании, связанном с записью в поток в рамках синхронизации потоков) Windows уведомляет пул о том, что один из его потоков прекратил работу. Пул для восполнения недостаточной загрузки процессора создает новый поток взамен заблокированного. К сожа-

¹ Готовые IRP-пакеты извлекаются из пула по алгоритму «первым пришел — первым обслужен».

² Предполагается, что другие потоки в это время отсутствуют. Большую часть времени действительно так, ведь большинство компьютеров не задействует процессор на 100 %. Однако даже при полной загрузке процессора все будет работать описанным образом, если исполняемые потоки имеют низкие приоритеты. Наличие других потоков приводит к переключениям контекста. Это плохо с точки зрения производительности, но хорошо с точки зрения надежности. Напоминаю, что Windows выделяет на каждый процесс, по крайней мере, один поток и переключает контекст, гарантируя, что даже блокировка одного потока не остановит работу приложения.

лению, такой выход из положения далек от идеального, потому что создание нового потока является довольно дорогостоящей операцией с точки зрения затрат времени и памяти.

Кроме того, позднее поток может быть разблокирован, и в итоге процессор окажется перегруженным, что приведет к переключению контекста и снижению производительности. Впрочем, эта проблема решается средствами пула. Завершившим свою работу потокам, которые вернулись в пул, не дают обрабатывать новые элементы, пока загрузка процессора не достигнет определенного уровня. Таким способом уменьшается количество переключений контекста и повышается производительность. Если впоследствии пул обнаружит, что потоков больше, чем необходимо, он просто позволит лишним потокам самоуничтожиться, освободив ресурсы.

Для реализации описанного поведения CLR-пул потоков использует такой ресурс Windows, как *порт завершения ввода-вывода* (I/O Completion Port). Он создается при инициализации CLR. Затем с этим портом можно связать подключаемые устройства, чтобы в результате их драйверы «знали», куда поставить в очередь IRP-пакет. Подробнее этот механизм описан в моей книге «Windows via C/C++» (Microsoft Press, 2007).

Асинхронный ввод-вывод кроме минимального использования ресурсов и уменьшения количества переключений контекста предоставляет и другие преимущества. Скажем, в начале сборки мусора CLR приостанавливает все потоки в процессе. Получается, чем меньше у нас потоков, тем быстрее произойдет сборка мусора. Кроме того, при сборке мусора CLR просматривает в поисках корней все стеки потоков. Соответственно, чем меньше у нас потоков, тем меньше стеков приходится просматривать и тем быстрее работает сборщик мусора. Плюс ко всему, если в процессе обработки потоки не были заблокированы, большую часть времени они будут проводить в пуле в режиме ожидания. А значит, в начале сборки мусора потоки окажутся наверху стека, и поиск корней не займет много времени.

При достижении отлаживаемым приложением точки останова Windows приостанавливает все его потоки. После возвращения к отладке следует возобновить все потоки, а значит, при наличии большого количества потоков пошаговая отладка будет выполняться крайне медленно. Асинхронный ввод-вывод позволяет обойтись всего несколькими потоками, повышая тем самым производительность отладки.

Выгоды этим не исчерпываются. Предположим, ваше приложение должно загрузить с различных сайтов 10 изображений. Загрузка каждого из них занимает 5 секунд. В синхронном режиме выполнения (загрузка одного изображения за другим) вам потребуется 50 секунд. Однако при помощи всего одного потока можно начать 10 асинхронных операций загрузки и получить все изображения всего за 5 секунд! То есть время выполнения нескольких синхронных операций ввода-вывода получается путем суммирования времени, которое занимает каждая отдельная операция, в то время как в случае набора

асинхронных операций ввода-вывода время их завершения определяется самой медленной из выполняемых операций.

Приложения с графическим интерфейсом благодаря асинхронным операциям получают интерфейс, всегда реагирующий на действия конечного пользователя. В приложениях Silverlight вообще все операции ввода-вывода выполняются только асинхронно. В версии библиотеки FCL для Silverlight просто отсутствуют методы выполнения синхронных операций. Это было сделано намеренно. Ведь приложения Silverlight запускаются в браузерах, например в Windows Internet Explorer, а потоки, обслуживающие синхронные операции, могут заблокировать ответ от веб-сервера. В результате браузер просто зависнет. Пользователь не сможет даже перейти на соседнюю вкладку. Именно поэтому в Windows Internet Explorer 8 для каждой вкладки создается свой процесс. И в итоге, если одна вкладка перестает отвечать, это никак не сказывается на работе остальных. К сожалению, такой подход требует большого количества ресурсов, хотя и позволяет получить быстро реагирующий пользовательский интерфейс¹.

Модель асинхронного программирования в CLR

Асинхронные операции являются ключом к созданию высокопроизводительных, масштабируемых приложений, выполняющих множество операций при помощи небольшого количества потоков. Вместе с пулом потоков они дают возможность эффективно задействовать все процессоры в системе. Осознавая этот огромный потенциал, разработчики CLR разработали эталон программирования, призванный сделать его доступным для всех программистов. Этот эталон был назван *моделью асинхронного программирования* (Asynchronous Programming Model, APM).

Лично мне эта модель очень симпатична, потому что ее довольно легко освоить, она проста в использовании и поддерживается многими типами библиотеки FCL. Вот несколько примеров.

- ❑ Все производные от `System.IO.Stream` классы, которые взаимодействуют с аппаратными устройствами (в том числе `FileStream` и `NetworkStream`), поддерживают методы `BeginRead` и `BeginWrite`. Эти методы поддерживаются и классами, производными от `Stream`, которые не взаимодействуют с аппаратными устройствами (в том числе классами `BufferedStream`, `MemoryStream` и `CryptoStream`). Однако код в этих методах выполняет лишь вычислительные операции, а не операции ввода-вывода, поэтому для выполнения последних требуется поток.
- ❑ Класс `System.Net.Dns` поддерживает методы `BeginGetHostAddresses`, `BeginGetHostByName`, `BeginGetHostEntry` и `BeginResolve`.

¹ Собственная вкладка выделяется каждому процессу также из соображений безопасности.

- ❑ Класс `System.Net.Sockets.Socket` поддерживает методы `BeginAccept`, `BeginConnect`, `BeginDisconnect`, `BeginReceive`, `BeginReceiveFrom`, `BeginReceiveMessageFrom`, `BeginSend`, `BeginSendFile` и `BeginSendTo`.
- ❑ Все классы, производные от `System.Net.WebRequest` (в том числе `FileWebRequest`, `FtpWebRequest` и `HttpWebRequest`), предлагают методы `BeginGetRequestStream` и `BeginGetResponse`.
- ❑ Класс `System.IO.Ports.SerialPort` обладает предназначенным только для чтения свойством `BaseStream`, которое возвращает объект `Stream`, обладающий, как вы знаете, методами `BeginRead` и `BeginWrite`.
- ❑ Класс `System.Data.SqlClient.SqlCommand` поддерживает методы `BeginExecuteNonQuery`, `BeginExecuteReader` и `BeginExecuteXmlReader`.

Кроме того, для всех типов делегатов определен метод `BeginInvoke`, который можно использовать в APM. Наконец, инструменты создания типов-представителей (проxy) для веб-сервисов (такие как `WSDL.exe` и `SvcUtil.exe`) также генерируют методы, пригодные для использования в рамках APM. Названия этих методов начинаются с `Begin` и у каждого из них есть парный метод, название которого начинается с `End`. Как видите, поддержка APM буквально «пронизывает» всю библиотеку FCL.

Для синхронного чтения байтов из объекта `FileStream` вызывается его метод `Read` с таким прототипом:

```
public Int32 Read(Byte[] array, Int32 offset, Int32 count)
```

Вызов функции, выполняющей синхронный ввод-вывод, приводит к непредсказуемому состоянию приложения, так как вы не знаете, когда метод возвратит управление и возвратит ли его вообще. Представьте, к примеру, что открываемый файл находится на сервере, но перед вызовом метода `Read` сервер перестал работать из-за отключения электроэнергии.

Так что для написания работоспособного, надежного и масштабируемого приложения следует пользоваться только методами, выполняющими асинхронный ввод-вывод. Например, методом `BeginRead` объекта `FileStream`:

```
IAAsyncResult BeginRead(Byte[] array, Int32 offset, Int32 numBytes,
    AsyncCallback userCallback, Object stateObject)
```

Обратите внимание, что первые три параметра методов `BeginRead` и `Read` идентичны. Более того, каждый метод `BeginXxx` имеет те же самые параметры, что и его парный синхронный метод. Однако, кроме того, в методах `BeginXxx` есть и пара дополнительных параметров: `userCallback` и `stateObject`. Первый принадлежит типу делегата `AsyncCallback`:

```
public delegate void AsyncCallback(IAAsyncResult ar);
```

Ему передается имя метода (или лямбда-выражение), который должен выполняться потоком пула после завершения асинхронного ввода-вывода. Второй дополнительный параметр метода `BeginXxx` — `stateObject` — ссылается на объ-

ект, который вы хотите передать методу обратного вызова. Доступ к `objectState` внутри метода обратного вызова осуществляется через предназначенное только для чтения свойство `AsyncState` интерфейса `IAAsyncResult`.

Все методы `BeginXxx` возвращают объект, реализующий интерфейс `System.IAsyncResult`. Любой из этих методов конструирует объект, который уникальным образом идентифицирует ваш запрос ввода-вывода, ставит его в очередь драйвера устройства и возвращает ссылку на объект `IAAsyncResult`. Этот объект можно представить как уведомление. Ссылку на него на самом деле можно проигнорировать, потому что она сохраняется внутри CLR. После завершения операции поток пула активизирует метод обратного вызова, которому автоматически передается ссылка на объект `IAAsyncResult`.

Внутри вашего метода вызывается соответствующий метод `EndXxx`, в который передается объект `IAAsyncResult`. Результат его работы аналогичен результату, который вы бы получили, вызвав синхронный метод. К примеру, метод `Read` объекта `FileStream` возвращает значение типа `Int32`, указывающее количество байтов, считанных из потока ввода-вывода. Метод `EndRead` объекта `FileStream` возвращает аналогичное значение:

```
Int32 EndRead(IAAsyncResult result); // Количество байтов, считанных
                                     // из потока ввода-вывода
```

Вот как выглядит код серверного класса для именованного канала `PipeServer`, реализованный при помощи APM:

```
internal sealed class PipeServer {
    // Каждый серверный объект выполняет в канале асинхронные операции
    private readonly NamedPipeServerStream m_pipe = new NamedPipeServerStream(
        "Echo", PipeDirection.InOut, -1, PipeTransmissionMode.Message,
        PipeOptions.Asynchronous | PipeOptions.WriteThrough);

    public PipeServer() {
        // Асинхронное принятие соединения с клиентом
        m_pipe.BeginWaitForConnection(ClientConnected, null);
    }

    private void ClientConnected(IAAsyncResult result) {
        // Клиент подсоединен, принимаем другого клиента
        new PipeServer(); // Accept another client

        // Принятие соединения с клиентом
        m_pipe.EndWaitForConnection(result);

        // Асинхронное чтение запроса со стороны клиента
        Byte[] data = new Byte[1000];
        m_pipe.BeginRead(data, 0, data.Length, GotRequest, data);
    }
}
```

```
private void GotRequest(IAsyncResult result) {
    // Обработка присланного клиентом запроса
    Int32 bytesRead = m_pipe.EndRead(result);
    Byte[] data = (Byte[])result.AsyncState;

    // Мой сервер просто меняет регистр символов.
    // но вы можете вставить сюда любую вычислительную операцию
    data = Encoding.UTF8.GetBytes(
        Encoding.UTF8.GetString(data, 0, bytesRead).ToUpper().ToCharArray());

    // Асинхронная отправка ответа клиенту
    m_pipe.BeginWrite(data, 0, data.Length, WriteDone, null);
}

private void WriteDone(IAsyncResult result) {
    // Ответ клиенту отправлен, закрываем соединение со своей стороны
    m_pipe.EndWrite(result);
    m_pipe.Close();
}
}
```

Экземпляр этого класса следует создать *до соединения* клиента с сервером, так как это соединение обеспечивается вызовом метода `BeginWaitForConnection`. После его установки поток пула вызывает метод `ClientConnect` и создает новый экземпляр класса `PipeServer`, предоставляя возможность соединения дополнительного клиента. Тем временем метод `ClientConnected` вызывает метод `BeginRead`, заставляя драйвер сетевого устройства дожидаться входящих данных от клиента и поместить их в указанный массив `Byte[]`.

При отправке клиентом данных поток из пула вызывает метод `GotRequest`. Этот метод получает доступ к массиву `Byte[]` (путем запроса свойства `AsyncState`), после чего обрабатывает данные. В моем примере преобразование массива `Byte[]` к типу `String` осуществляется при помощи кодировщика UTF-8, затем символы строки переводятся в верхний регистр, после чего строка преобразуется обратно в массив `Byte[]`. Вы же можете заменить этот код собственной версией вычислительной операции, чтобы сервер выполнял ту работу, которая реально требуется. Далее метод `GotRequest` вызывает метод `BeginWrite` и отправляет полученный результат обратно клиенту. Когда драйвер устройства завершает отправку данных, поток пула вызывает метод `WriteDone`, закрывающий канал и обрывающий соединение.

Обратите внимание, что все методы следуют единому эталону. Они заканчиваются вызовом метода `BeginXxx` (исключая последний метод, `WriteDone`) и начинаются вызовом метода `EndXxx` (исключая конструктор). Между методами `EndXxx` и `BeginXxx` выполняются только вычислительные операции; ввод и вывод находятся «на границе» методов, поэтому потоки исполнения никогда не блокируются. После исполнения каждого метода поток возвращается в пул, где может заняться обработкой выходящих клиентских запросов или ответов сети. Если

пул окажется слишком загруженным, будут автоматически созданы новые потоки — мой сервер сам подстраивается под рабочую нагрузку и число процессоров!

Я создал свое серверное приложение как консольное и инициализировал его следующим образом:

```
public static void Main() {
    // Запуск одного сервера на каждый процессор
    for (Int32 n = 0; n < Environment.ProcessorCount; n++)
        new PipeServer();

    Console.WriteLine("Press <Enter> to terminate this server application.");
    Console.ReadLine();
}
```

А теперь посмотрим на код клиентского класса для именованного канала, также реализованный при помощи APM. Обратите внимание на одинаковую структуру классов PipeClient и PipeServer.

```
internal sealed class PipeClient {
    // В канале каждый клиент выполняет асинхронную операцию
    private readonly NamedPipeClientStream m_pipe;

    public PipeClient(String serverName, String message) {
        m_pipe = new NamedPipeClientStream(serverName, "Echo",
            PipeDirection.InOut,
            PipeOptions.Asynchronous | PipeOptions.WriteThrough);
        m_pipe.Connect(); // Требуется вызвать Connect перед установкой ReadMode
        m_pipe.ReadMode = PipeTransmissionMode.Message;

        // Асинхронная отправка данных на сервер
        Byte[] output = Encoding.UTF8.GetBytes(message);
        m_pipe.BeginWrite(output, 0, output.Length, WriteDone, null);
    }

    private void WriteDone(IAsyncResult result) {
        // Данные отправлены на сервер
        m_pipe.EndWrite(result);

        // Асинхронное чтение ответа сервера
        Byte[] data = new Byte[1000];
        m_pipe.BeginRead(data, 0, data.Length, GotResponse, data);
    }

    private void GotResponse(IAsyncResult result) {
        // Вывод ответа сервера и закрытие соединения
        Int32 bytesRead = m_pipe.EndRead(result);

        Byte[] data = (Byte[])result.AsyncState;
        Console.WriteLine(
```

```
        "Server response: " + Encoding.UTF8.GetString(data, 0, bytesRead));  
    m_pipe.Close();  
}  
}
```

А при помощи этого кода мое консольное приложение отправляет на сервер сотню запросов:

```
public static void Main() {  
    // Делаем 100 клиентских запросов серверу  
    for (Int32 n = 0; n < 100; n++)  
        new PipeClient("localhost", "Request #" + n);  
  
    // Так как все запросы выполняются асинхронно, конструктор, скорее всего,  
    // вернет управление до их завершения. Нижняя строка не дает приложению  
    // завершить работу, пока не выведены все ответы  
    Console.ReadLine();  
}
```

Разработчики часто реализуют серверы из расчета один поток на один клиентский запрос. Однако 32-разрядный процесс может создать не более чем 1360 потоков, потом у него просто кончается виртуальное адресное пространство. Это означает, что сервер, использующий модель «один поток на одного клиента», не может одновременно обслужить больше 1360 клиентов. Но я отредактировал свою программу, сконструировав набор объектов PipeServer, и получил возможность создать в 32-разрядном процессе более 4 миллионов потоков до возникновения проблем с памятью. Как видите, асинхронная модель обеспечивает одновременную работу большого количества клиентов, требует меньше ресурсов, быстрее обрабатывает запросы (благодаря меньшему количеству переключений контекста), сокращает время сборки мусора, а заодно повышает производительность отладки! Казалось бы, чего еще желать?

Класс AsyncEnumerator

В принципе, осталось пожелать только, чтобы асинхронное программирование стало проще. Большинство разработчиков игнорируют модель АРМ из-за ее сложности. Вот с какими проблемами они сталкиваются:

- ❑ Код приходится разбивать на множество методов обратного вызова.
- ❑ Невозможно пользоваться аргументами и локальными переменными, так как этим переменным выделяется память в стеке потока и поэтому доступ к ним из другого потока или другого метода невозможен.
- ❑ В ситуации, когда конструкция начинается одним методом, а заканчивается другим, недопустимы многие конструкции языка C#, в частности try/catch/finally, using, for, do, while и foreach. К примеру, в моем классе Server не-

плохо было бы воспользоваться инструкцией `using` для открытия канала, а затем автоматически закрыть его в блоке `finally`. Однако это невозможно, потому что канал открывается в конструкторе класса `Server`, а закрывается в методе `WriteDone`.

- ❑ Сложно реализовать ряд других возможностей, к которым привыкли разработчики, в частности координировать группу одновременно выполняющихся операций, поддерживать отмену и тайм-ауты, обеспечивать продвижение потока обновления элементов графического интерфейса.

Именно по этим причинам многие разработчики отказываются от модели APM. Именно для них я создал класс `AsyncEnumerator`, позволяющий решить эти проблемы. Коротко говоря, мой класс `AsyncEnumerator` обеспечивает выполнение асинхронных операций посредством модели синхронного программирования и возможностям итератора языка C#. Класс входит в мою библиотеку `Power Threading` и предоставляется бесплатно. Существуют версии библиотеки для CLR настольных компьютеров, `Silverlight` и `.NET Compact Framework`. Новейшую версию и примеры кода можно найти по адресу <http://Wintellect.com/PowerThreading.aspx>. Эта страница содержит также ссылку на мою колонку `Concurrent Affairs` в сетевом журнале `MSDN Magazine`, в которой я объясняю, как работает класс `AsyncEnumerator`. Есть также ссылки на демонстрационные видеозаписи и группу новостей, в которой я предлагаю бесплатную техническую поддержку.

А пока рассмотрим возможности, которые предлагает класс `AsyncEnumerator`:

- ❑ Он реализует модель APM таким образом, что она легко интегрируется со всеми технологиями `.NET Framework`, такими как `ASP.NET` и `Windows Communication Foundation (WCF)`. Также он делает возможным сочетание асинхронных подпрограмм, то есть вызов одним объектом `AsyncEnumerator` другого.
- ❑ Он координирует одновременно выполняемые асинхронные операции. То есть вы можете запустить на выполнение произвольное количество таких операций, а класс `AsyncEnumerator` может уведомлять как о завершении каждой отдельной операции, так и о завершении их всех.
- ❑ Он поддерживает *группы сброса* (`discard groups`), позволяющие выполнить набор асинхронных операций и после их завершения заставить объект `AsyncEnumerator` отказаться от результатов. Представьте, что вы запросили температуру в Лондоне с трех различных сайтов, сделав это внутри одной группы сброса. Первый ответивший сервер дает приложению информацию, которую оно начинает обрабатывать. А так как остальные операции являются частью группы сброса, их результатов можно не дожидаться.
- ❑ В случаях когда результат вас не волнует, этот класс позволяет автоматически отменять наборы асинхронных операций, вызывая для каждой из них метод `EndXxx` (и проглатывая исключения). Такая возможность особенно полезна для приложений с графическим интерфейсом, так как пользователь

может отменить операцию, если он устал ждать ее результата. Кроме того, можно сделать так, чтобы отмена автоматически совершалась после указанного разработчиком промежутка времени. Это неоценимо для серверных приложений, которые должны за определенное время отправить клиенту ответ на запрос.

- ❑ Вам теперь не нужно заботиться о потоковой модели приложения (она обсуждается чуть позднее). К примеру, в приложениях с графическим интерфейсом класс AsyncEnumerator по умолчанию задействует код GUI-потока, давая возможность обновить элементы управления пользовательского интерфейса. Для приложений ASP.NET Web Form или XML Web Service класс AsyncEnumerator автоматически запускает код с региональными и идентификационными параметрами клиента.
- ❑ Класс поддерживает обработку ошибок. Если асинхронная операция завершается после того, как итератор вернул управление, класс AsyncEnumerator вбрасывает исключение, информирующее о невозможности вызова метода EndXxx и утечке ресурсов приложения.
- ❑ Класс поддерживает отладку. Обычно в памяти приложения находится несколько объектов AsyncEnumerator. Особенно много их у серверных приложений. Список этих объектов можно получить, вызвав в отладчике статический метод GetInProgressList класса AsyncEnumerator. Объекты, дольше всего ожидающие завершения операции, находятся наверху списка. Если вам кажется, что приложение зависло, достаточно посмотреть на верхнюю строчку, и вы поймете, какой фрагмент кода ожидает завершения операции. Кроме того, просмотр отдельного объекта AsyncEnumerator в отладчике дает информацию об определяемых пользователем тегах, идентифицирующих операцию, о временной отметке последней выполненной асинхронной операции, о завершенных и еще не завершенных операциях. Демонстрируется также исходный код файла и строка, в которой началась асинхронная операция.

Вот как выглядит серверный код канала, реализованный при помощи моего класса AsyncEnumerator¹:

```
private static IEnumerator<Int32> PipeServerAsyncEnumerator(
    AsyncEnumerator ae) {
    // В этом канале каждый сервер выполняет асинхронные операции
    using (var pipe = new NamedPipeServerStream(
        "Echo", PipeDirection.InOut, -1, PipeTransmissionMode.Message,
        PipeOptions.Asynchronous | PipeOptions.WriteThrough)) {

        // Асинхронное принятие клиентского соединения
        pipe.BeginWaitForConnection(ae.End(), null);
        yield return 1;
    }
}
```

продолжение ➤

¹ Показанный в этой книге пример содержит код клиентского канала, повторно реализованный для использования класса AsyncEnumerator. Данная версия кода соответствует показанной ранее структуре.

```
// Клиент подсоединен, примем еще одного клиента
var aeNewClient = new AsyncEnumerator();
aeNewClient.BeginExecute(PipeServerAsyncEnumerator(aeNewClient),
    aeNewClient.EndExecute());

// Принятие клиентского соединения
pipe.EndWaitForConnection(ae.DequeueAsyncResult());

// Асинхронное чтение клиентского запроса
Byte[] data = new Byte[1000];
pipe.BeginRead(data, 0, data.Length, ae.End(), null);
yield return 1;

// Обработка присланного клиентом запроса
Int32 bytesRead = pipe.EndRead(ae.DequeueAsyncResult());

// В моем примере сервер просто меняет регистр символов,
// но вы можете вставить сюда любую вычислительную операцию
data = Encoding.UTF8.GetBytes(
    Encoding.UTF8.GetString(data, 0, bytesRead).ToUpper().ToCharArray());

// Асинхронная отправка ответа клиенту
pipe.BeginWrite(data, 0, data.Length, ae.End(), null);
yield return 1;

// Ответ отправлен, закрываем соединение со стороны сервера
pipe.EndWrite(ae.DequeueAsyncResult());
} // Теперь закрытие осуществляется в блоке finally!
}
```

По поводу данной версии кода следует сделать несколько замечаний.

- ❑ Весь код помещен в один метод, а не «размазан» по всему классу среди множества методов. Так как класс отсутствует, отсутствуют и поля. Все переменные являются локальными.
- ❑ Там, где раньше приходилось разделять методы, теперь имеется инструкция `yield return 1`. Она позволяет потоку вернуться в исходную точку, что повышает продуктивность его работы.
- ❑ Каждому методу `BeginXxx` передается метод `ae.End()`; этот метод возвращает делегата, ссылающегося на метод внутри объекта `AsyncEnumerator`. Когда операция завершается, поток пула уведомляет объект `AsyncEnumerator`, который, в свою очередь, продолжает выполнение итератора, следующего за инструкцией `yield return 1`.
- ❑ В качестве последнего аргумента каждого метода `BeginXxx` всегда передается значение `null`. Следовательно, отпадает необходимость вызывать свойство `AsyncState` объекта `IAAsyncResult` и приводить возвращаемое им значение к нужному типу. Вместо этого используется локальная переменная.

- В каждый метод `EndXxx` передается результат вызова метода `ae.DequeueAsyncResult()`. Данный метод возвращает объект `IAAsyncResult`, который в момент завершения асинхронной операции передается потоком пула объекту `AsyncEnumerator`.
- И наконец, что не менее важно, я теперь могу использовать инструкцию `using` языка `C#` для управления временем жизни объекта `NamedPipeServerStream`. Это означает, что если какой-нибудь другой код не станет источником необработанного исключения, данный объект будет закрыт в блоке `finally`.

Я надеюсь, вы получили представление о возможностях класса `AsyncEnumerator` и о том, насколько он упрощает асинхронное программирование. Показанный в данном разделе код, к сожалению, демонстрирует не все его удивительные возможности, некоторые остались «за кадром». Вы можете подробнее познакомиться с данным классом, посетив сайт `Wintellect`, ссылка на который была предоставлена ранее.

Модель асинхронного программирования и исключения

Любой вызов метода `BeginXxx` может привести к исключению. Это обычно происходит, если асинхронная операция не была поставлена в очередь, а значит, поток пула не активизировал ни одного метода обратного вызова, который можно было бы передать методу `BeginXxx`.

Если при обработке драйвером устройства асинхронного запроса что-то пойдет не так, `Windows` нужно проинформировать об этом приложение. К примеру, представим, что закончилось время ожидания при отправке байтов по сети. Если данные не приходят вовремя, драйвер устройства сообщает вам, что асинхронная операция завершилась с ошибкой. Для этого он отправляет готовый `IRP`-пакет в `CLR`-пул потоков и помещает код ошибки в объект `IAAsyncResult`, представляющий асинхронную операцию. После этого поток пула передает в метод обратного вызова объект `IAAsyncResult`. После этого метод обратного вызова передаст объект `IAAsyncResult` нужному методу `EndXxx`, который, обнаружив код ошибки, преобразует его в объект, производный от класса `Exception`, тем самым вбрасывая исключение.

В результате все исключения работают одинаково независимо от синхронного или асинхронного программирования. Впрочем, обычно исключения, брошенные вызовом метода `BeginXxx`, нам не интересны, интерес вызывают исключения, появившиеся в результате вызова метода `EndXxx`. Вот пример обработки исключения в `APM`:

```
internal static class ApmExceptionHandling {  
    public static void Main() {  
        // Попытка доступа к несуществующему IP-адресу
```



```

    WebRequest webRequest = WebRequest.Create("http://0.0.0.0/");
    webRequest.BeginGetResponse(ProcessWebResponse, webRequest);
    Console.ReadLine();
}

private static void ProcessWebResponse(IAsyncResult result) {
    WebRequest webRequest = (WebRequest)result.AsyncState;

    WebResponse webResponse = null;
    try {
        webResponse = webRequest.EndGetResponse(result);
        Console.WriteLine("Content length: " + webResponse.ContentLength);
    }
    catch (WebException we) {
        Console.WriteLine(we.GetType() + ": " + we.Message);
    }
    finally {
        if (webResponse != null) webResponse.Close();
    }
}
}

```

После запуска программы я получил следующий результат:

System.Net.WebException: Unable to connect to the remote server

ПРИМЕЧАНИЕ

При выполнении множества асинхронных HTTP-запросов часто возникает неожиданная для большинства разработчиков проблема. Протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP), определенный в RFC 2616, устанавливает, что клиентское приложение не может иметь более двух одновременных соединений с одним сервером. FCL-классы фактически узаконивают это правило, и любой поток исполнения, пытающийся создать дополнительное соединение с сервером, блокируется вплоть до закрытия одного из двух уже установленных соединений. То есть следует либо разрабатывать приложение таким образом, чтобы количество исходящих соединений с одним сервером не превышало двух, либо увеличить максимальное количество одновременных соединений, присвоив нужное значение статическому свойству `DefaultConnectionLimit` класса `System.Net.ServicePointManager`.

Потоковые модели приложений

В .NET Framework поддерживаются разнообразные прикладные модели, каждая из которых может предложить собственную потоковую модель. Консольные приложения и Windows-службы (которые фактически тоже являются консольными приложениями, просто вы не видите консоль) не навязывают никакой потоковой модели; то есть поток может делать все, что он хочет и когда хочет.

Однако приложения с графическим пользовательским интерфейсом (GUI), в том числе Windows Forms, Windows Presentation Foundation (WPF) и Silverlight, предлагают такую модель, в которой обновлять окно можно только создавшему его потоку. GUI-потоки обычно порождают асинхронные операции, чтобы предотвратить блокировку и не допустить отсутствия реакции интерфейса на средства пользовательского ввода — мышь, клавиатуру, сенсорный экран. Однако завершаясь, асинхронная операция прекращает использовать поток пула, который уже не может обновлять пользовательский интерфейс, представляющий результаты. То есть в пуле потоков должен существовать GUI-поток, призванный каким-то образом обновлять пользовательский интерфейс.

Подобно консольным приложениям, веб-формы ASP.NET и веб-службы XML позволяют потокам вести себя как угодно. Начав обрабатывать клиентский запрос, поток пула может выбрать пользовательские региональные стандарты (System.Globalization.CultureInfo), позволив серверу осуществить принятые в рассматриваемом регионе форматы чисел, дат и времени¹. Также веб-сервер может определить идентификационные данные клиента (System.Security.Principal.IPrincipal), оставив ему доступ только на те ресурсы, на которые у него есть права. Порожденная одним потоком пула асинхронная операция заканчивается другим потоком, который обрабатывает ее результат. Хотя эта работа и выполняется по поручению клиентского запроса, региональные стандарты и идентификационные данные клиента не передаются от одного потока пула к другому. В идеале данное ограничение хотелось бы устранить, предоставив эту информацию всем потокам, действующим от имени одного и того же клиента.

К счастью, в FCL определен базовый класс System.Threading.SynchronizationContext, позволяющий решить все описанные проблемы. Объект, производный от этого класса, связывает прикладную модель с потоковой. В FCL имеется группа классов, производных от класса SynchronizationContext, но обычно напрямую они не используются; более того, многие из них даже не документированы. Вот как выглядит класс SynchronizationContext (в примере показаны только те члены, которые имеют отношение к данному обсуждению):

```
public class SynchronizationContext {  
    public static SynchronizationContext Current { get; }  
    public virtual void Post(                // Асинхронный вызов  
        SendOrPostCallback d, object state);  
    public virtual void Send(                // Асинхронный вызов  
        SendOrPostCallback d, object state);  
}
```

// Делегат SendOrPostCallback определен так:

```
public delegate void SendOrPostCallback(Object state);
```

¹ Дополнительная информация по данной теме находится по адресу <http://msdn.microsoft.com/ru-ru/library/bz9tc508.aspx>

Для приложений Windows Forms, WPF и Silverlight с GUI-потокom связывается объект, производный от класса `SynchronizationContext`. Вы можете получить ссылку на него, заставив GUI-поток запросить статическое свойство `Current` класса `SynchronizationContext`. Эту ссылку потом можно сохранить в какой-нибудь переменной общего доступа (например, в статическом поле вашего класса). В результате, как только потоку пула потребуется заставить GUI-поток обновить пользовательский интерфейс, он вызовет метод `Post` сохраненного объекта, передав ему метод, который следует вызвать средствами графического интерфейса (его сигнатура должна совпадать с сигнатурой делегата `SendOrPostCallback`), и передаваемый в этот метод аргумент.

Я рекомендую использовать метод `Post`, так как он ставит обратный вызов в очередь GUI-потока, позволяя потоку пула немедленно возратить управление. Метод `Send` тоже ставит обратный вызов в очередь GUI-потока, но блокирует поток пула до завершения обратного вызова GUI-потокom. В большинстве случаев это приводит к созданию в пуле нового потока и увеличению потребления ресурсов на фоне падения производительности. Поэтому я никогда не пользуюсь методом `Send`.

ПРИМЕЧАНИЕ

Вот как различные классы, производные от класса `SynchronizationContext`, заставляют GUI-поток вызвать метод `SendOrPostCallback`. Для приложений Windows Forms метод `Post` класса `System.Windows.Forms.WindowsFormsSynchronizationContext` вызывает метод `BeginInvoke` класса `System.Windows.Forms.Control`, а его метод `Send` вызывает метод `Invoke` класса `Control`. Для приложений WPF и Silverlight метод `Post` класса `System.Windows.Threading.DispatcherSynchronizationContext` вызывает метод `BeginInvoke` класса `System.Windows.Threading.Dispatcher`, а его метод `Send` вызывает метод `Invoke` класса `Dispatcher`.

Для приложений ASP.NET Web Form и веб-служб XML с потоком пула, который запустится при получении входящего клиентского запроса, будет связан объект, производный от класса `SynchronizationContext`. Он содержит информацию о региональных стандартах и идентификационные данные клиента. Для получения ссылки на него потоку пула достаточно сделать запрос к свойству `Current` класса `SynchronizationContext`. Полученную ссылку имеет смысл сохранить в поле объекта, обрабатывающего клиентский запрос. При обратном вызове метода этого класса другим потоком пула обрабатывающий поток вызовет метод `Post` сохраненного объекта, передав ему метод, который следует вызвать, воспользовавшись информацией о региональных стандартах и идентификационными данными клиента. Сигнатура этого метода должна совпадать с сигнатурой делегата `SendOrPostCallback delegate`. Он будет выполнен тем же самым потоком, который вызвал метод `Post`. Аналогично выполняются методы `Post` и `Send` типа, производного от `SynchronizationContext`, для приложений ASP.NET.

Для консольных приложений и Windows-служб не существует связанного с потоком объекта, производного от класса `SynchronizationContext`; запрос к статическому свойству `Current` класса `SynchronizationContext` возвращает значение `null`.

Поначалу во всем этом легко запутаться, поэтому я написал метод, призванный сильно упростить ситуацию:

```
private static AsyncCallback SyncContextCallback(AsyncCallback callback) {  
    // Фиксируем производный от SynchronizationContext  
    // объект вызывающего потока  
    SynchronizationContext sc = SynchronizationContext.Current;  
  
    // При отсутствии контекста синхронизации возвращаем переданное в метод  
    if (sc == null) return callback;  
  
    // Возвращаем делегат, который отправляет в фиксированный  
    // контекст синхронизации метод, передающий в исходный вызов  
    // AsyncCallback аргумент IAsyncResult  
    return asyncResult => sc.Post(result => callback(  
        (IAsyncResult)result), asyncResult);  
}
```

Этот код преобразует нормальный метод `AsyncCallback`, иницилируя его вызов через производный от класса `SynchronizationContext` потоковый объект. Это гарантирует применение корректной потоковой модели вне зависимости от используемой прикладной модели¹. Вот пример использования модели APM и метода `SyncContextCallback` приложением `Windows Forms` с гарантией корректной работы:

```
internal sealed class MyWindowsForm : Form {  
    public MyWindowsForm() {  
        Text = "Click in the window to start a Web request";  
        Width = 400; Height = 100;  
    }  
  
    protected override void OnMouseClick(MouseEventArgs e) {  
        // GUI-поток иницирует асинхронный веб-запрос  
        Text = "Web request initiated";  
        var webRequest = WebRequest.Create("http://Wintellect.com/");  
        webRequest.BeginGetResponse(  
            SyncContextCallback(ProcessWebResponse), webRequest);  
        base.OnMouseClick(e);  
    }  
}
```

продолжение ➤

¹ Можно спросить, почему в .NET Framework автоматически не реализовано то, что делает мой метод `SyncContextCallback`. После появления версии .NET Framework 1.0 стало ясно, насколько глобальной является проблема наличия у разных прикладных моделей собственных потоковых моделей. В результате в версии 2.0 появился класс `SynchronizationContext`. Однако этот инструмент нельзя сделать используемым по умолчанию, так как он повлияет на работу уже существующих приложений.

```
private void ProcessWebResponse(IAsyncResult result) {
    // Если мы попали сюда, это должен быть GUI-поток,
    // значит, обновляем пользовательский интерфейс
    var webRequest = (WebRequest)result.AsyncState;
    using (var webResponse = webRequest.EndGetResponse(result)) {
        Text = "Content length: " + webResponse.ContentLength;
    }
}
}
```

Теперь посмотрим на аналогичную версию для WPF¹:

```
private sealed class MyWpfWindow : System.Windows.Window {
    public MyWpfWindow() {
        Title = "Click in the window to start a Web request";
        Width = 400; Height = 100;
    }

    protected override void OnMouseDown(MouseButtonEventArgs e) {
        // GUI-поток инициирует асинхронный веб-запрос
        Title = "Web request initiated";
        var webRequest = WebRequest.Create("http://Wintelllect.com/");
        webRequest.BeginGetResponse(
            SyncContextCallback(ProcessWebResponse), webRequest);
        base.OnMouseDown(e);
    }

    private void ProcessWebResponse(IAsyncResult result) {
        // Если мы попали сюда, это должен быть GUI-поток,
        // значит, обновляем пользовательский интерфейс
        var webRequest = (WebRequest)result.AsyncState;
        using (var webResponse = webRequest.EndGetResponse(result)) {
            Title = "Content length: " + webResponse.ContentLength;
        }
    }
}
```

Асинхронная реализация сервера

Ранее в этой главе уже демонстрировался мой сервер канала, реализованный в виде консольного приложения с использованием преимуществ APM. На основе APM с помощью различных прикладных моделей могут быть реализованы и другие серверы. Однако я встречал множество людей, которые даже не подозревают о такой возможности. Данный раздел призван сообщить, что вы

¹ Версия для Silverlight практически идентична. Различия связаны только с заданием заголовка и определением щелчка мыши, весь же код, связанный с APM, остается тем же.

можете асинхронно реализовать все типы серверов. Детальную информацию вы найдете в документации.NET Framework SDK.

Чтобы асинхронно реализовать страницу ASP.NET Web Form, откройте файл с расширением `.aspx` и добавьте к директиве `Page` запись `"Async=true"`. Затем в коде вызовите метод `AddOnPreRenderCompleteAsync` (ваш класс наследует его от `System.Web.UI.Page`) и передайте ему имена ваших методов `BeginXxx` и `EndXxx`. В методе `BeginXxx` начните асинхронную операцию и позвольте потоку пула вернуться в пул. При этом даже после возвращения потока в пул страница не отправится назад клиенту. После завершения асинхронной операции будет вызван метод `EndXxx`. Возьмите данные, обновите элементы управления страницы и только после этого отправьте ее клиенту.

Для асинхронной реализации веб-службы ASP.NET превратите веб-метод в методы `BeginXxx` и `EndXxx`, воспользовавшись описанной в этой главе схемой. Оба метода пометьте атрибутом `[WebMethod]`.

Для асинхронной реализации веб-службы Windows Communication Foundation определите в своих контрактах методы `BeginXxx` и `EndXxx` по описанному в этой главе образцу. Затем пометьте метод `BeginXxx` атрибутом `[OperationContract(AsyncPattern=true)]`.

Еще раз напомним, что мой класс `AsyncEnumerator` значительно упрощает написание кода для асинхронной реализации серверов.

Модель асинхронного программирования и вычислительные операции

В главе 26 показано, как выполнять вычислительные операции, вызывая метод `QueueUserWorkItem` класса `ThreadPool` и используя класс `System.Threading.Tasks.Task`. Теперь же мы поговорим о том, как выполнять такие операции средствами APM. Наличие в .NET Framework самых разных программных моделей для решения одних и тех же задач не может не огорчать, так как это осложняет выбор разработчикам. А так как многоядерные технологии пока только начинают развиваться, скорее всего, в будущем появятся и другие программные модели. Много лет назад все было проще, но со временем вещи становятся только сложнее. В последнем разделе данной главы мы сравним различные версии APM, предлагаемые в настоящее время .NET Framework.

При помощи APM можно вызвать любой метод, но сначала вам потребуется делегат с сигнатурой, аналогичной сигнатуре этого метода. Предположим, что мы собираемся вызвать метод, складывающий числа от 1 до n . При больших значениях n выполнение этого вычислительного задания (ввода-вывода здесь не требуется) может занять много времени¹. Вот наш метод `Sum`.

¹ Я знаю, что для любых n сумма быстро вычисляется по формуле: $n \times (n + 1) / 2$. Но в данном случае мы намеренно ее забыли и складываем все числа вручную.

```
private static UInt64 Sum(UInt64 n) {
    UInt64 sum = 0;
    for (UInt64 i = 1; i <= n; i++) {
        checked {
            // Я проверяю код, и если сумма не поместится в структуре UInt64,
            // будет выброшено исключение OverflowException
            sum += i;
        }
    }
    return sum;
}
```

При больших значениях *n* метод *Sum* будет работать долго. Чтобы интерфейс моего приложения при этом не зависал, а также чтобы воспользоваться преимуществами второго процессора, выполним метод асинхронно.

Для этого задействуем универсальный делегат *System.Func<T, TResult>*, принимающий параметры двух типов, один — для аргумента, второй — для возвращаемого типа:

```
public delegate TResult Func<T, TResult>(T arg);
```

Как обсуждалось в главе 17, компилятор *C#* преобразует эту строку в определение класса, которое логически выглядит так:

```
public sealed class Func<T, TResult> : MulticastDelegate {
    public Func(Object object, IntPtr method);
    public TResult Invoke(T arg);
    public IAsyncResult BeginInvoke(
        T arg, AsyncCallback callback, Object object);
    public TResult EndInvoke(IAsyncResult result);
}
```

Когда вы определите делегат в исходном коде на *C#*, компилятор создаст класс с методами *BeginInvoke* и *EndInvoke*. Первый метод содержит все параметры из определения делегата, плюс еще два дополнительных параметра в конце: *AsyncCallback* и *Object*. Все методы *BeginInvoke* возвращают объект *IAsyncResult*. Метод *EndInvoke* имеет один параметр — объект *IAsyncResult* и возвращает тот же самый тип данных, что и сигнатура делегата.

Теперь, когда вы вооружены этой информацией, использование делегатов для выполнения вычислительных операций должно стать тривиальной процедурой, так как достаточно следовать эталону АРМ, который мы подробно рассмотрели ранее. Вот код, демонстрирующий асинхронный вызов метода *Sum*:

```
public static void Main() {
    // Переменной делегата присваивается ссылка на асинхронно вызываемый метод
    Func<UInt64, UInt64> sumDelegate = Sum;

    // Вызов метода при помощи потока из пула
    sumDelegate.BeginInvoke(1000000000, SumIsDone, sumDelegate);
}
```

```
// Здесь выполняется какой-то другой код...

// Для демонстрации я просто приостановлю основной поток
Console.ReadLine();
}
```

Переменной `sumDelegate` первым делом присваивается ссылка на метод, который мы собираемся вызвать асинхронно. Затем при помощи метода `BeginInvoke` инициализируется асинхронный вызов метода. Средствами CLR конструируется определяющий асинхронную операцию объект `IAAsyncResult`. Как вы помните, операции ввода-вывода ставятся в очередь драйверов устройств Windows; но метод `BeginInvoke` делегата ставит вычислительную операцию в очередь CLR-пула потоков, вызывая метод `QueueUserWorkItem` класса `ThreadPool`. Ну и, наконец, метод `BeginInvoke` возвращает вызывающему коду объект `IAAsyncResult` (код обычно игнорирует его).

Так как метод `BeginInvoke` ставит операцию в очередь в CLR-пуле потоков, пул просыпается, берет рабочий элемент из очереди и вызывает вычислительный метод (в нашем примере это метод `Sum`). Обычно, вернув управление после выполнения метода, поток возвращается в пул. Но в моем примере во второй с конца параметр метода `BeginInvoke` было передано имя метода (`SumIsDone`). Из-за этого, когда метод `Sum` возвращает управление, поток не возвращается в пул, а вызывает метод `SumIsDone`. Другими словами, после завершения вычислительной операции активизируется обратный вызов, как это обычно происходит при завершении операций ввода-вывода. Вот как выглядит метод `SumIsDone`:

```
private static void SumIsDone(IAAsyncResult ar) {
    // Извлекаем sumDelegate (состояние) из объекта IAAsyncResult
    var sumDelegate = (Func<UInt64, UInt64>) ar.AsyncState;

    try {
        // Получаем и выводим результат
        Console.WriteLine("Sum's result: " + sumDelegate.EndInvoke(result));
    }
    catch (OverflowException) {
        Console.WriteLine("Sum's result is too large to calculate");
    }
}
```

Анализ модели асинхронного программирования

Несмотря на свою горячую приверженность APM, я не могу не признать ряда недостатков данной модели, которые Microsoft стоило бы устранить или хотя бы создать для разработчиков руководство. Обсудим эти недостатки более подробно.

Использование модели асинхронного программирования без пула потоков

В этой главе мы говорили о том, как воспользоваться APM и заставить пул потоков активизировать ваши методы обратного вызова при завершении асинхронных операций. Я показал самый предпочтительный способ работы с APM, требующий минимум ресурсов и дающий в итоге отличную производительность. Но APM может предложить и другие способы оповещения о завершении асинхронных операций.

Все методы `BeginXxx` возвращают ссылку на объект, реализующий интерфейс `AsyncResult`:

```
public interface IAsyncResult {  
    Object AsyncState { get; }  
    WaitHandle AsyncWaitHandle { get; } // Так делать не стоит  
    Boolean IsCompleted { get; } // Так делать не стоит  
    Boolean CompletedSynchronously { get; } // Верно при  
                                           // синхронном завершении  
}
```

Во-первых, если поток вызывает метод `EndXxx`, передавая объект `IAsyncResult` до завершения операции, вызывающий поток блокируется в ожидании завершения и снова начинает работать только после получения результата от метода `EndXxx`. Во-вторых, поток можно заблокировать в ожидании результата путем вызова метода `WaitOne` (о нем мы поговорим в следующей главе) для дескриптора `WaitHandle`, возвращенного после запроса к свойству `AsyncWaitHandle` объекта `IAsyncResult`. Этих первых двух приемов следует избегать, так как, блокируя один поток, они потенциально заставляют пул породить другой.

В-третьих, чтобы узнать о завершении операции, поток может непрерывно в цикле запрашивать свойство `IsCompleted` объекта `IAsyncResult`. Но делать это тоже не следует из-за пустой траты ресурсов процессора. Реализовав циклический опрос, ожидающий завершения вычислительной операции, вы отнимаете время у самой операции, отдаляя время ее завершения. Часто для снижения частоты опроса программисты на каждой итерации цикла вызывают метод `Thread.Sleep`. Сделав это, вы заблокируете и поток, и опрос!

Всегда вызывайте метод `EndXxx`, но делайте это только один раз

Без метода `EndXxx` произойдет утечка ресурсов. В коде некоторых разработчиков метод `BeginXxx` вызывается для записи данных на устройство, а так как после этого данным уже не требуется никакая обработка, вызывать метод `EndXxx` они считают уже лишним. Однако вызвать его требуется по двум причинам. Во-первых, при инициализации асинхронной операции CLR выделяет внутренние ресурсы. И после завершения операции они не освобождаются, пока не будет

вызван метод `EndXxx`. Если этого не произойдет, ресурсы можно будет освободить, только завершив процесс. Во-вторых, инициализируя асинхронную операцию, вы не знаете, будет ли она успешно выполнена или же произойдет сбой. Единственным способом узнать это является вызов метода `EndXxx`, который либо вернет доступное для проверки значение, либо вбросит исключение.

Для одной асинхронной операции вызывать метод `EndXxx` можно не более одного раза. Этот метод получает доступ к внутренним ресурсам и освобождает их. Повторный вызов метода `EndXxx` даст непредсказуемый результат, так как ресурсы уже освобождены. Тем не менее в реальности многократный вызов для одной операции метода `EndXxx` иногда работает — все зависит от написания класса, реализующего интерфейс `IAsyncResult`. Так как специалисты Microsoft пока не определили, как должен себя вести этот класс, разработчики реализуют его по-разному. Наверняка можно сказать только то, что однократный вызов метода `EndXxx` всегда даст требуемый результат.

При вызове метода `EndXxx` всегда используйте тот же самый объект

Какой бы объект вы ни использовали, вызывая метод `BeginXxx`, именно этот объект должен фигурировать при вызове метода `EndXxx`. К примеру, не стоит создавать делегат и вызывать его метод `BeginInvoke`, а потом создавать еще один делегат (того же типа, ссылающийся на тот же самый объект/метод) и вызывать его метод `EndInvoke`. На первый взгляд кажется, что при таком подходе все будет работать (ведь оба делегата идентичны), но на самом деле это не так, потому что объект `IAsyncResult` сохраняет ссылку на объект, использовавшийся при вызове метода `BeginInvoke`, и вызов метода `EndInvoke` с другим делегатом станет источником исключения `InvalidOperationException` со следующим сообщением (имеющийся объект `IAsyncResult` не подходит данному делегату):

The `IAsyncResult` object provided does not match this delegate

Впрочем, код, в котором один объект служит для вызова метода `BeginInvoke`, а другой — для вызова метода `EndInvoke`, может работать для некоторых типов объектов; все зависит от их реализации.

Аргументы `ref`, `out` и `params` в методах `BeginXxx` и `EndXxx`

Если в неасинхронной версии метода используются параметры `out/ref` или если параметры помечены ключевым словом `params`, то параметры методов `BeginXxx` и `EndXxx` будут слегка отличаться от описанных в этой главе. Так как подобная ситуация встречается крайне редко, я не буду приводить пример. Просто вам следует помнить о такой возможности. Если вам вдруг понадобится вызвать подобные методы, вы легко сможете сообразить, как это правильно сделать.

Невозможность отмены асинхронной операции ввода-вывода

В настоящее время способов отмены асинхронной операции ввода-вывода, ожидающей выполнения, не существует. Многие разработчики были бы рады появлению такой возможности, но реализовать ее крайне сложно. В конце концов, если вы запросили у сервера 1000 байт, а затем решили, что они вам больше не требуются, заставить сервер проигнорировать ваш запрос невозможно. Вам остается только дождаться ответа сервера и выбросить его. Ведь в данном случае возникает состояние гонки: ваш запрос на отмену отправленного ранее запроса может прийти сразу после того, как будет прочитан последний байт. И что в этом случае делать приложению? Вам пришлось бы обрабатывать потенциальное состояние гонки в вашем собственном коде и решать, что же делать с данными — отбросить их или же обработать каким-то образом. Некоторые методы `BeginXxx` могут как вернуть объект, реализующий интерфейс `AsyncResult`, так и предложить своего рода метод отмены. В этом случае вы сможете отменить операцию. Узнать, поддерживается ли отмена методом `BeginXxx` или возвращаемым им классом, можно в соответствующей документации.

Потребление памяти

Метод `BeginXxx` создает экземпляр типа, реализующего интерфейс `AsyncResult`. Это означает, что для каждой асинхронной операции создается свой объект. Это повышает нагрузку и заполняет кучу дополнительными объектами, добавляя работы сборщику мусора. В результате имеет место падение производительности приложения. Так что если вы достоверно знаете, что ваши операции ввода-вывода будут выполняться быстро, имеет смысл сделать их асинхронными. Многие разработчики (включая меня) хотели бы, чтобы модель APM возвращала значимые типы или предоставила какой-нибудь другой упрощенный способ идентификации поставленных в очередь асинхронных операций; возможно однажды Microsoft наделит CLR поддержкой такой функциональности.

Некоторые операции ввода-вывода приходится выполнять синхронно

В Win32 API существует множество функций, выполняющих операции ввода-вывода. К сожалению, не все из них допускают асинхронное выполнение. К примеру, Win32-метод `CreateFile` (вызываемый конструктором объекта `FileStream`) всегда выполняется синхронно. При попытке создать или открыть файл на сервере в сети до возвращения управления методом `CreateFile` может пройти несколько секунд — в это время вызывающий поток ничего не делает. Приложения, разработанные с прицелом на оптимальные производительность и масштабируемость, должны использовать Win32-функцию, поддерживающую

асинхронные операции открытия и создания файлов. В этом случае поток не будет ждать ответа с сервера. К сожалению, в Win32 нет функции, подобной `CreateFile`, а, значит, FCL не предлагает эффективного средства асинхронного открытия файлов.

Рассмотрим ситуацию, когда такое поведение становится серьезной проблемой. Представьте, что вам нужно написать простой элемент интерфейса, позволяющий пользователю вводить маршрут доступа к файлу и обеспечивающий автоматическое завершение (примерно как в широко используемом диалоговом окне открытия файла). Этот элемент управления должен задействовать отдельные потоки для нумерации папок, в которых осуществляется поиск файлов, так как в Windows не существует функции асинхронной нумерации файлов. По мере того как пользователь продолжает вводить маршрут доступа, вам придется подключать дополнительные потоки, игнорируя результаты ранее порожденных потоков. В Windows Vista появилась новая Win32-функция `CancelSynchronousIO`. Она позволяет одному потоку отменять синхронную операцию ввода-вывода, проводимую другим. Эта функция не отражена в FCL, но если вы решите воспользоваться ею из управляемого кода, вызовите ее при помощи механизма `P/Invoke`. Сигнатура `P/Invoke` для данного случая рассмотрена в следующем разделе.

Многие полагают, что с синхронным прикладным программным интерфейсом работать проще, и во многих случаях это действительно так. Но иногда именно синхронные интерфейсы сильно осложняют жизнь. Командой разработчиков Windows изучаются интерфейсы, являющиеся исключительно синхронными, и решается, какие из них в будущем должны выйти в виде асинхронных версий. После этого их функциональность будет отражена в FCL.

Любой метод через метод `BeginInvoke` делегата можно вызвать асинхронно, но при этом используется поток, а значит, теряется эффективность. Кроме того, при помощи делегата невозможно вызвать конструктор. Поэтому единственным способом асинхронного конструирования объекта `FileStream` является асинхронный вызов какого-то другого метода, который и создаст данный объект. В Windows отсутствуют функции асинхронного доступа к регистрам, доступа к журналу регистрации событий, получения файлов и вложенных папок из определенной папки или изменения атрибутов файла/папки. И это далеко не полный список.

Проблемы объекта `FileStream`

При создании объекта `FileStream` флаг `FileOptions.Asynchronous` позволяет указать, посредством каких операций — синхронных или асинхронных — вы собираетесь взаимодействовать (что эквивалентно вызову Win32-функции `CreateFile` и передаче ей флага `FILE_FLAG_OVERLAPPED`). При отсутствии этого флага Windows выполняет все операции с файлом в синхронном режиме. Разумеется, ничто не мешает вызвать метод `BeginRead` объекта `FileStream`. С точки зрения приложения это выглядит как асинхронное выполнение операций. Но на самом деле класс

FileStream эмулирует асинхронное поведение при помощи дополнительного потока, который впустую тратит ресурсы и снижает производительность.

В то же время можно создать объект FileStream, указав флаг `FileOptions.Asynchronous`. После этого вы можете вызвать метод `Read` объекта FileStream для выполнения синхронной операции. Класс FileStream эмулирует такое поведение, запуская асинхронную операцию и немедленно переводя вызывающий поток в спящий режим до завершения операции. Это тоже не самый эффективный способ, но он лучше, чем вызов метода `BeginRead` с применением объекта FileStream, созданного без флага `FileOptions.Asynchronous`.

Подведем итоги. При работе с объектом FileStream следует заранее выбрать, синхронным или асинхронным будет ввод-вывод файлов, и установить флаг `FileOptions.Asynchronous` (или не делать этого). После установки флага всегда вызывайте метод `BeginRead`. В противном случае пользуйтесь методом `Read`. Это даст вам наилучшую производительность. Если вы собираетесь выполнить синхронную или асинхронную операцию с объектом FileStream, эффективней всего конструировать данный объект с флагом `FileOptions.Asynchronous`. В качестве альтернативы можно создать два объекта FileStream для одного файла. Один объект FileStream будет открыт для асинхронного ввода-вывода, второй — для синхронного.

Следует также помнить, что драйвер устройства файловой системы NTFS выполняет некоторые операции в синхронном режиме вне зависимости от способа открытия файла. Дополнительную информацию по этой теме вы найдете по адресу <http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B156932>.

Приоритеты запросов ввода-вывода

В главе 25 было показано, каким образом приоритет потока влияет на способ его исполнения. Однако сами потоки также выполняют запросы ввода-вывода к различным аппаратным устройствам для чтения и записи данных. Если время процессора окажется выделено под запросы с низким приоритетом, в очереди очень быстро окажутся сотни и даже тысячи запросов. Так как для их обработки требуется время, скорее всего, поток с низким приоритетом повлияет на быстродействие системы, приостановив более приоритетные потоки. Именно поэтому можно наблюдать снижение быстродействия компьютера при выполнении длительных низкоприоритетных заданий, таких как дефрагментация диска, сканирование на вирусы, индексирование содержимого и т. п.¹

Начиная с Windows Vista, появилась возможность указать приоритет потока при выполнении запросов ввода-вывода. К сожалению, данная функциональ-

¹ Windows-функция SuperFetch использует низкоприоритетные запросы ввода-вывода в своих интересах.

ность еще не включена в FCL; надеюсь, она появится в следующей версии. Однако преимуществом данной функции уже можно воспользоваться при помощи механизма P/Invoking. Вот как выглядит такой код:

```
internal static class ThreadIO {
    public static BackgroundProcessingDisposer BeginBackgroundProcessing(
        Boolean process = false) {

        ChangeBackgroundProcessing(process, true);
        return new BackgroundProcessingDisposer(process);
    }

    public static void EndBackgroundProcessing(Boolean process = false) {
        ChangeBackgroundProcessing(process, false);
    }

    private static void ChangeBackgroundProcessing(
        Boolean process, Boolean start) {
        Boolean ok =
            process ? SetPriorityClass(GetCurrentWin32ProcessHandle(),
                start ? ProcessBackgroundMode.Start : ProcessBackgroundMode.End)
                : SetThreadPriority(GetCurrentWin32ThreadHandle(),
                start ? ThreadBackgrounddMode.Start : ThreadBackgrounddMode.End);
        if (!ok) throw new Win32Exception();
    }

    // Эта структура позволяет инструкции using выйти
    // из режима фоновой обработки
    public struct BackgroundProcessingDisposer : IDisposable {
        private readonly Boolean m_process;
        public BackgroundProcessingDisposer(
            Boolean process) { m_process = process; }
        public void Dispose() { EndBackgroundProcessing(m_process); }
    }

    // См. Win32-функции THREAD_MODE_BACKGROUND_BEGIN
    // и THREAD_MODE_BACKGROUND_END
    private enum ThreadBackgrounddMode { Start = 0x10000, End = 0x20000 }

    // См. Win32-функции PROCESS_MODE_BACKGROUND_BEGIN
    // и PROCESS_MODE_BACKGROUND_END
    private enum ProcessBackgroundMode { Start = 0x100000, End = 0x200000 }

    [DllImport("Kernel32", EntryPoint = "GetCurrentProcess",
        ExactSpelling = true)]
```

```
private static extern SafeWaitHandle GetCurrentWin32ProcessHandle();

[DllImport("Kernel32", ExactSpelling = true, SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
private static extern Boolean SetPriorityClass(
    SafeWaitHandle hprocess, ProcessBackgroundMode mode);

[DllImport(
    "Kernel32", EntryPoint = "GetCurrentThread", ExactSpelling = true)]
private static extern SafeWaitHandle GetCurrentWin32ThreadHandle();

[DllImport("Kernel32", ExactSpelling = true, SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
private static extern Boolean SetThreadPriority(
    SafeWaitHandle hthread, ThreadBackgrounddMode mode);

// http://msdn.microsoft.com/en-us/library/aa480216.aspx
[DllImport(
    "Kernel32", SetLastError = true, EntryPoint = "CancelSynchronousIo")]
[return: MarshalAs(UnmanagedType.Bool)]
private static extern Boolean CancelSynchronousIO(SafeWaitHandle hThread);
}
```

И вот код, демонстрирующий, как все это использовать:

```
public static void Main () {
    using (ThreadIO.BeginBackgroundProcessing()) {
        // Здесь располагается низкоприоритетный запрос ввода-вывода
        // (например, вызов BeginRead/BeginWrite)
    }
}
```

Вы объясняете Windows, что поток должен выполнять низкоприоритетные запросы ввода-вывода при помощи метода `BeginBackgroundProcessing` класса `ThreadIO`. Обратите внимание, что при этом также снижается приоритет выполнения потока процессором. Вернуть поток к выполнению запросов ввода-вывода обычной важности (и к обычному приоритету выполнения потока процессором) можно методом `EndBackgroundProcessing` или же вызвав метод `Dispose` для значения, возвращенного методом `BeginBackgroundProcessing` (при помощи инструкции `using` языка C#, как показано в примере). Поток может влиять только на собственный режим фоновой обработки; Windows не позволяет одному потоку менять режим фоновой обработки другого потока.

Чтобы заставить все потоки в процессе обрабатывать низкоприоритетные запросы ввода-вывода и снизить приоритет выполнения потоков процессором, можно вызвать метод `BeginBackgroundProcessing`, передав ему значение `true` для параметра `process`. Процесс может воздействовать только на собственный фоновый режим обработки; Windows не позволяет потоку менять фоновый режим обработки другого процесса.

ВНИМАНИЕ

В сферу ответственности разработчика входит применение новых фоновых приоритетов, обеспечивающих более высокое быстродействие активных приложений, и забота об отсутствии смены приоритетов. При наличии активных операций ввода-вывода с обычным приоритетом поток, работающий в фоновом режиме, будет получать результаты запросов ввода-вывода с задержкой в несколько секунд. Если поток с низким приоритетом в рамках синхронизации потоков получит право на записание, ожидаемое потоком с обычным приоритетом, последний может оказаться в ожидании окончания обработки низкоприоритетных запросов ввода-вывода. Для возникновения проблемы потоку с фоновым приоритетом не придется даже отправлять запросы на ввод-вывод. Поэтому следует свести к минимуму совместное использование объектов синхронизации потоками с обычным и фоновым приоритетами (а лучше вообще этого избегать). Это избавит вас от смены приоритетов, при которой потоки с обычным приоритетом блокируются из-за того, что право на записание получают потоки с фоновым приоритетом.

Преобразование объекта IAsyncResult в объект Task

Ранее в этой главе была рассмотрена процедура выполнения вычислительной операции при помощи APM. А теперь мы обсудим противоположную ситуацию — выполнение операций ввода-вывода при помощи объекта Task.

В пространстве имен System.Threading.Tasks существует класс TaskFactory. Хотя он обсуждался в главе 26, предлагаемый этим классом метод FromAsync мы не рассматривали. Он принимает четыре аргумента и возвращает ссылку на объект Task. В качестве аргументов выступают методы BeginXxx, EndXxx, состояние Object и при желании значение TaskCreationOptions¹. Например:

```
WebRequest webRequest = WebRequest.Create("http://Wintellect.com/");
webRequest.BeginGetResponse(result => {
    WebResponse webResponse = null;
    try {
        webResponse = webRequest.EndGetResponse(result);
        Console.WriteLine("Content length: " + webResponse.ContentLength);
    }
    catch (WebException we) {
        Console.WriteLine("Failed: " + we.GetBaseException().Message);
    }
});
```

продолжение »

¹ Метод FromAsync имеет перегруженные версии, позволяющие передавать в метод BeginXxx до трех параметров. А для метода BeginXxx, принимающего большее количество параметров, существует перегруженная версия метода FromAsync, принимающая параметр IAsyncResult; здесь вы сами вызываете метод BeginXxx и возвращаемое им значение. По возможности избегайте этой перегруженной версии, так как она далеко не так эффективна, как перегруженная версия метода FromAsync, не принимающая объект IAsyncResult.


```
finally { if (webResponse != null) webResponse.Close(); }
}. null);
```

Вместо выполнения этой асинхронной операции можно перейти к объекту Task и использовать его вместе со всей прилагающейся инфраструктурой:

```
WebRequest webRequest = WebRequest.Create("http://Wintellect.com/");
Task.Factory.FromAsync<WebResponse>(
    webRequest.BeginGetResponse, webRequest.EndGetResponse,
    null, TaskCreationOptions.None)
    .ContinueWith(task => {
        WebResponse webResponse = null;
        try {
            webResponse = task.Result;
            Console.WriteLine("Content length: " + webResponse.ContentLength);
        }
        catch (AggregateException ae) {
            if (ae.GetBaseException() is WebException)
                Console.WriteLine("Failed: " + ae.GetBaseException().Message);
            else throw;
        }
        finally { if (webResponse != null) webResponse.Close(); }
    });
```

Следует упомянуть, что класс Task реализует интерфейс IAsyncResult, поэтому задания до определенной степени поддерживают АРМ. Свойство AsyncState объекта Task является свойством AsyncState объекта IAsyncResult, и оно возвращается при передаче параметру state объекта TaskFactory любого состояния. Так как объект Task автоматически вызовет для вас метод EndXxx, потенциально интересными являются такие члены интерфейса IAsyncResult, как AsyncWaitHandle, IsCompleted и, возможно, CompletedSynchronously. Но как уже упоминалось в этой главе, свойств AsyncWaitHandle и IsCompleted следует избегать, а свойство CompletedSynchronously является, скорее, информативным, чем пригодным к практическому применению.

Эталон асинхронного программирования на базе событий

В начале своего существования платформа .NET Framework предлагала единственную модель асинхронного программирования на основе интерфейса IAsyncResult. Именно ее мы обсуждаем на протяжении всей главы. Однако команда разработчиков Windows Forms решила, что эта модель слишком сложна, и предложила новую — эталон асинхронного программирования на базе событий (Event-based Asynchronous Pattern, EAP)¹. Основным преимуществом

¹ Многие сотрудники Microsoft перешли из команды, занимающейся разработкой Windows Forms, в команду разработчиков WPF, поэтому приложения WPF и Silverlight также приняли модель EAP.

этой модели стало ее объединение с пользовательским интерфейсом приложения Microsoft Visual Studio. То есть большинство классов, реализующих EAP, в Visual Studio можно перетащить в область конструирования приложения. Затем достаточно двойного щелчка на именах событий для автоматической генерации методов обратного вызова и связывания этих методов с событиями.

Многие люди, включая меня, полагают, что EAP не следовало вводить в .NET Framework, так как это породило еще больше сложностей, чем было раньше. К примеру, непонятно, следует ли теперь в классах, предлагающих асинхронное поведение, реализовывать оба эталона программирования? Или у нас будут два класса с одинаковой функциональностью, но отличающиеся эталонами асинхронного программирования? И по каким критериям пользователи должны выбирать эталон? Существуют и другие технические проблемы, о которых мы поговорим в конце данного раздела.

Меня часто спрашивают о странице MSDN по адресу <http://msdn2.microsoft.com/en-gb/library/ms228966.aspx>. Там разработчикам классов рекомендуется реализовывать асинхронное поведение посредством модели EAP, а не APM. Также там написано: «Эталон `IAsyncResult` редко реализуется без реализации эталона на базе событий». Эта страница была написана разработчиками из команды Windows Forms, и в Microsoft практически нет сотрудников, согласных с написанным. В действительности Microsoft предлагает только три класса, реально реализующие оба эталона и подходящие под фразу с этой страницы. Так как я не являюсь сторонником эталона EAP и не одобряю его применения, у меня нет желания тратить на него много времени. Однако я знаю людей, которым он нравится и которые хотели бы с ним работать, поэтому мы его все-таки рассмотрим.

Так как эталон EAP был создан разработчиками Windows Forms, приведем использующий его код Windows Forms:

```
internal static class Eap {
    public static void Main() {
        // Создание и вывод формы
        Application.Run(new MyForm());
    }

    private sealed class MyForm : Form {
        protected override void OnClick(EventArgs e) {
            // Класс System.Net.WebClient поддерживает эталон EAP
            WebClient wc = new WebClient();

            // В момент завершения загрузки объект WebClient генерирует
            // событие DownloadStringCompleted, вызывающее метод ProcessString
            wc.DownloadStringCompleted += ProcessString;

            // Начало асинхронной операции (например, вызова метода BeginXxx)
            wc.DownloadStringAsync(new Uri("http://Wintellect.com"));
            base.OnClick(e);
        }
    }
}
```

продолжение ➤

```

// Этот метод гарантированно вызывается GUI-потоком
private void ProcessString(
    Object sender, DownloadStringCompletedEventArgs e) {
    // В случае ошибки выводим ее;
    // в противном случае выводим загруженную строку
    MessageBox.Show((e.Error != null) ? e.Error.Message : e.Result);
}
}
}

```

Код данного примера написан вручную, но можно было воспользоваться приложением Visual Studio и перетащить на форму элементы управления WebClient. После этого приложение Visual Studio вызвало бы метод ProcessString без кода и создало бы код, регистрирующий метод с событием DownloadStringCompleted. Эталон EAP гарантирует появление события в GUI-потоке приложения, что позволяет коду метода обработки события обновить элементы управления интерфейса. Это еще одна замечательная EAP-функция вдобавок к поддержке конструктора Visual Studio. То есть классы, поддерживающие EAP, автоматически накладывают прикладную модель на потоковую модель; это достигается благодаря классу SynchronizationContext. Кроме того, некоторые EAP-классы предлагают механизм отмены и отчет о выполнении работы.

В FCL имеется всего 17 типов, реализующих эталон EAP. Некоторые из них являются производными от класса System.ComponentModel.Component, позволяющего перетаскивать элементы в область конструирования Visual Studio, но большинство наследует непосредственно от класса System.Object. Далее представлен список поддерживающих EAP классов.

Типы, производные от System.ComponentModel.Component (последний тип является производным от класса Control):

```

System.ComponentModel.BackgroundWorker
System.Media.SoundPlayer
System.Net.WebClient
System.Net.NetworkInformation.Ping
System.Windows.Forms.PictureBox

```

Типы, производные от System.Object:

```

System.Net.Mail.SmtpClient
System.Deployment.Application.ApplicationDeployment
System.Deployment.Application.InPlaceHostingManager
System.Activities.WorkflowInvoker
System.ServiceModel.Activities.WorkflowControlClient
System.Net.PeerToPeer.PeerNameResolver
System.Net.PeerToPeer.Collaboration.ContactManager
System.Net.PeerToPeer.Collaboration.Peer
System.Net.PeerToPeer.Collaboration.PeerContact
System.Net.PeerToPeer.Collaboration.PeerNearMe

```

```
System.ServiceModel.Discovery.AnnouncementClient  
System.ServiceModel.Discovery.DiscoveryClient
```

FCL предлагает 60 классов, реализующих эталон `IAAsyncResult`, в том числе и следующие классы, для которых в эталоне EAP не существует эквивалента: различные классы, производные от `Stream` (`FileStream`, `IsolatedStorageFileStream`, `DeflateStream`, `GZipStream` и `PipeStream`), `SqlCommand` и многие другие.

Следует упомянуть, что инструменты, создающие классы-представители для веб-служб, например `WSDL.exe` и `SvcUtil.exe`, предлагают классы, поддерживающие оба эталона (APM и EAP).

Если внимательно посмотреть на перечисленные 17 классов, выяснится, что все они за исключением класса `BackgroundWorker` связаны с вводом-выводом. Класс `BackgroundWorker` планировался для асинхронных вычислительных операций, но, к сожалению, большинство разработчиков использовало его для синхронного ввода-вывода, блокировавшего поток. Операции ввода-вывода следует выполнять при помощи остальных 16 EAP-классов или любых APM-классов. Класс `BackgroundWorker` предлагает следующие события:

- ❑ `DoWork`. Зарегистрированный с этим событием метод должен содержать вычислительный код. Событие инициируется потоком пула.
- ❑ `ProgressChanged`. Метод, зарегистрированный с этим событием, должен содержать код, обновляющий пользовательский интерфейс, с информацией о состоянии задания. Событие всегда инициируется GUI-потоком. Его обработчик должен периодически вызывать метод `ReportProgress` класса `BackgroundWorker` для активизации события `ProgressChanged`.
- ❑ `RunWorkerCompleted`. Метод, зарегистрированный с этим событием, должен содержать код, обновляющий пользовательский интерфейс с результатами вычислительной операции. Это событие также всегда инициируется GUI-потоком. Обработчик события `DoWork` передается по ссылке объекту `DoWorkEventArgs`. Свойству `Result` этого объекта и присваивается значение, которое пытается вернуть вычислительная операция.

Преобразование эталона асинхронного программирования на базе событий в объект `Task`

Ранее в этой главе было показано, как средствами `IAAsyncResult` эталона APM превратить асинхронную операцию в объект `Task`, дав возможность использовать его с инфраструктурой этого объекта. Эталон EAP также позволяет осуществлять подобные манипуляции. В пространстве имен `System.Threading.Tasks` класс `TaskCompletionSource` определен следующим образом:

```
public class TaskCompletionSource<TResult> {  
    public TaskCompletionSource();
```

продолжение ➤

```

public TaskCompletionSource(
    Object state, TaskCreationOptions creationOptions);
public void SetCanceled();
public void SetException(IEnumerable<Exception> exceptions);
public void SetResult(TResult result);
public Task<TResult> Task { get; }
// Менее важные методы не показаны
}

```

Создание объекта `TaskCompletionSource` приводит к появлению объекта `Task`, на который можно сослаться через свойство `Task` класса `TaskCompletionSource`. После завершения асинхронной операции объекту `TaskCompletionSource` присваивается значение, соответствующее причине завершения: отмена, необработанное исключение или результат операции. Состояние лежащего в основе объекта `Task` задается методами `SetXxx`. Вот код, демонстрирующий преобразование эталона EAP в задание (объект `Task`):

```

internal sealed class MyFormTask : Form {
    protected override void OnClick(EventArgs e) {
        // Класс System.Net.WebClient поддерживает EAP
        WebClient wc = new WebClient();

        // Создание объектов TaskCompletionSource и Task
        var tcs = new TaskCompletionSource<String>();

        // Когда завершается загрузка строки, объект WebClient активизирует
        // событие DownloadStringCompleted, вызывающее метод ProcessString
        wc.DownloadStringCompleted += (sender, ea) => {
            // Этот код всегда выполняется GUI-поток; задает состояние задания
            if (ea.Cancelled) tcs.SetCanceled();
            else if (ea.Error != null) tcs.SetException(ea.Error);
            else tcs.SetResult(ea.Result);
        };

        // Продолжает задание объект Task, выводящий результат в окне
        // ПРИМЕЧАНИЕ. Флаг ExecuteSynchronously обеспечивает выполнение кода
        // GUI-поток; без него код будет выполняться потоком пула
        tcs.Task.ContinueWith(t => {
            try {
                MessageBox.Show(t.Result);
            }
            catch (AggregateException ae) {
                MessageBox.Show(ae.GetBaseException().Message);
            }
        }, TaskContinuationOptions.ExecuteSynchronously);

        // Начало асинхронной операции (например, вызов метода BeginXxx)
        wc.DownloadStringAsync(new Uri("http://Wintellect.com"));
    }
}

```

```
base.OnClick(e);  
}  
}
```

Сравнение эталонов АРМ и ЕРМ

Самым большим преимуществом эталона ЕРМ перед АРМ является возможность использовать его в программе Visual Studio, которая дает возможность вызывать асинхронные операции непосредственно во время разработки. Кроме того, эталон ЕАР появился в FCL одновременно с классом `SynchronizationContext`, поэтому он обладает встроенной способностью понимать потоковую модель приложения. Это гарантирует, что в приложениях с графическим интерфейсом методы обработки событий будут выполняться в GUI-потоке.

Однако эталон АРМ ближе к аппаратуре, поэтому внутренне ЕАР-классы обычно реализуются с использованием АРМ. Это означает, что ЕАР-классы требуют больше памяти и работают медленнее своих АРМ-эквивалентов. Ведь ЕАР приходится выделять место под объекты, производные от класса `EventArgs`, для всех событий, связанных с отчетом о выполняемой работе и завершением. Кроме того, некоторые ЕРМ-классы содержат коллекции объектов `UserState`, идентифицирующие отдельные операции, а также объект `AsyncOperation`. В типичном приложении с графическим интерфейсом это дополнительное место в памяти (приводящее к дополнительной загрузке сборщика мусора), скорее всего, не имеет особого значения. Однако при создании высокопроизводительных серверных приложений применять ЕРМ вряд ли стоит.

Для простых сценариев ЕРМ — прекрасный выбор, так как этот эталон прост в применении. Но встречаются и сценарии, в которых воспользоваться им крайне сложно. Если метод `XxxAsync` вызывается до регистрации метода обработки события, асинхронная операция также может завершиться раньше, в итоге обработчик события так и не будет вызван. А так как события имеют свойство накапливаться, придется отменить имеющуюся регистрацию и зарегистрировать с событием следующий метод на случай, если вы хотите, чтобы ваша асинхронная операция пользовалась другим методом. Статические методы и одноэлементные классы по ряду причин не работают с ЕРМ: различные части приложений могут регистрировать события одновременно, в результате при завершении операции вызываются все обработчики вне зависимости от того, какой частью приложения выполнялась асинхронная операция.

Обработка ошибок в ЕАР не стыкуется с остальной частью системы. И в первую очередь, не вбрасываются исключения; следует проверить, имеет ли свойство `Exception` класса `AsyncCompletedEventArgs` в обработчике события значение `null` или нет. Если оно не равно `null`, для определения типа объекта, производного от класса `Exception`, следует воспользоваться инструкцией `if` вместо блоков `catch`. А если ваш код игнорирует ошибку, появляется необработанное исключение, а значит, ошибка остается нераспознанной, и результат работы приложения становится непредсказуемым.

Сводная информация по моделям программирования

С годами в .NET Framework вводились различные модели асинхронного программирования, каждая из которых имеет свои сильные и слабые стороны. И я уверен, что в будущем можно ждать появления новых моделей. Чтобы облегчить вам жизнь, я суммировал сведения о существующих в настоящее время моделях в табл. 27.1. Здесь есть все: основное назначение каждой модели (вычислительные операции или ввод-вывод), способности к эмуляции остальных видов операций, поддержка наследования, наличие встроенных отчетов о ходе выполнения, поддержка отмены, блокировка потока до завершения операции (этой функциональности лучше избегать), уведомление об окончании времени ожидания и возможность получения результата (или исключения) операции после ее завершения. Вот дополнительные замечания по поводу таблицы.

- ❑ В случаях когда из-за перехвата очереди скапливается множество заданий, объекты `Task` предлагают более высокую производительность, чем метод `ThreadPool.QueueUserWorkItem` или метод `BeginInvoke` делегата.
- ❑ Флаг `PreFairness` позволяет добиться того же поведения пула, что и метод `ThreadPool.QueueUserWorkItem` или метод `BeginInvoke` делегата.
- ❑ Можно использовать настроенный планировщик `TaskScheduler`, позволяющий менять алгоритм планирования, не вмешиваясь в код модели программирования.
- ❑ Объекты `Task` потребляют больше памяти, чем вызов метода `ThreadPool.QueueUserWorkItem` или метода `BeginInvoke` делегата. Впрочем, последний метод является причиной известных проблем производительности, поэтому, несмотря на большее потребление памяти объектом `Task`, задания выполняются быстрее и могут рассматриваться как более разумный выбор.
- ❑ АРМ-интерфейс `IAsyncResult` предлагает четыре варианта взаимодействия, что усложняет модель. Но если ограничить выбор методом обратного вызова (как это делаю я), модель кажется намного проще.
- ❑ АРМ-интерфейс `IAsyncResult` в общем случае работает быстрее и потребляет меньше ресурсов, чем эталон `EAP`.
- ❑ Некоторые реализованные в `EAP` классы поддерживают отмену.
- ❑ АРМ-интерфейс `IAsyncResult` вообще не поддерживает отмену; однако нужное поведение всегда можно получить, установив флаг и отбрасывая результаты после их получения. Используйте в качестве оболочки эталона `IAsyncResult` объект `Task` и корректно настройте обратные вызовы `ContinueWith`.
- ❑ Эталон `EAP` основан на событиях, поэтому вы легко можете использовать его в рабочей области приложений `Windows Forms`, `WPF` и `Silverlight`, а методы уведомления будут вызываться в нужном потоке пользовательского интерфейса.

Таблица 27.1. Сравнение асинхронных моделей программирования в .NET Framework

Модель	Назначение	Эмуляции	Наследование	Отчет о выполнении	Отмена	Ожидание	Таймаут	Возвращение исключения
QueueUserWorkItem	Вычисления	Синхронный ввод-вывод	Нет	Нет	Нет	Нет	Нет	Нет
Timer	Вычисления	Синхронный ввод-вывод	Нет	Нет	Через Dispose	Нет	Да	Нет
RegisterWaitForSingleObject	Вычисления	Синхронный ввод-вывод	Нет	Нет	Через Unregister	Нет	Да	Нет
Tasks	Вычисления	Синхронный ввод-вывод или taskCompletionSource и FromAsync планировщика заданий	Да	Нет	Вычисления: перед началом задания, если оно поддерживает отмену. Ввод-вывод: результат сбрасывается	Да	Да	Да
IAsyncResult APM	Ввод-вывод	BeginInvoke	Нет	Нет	Нет	Да	Нет	Да
Event-based PM	Ввод-вывод	BackgroundWorker	Нет	Иногда	Некоторые типы отменяют результат	Нет	Нет	Да
AsyncEnumerator	Ввод-вывод	BeginInvoke	Нет	Нет	Да	Нет	Да	Да

Глава 28. Простейшие конструкции синхронизации потоков

При блокировке потока в пуле пул порождает дополнительные потоки исполнения, хотя при этом приходится тратить соответствующие ресурсы (времени и памяти) на создание, планирование и удаление потоков. Многие разработчики, обнаружив в программе простаивающие потоки, считают, что дополнительные потоки уж точно будут делать что-нибудь полезное. Однако при разработке масштабируемого и быстродействующего приложения нужно стараться избегать блокировки потоков, только в этом случае их можно будет снова и снова использовать для решения других задач. В главе 26 мы говорили о том, как потоки выполняют вычислительные операции, в то время как глава 27 была посвящена выполнению потоками операций ввода-вывода.

Теперь пришло время обсудить вопросы синхронизации потоков. Синхронизация позволяет предотвратить повреждение общих данных при *одновременном* доступе к этим данным разных потоков. Слово «одновременно» выделено не зря, ведь синхронизация потоков целиком и полностью базируется на контроле времени. Если доступ к неким данным со стороны двух потоков осуществляется таким образом, что потоки никак не могут помешать друг другу, синхронизации не требуется. В главе 27 демонстрировался код, реализующий серверный именованный канал, который действует следующим образом. В методе `ClientConnected` поток выделяет место под массив `Byte[]`, предназначенный для заполнения получаемыми от клиента данными. При отправке клиентом данных другой поток пула вызывает метод `GetRequest`, обрабатывающий данные в массиве `Byte[]`. То есть мы имеем два потока, работающие с одними и теми же данными. Однако архитектура приложения исключает одновременный доступ этих потоков к одному и тому же фрагменту массива `Byte[]`. Именно поэтому для именованного канала в этом приложении синхронизация не предусмотрена.

Этот случай можно считать идеальным, так как синхронизация потоков влечет за собой много проблем. Во-первых, программировать код синхронизации крайне утомительно и при этом легко допустить ошибку. В коде следует выделить все данные, которые потенциально могут обрабатываться различными потоками в одно и то же время. Затем пишется дополнительный код, обрамляющий эти данные и призванный обеспечить их запираение и отпираение. Запираение гарантирует, что доступ к ресурсу в каждый момент времени сможет получить только один поток. Однако достаточно при программировании забыть запереть хотя бы один фрагмент кода, и ваши данные будут повреждены. К тому же нет способа проверить, правильно ли работает запирающий код. Остается только запустить приложение, провести многочисленные нагрузочные испытания и надеяться, что все пройдет благополучно. При этом тестирование

желательно осуществлять на машине с максимально возможным количеством процессоров, так как это повышает шансы выявить ситуацию, когда два и более потока попытаются получить одновременный доступ к ресурсу. А значит, повышаются шансы распознать проблему.

Второй проблемой записи является снижение производительности. Записи и отписи требуют времени, так как для этого вызываются дополнительные методы, причем процессоры должны координировать совместную работу, определяя, который из потоков нужно записывать первым. Подобное взаимодействие процессоров не может не сказываться на производительности. К примеру, рассмотрим код, добавляющий узел в начало связанного списка:

```
// Этот класс используется классом LinkedList
public class Node {
    internal Node m_next;
    // Остальные члены не показаны
}

public sealed class LinkedList {
    private Node m_head;

    public void Add(Node newNode) {
        // Эти две строки реализуют быстрое присваивание ссылок
        newNode.m_next = m_head;
        m_head = newNode;
    }
}
```

Метод Add просто очень быстро присваивает ссылки. И если мы хотим сделать вызов этого метода безопасным, дав возможность разным потокам одновременно вызывать его без риска повредить связанный список, следует добавить к методу Add код записи и отписи:

```
public sealed class LinkedList {
    private SomeKindOfLock m_lock = new SomeKindOfLock();
    private Node m_head;

    public void Add(Node newNode) {
        m_lock.Acquire();
        // Эти две строки реализуют быстрое присваивание ссылок
        newNode.m_next = m_head;
        m_head = newNode;
        m_lock.Release();
    }
}
```

Теперь метод Add стал безопасным в отношении потоков, но скорость его выполнения серьезно упала. Снижение скорости работы зависит от вида выбранного механизма записи; сравнение производительности различных

вариантов записи делается как в этой, так и в следующей главах. Но даже самое быстрое записи заставляет метод `Add` работать в несколько раз медленнее по сравнению с его версией без записи. И разумеется, вызов метода `Add` в цикле для вставки в связанный список дополнительных узлов также значительно снижает производительность.

Третья проблема состоит в том, что при записи в каждый момент времени допускается доступ к ресурсам только одного потока. Собственно, для этого и было придумано записи, но, к сожалению, подобное поведение приводит к созданию дополнительных потоков. То есть если поток пула пытается получить доступ к запертому ресурсу и не получает его, скорее всего, пул создаст еще один поток для сохранения загрузки процессора. Как обсуждалось в главе 25, эта процедура обходится крайне дорого в смысле затрат памяти и снижения производительности. Но хуже всего то, что после разблокирования старый поток появляется в пуле вместе с новым; то есть операционной системе приходится планировать работу потоков, количество которых превышает количество процессоров, а значит, имеет место переключение контекста, что, опять же, отрицательно сказывается на производительности.

Словом, синхронизация потоков является той процедурой, которой по возможности следует избегать. Не помещайте общие данные в статические поля. Когда поток конструирует новый объект при помощи оператора `new`, оператор возвращает ссылку на этот объект. Причем в этот момент ссылка имеется только у создающего объект потока, другие потоки не имеют к нему доступа. Если не передавать эту ссылку другому потоку, который может использовать объект одновременно с потоком, создавшим объект, необходимость в синхронизации отпадает.

Пытайтесь по возможности работать со значимыми типами, потому что они всегда копируются, и каждый поток в итоге работает с собственной копией. Ну и, наконец, нет ничего страшного в одновременном доступе разных потоков к общим данным, если эти данные предназначены только для чтения. К примеру, многие приложения в процессе инициализации создают структуры данных. После инициализации приложение может создать столько потоков, сколько считает необходимым; и если все эти потоки решат получить доступ к этим данным, они смогут сделать это одновременно, не прибегая ни к записи, ни к отписи. Примером такого поведения может служить тип `String`. Созданные строки неизменны, поэтому одновременный доступ к ним можно предоставить произвольному количеству потоков, не боясь, что данные будут повреждены.

Библиотеки классов и безопасность потоков

А сейчас хотелось бы сказать несколько слов о библиотеках классов и синхронизации потоков. Библиотека `FCL` разработки Microsoft гарантирует безопасность в отношении потоков всех статических методов. Это означает, что

одновременный вызов статического метода двумя потоками не приводит к повреждению данных. Механизм защиты реализован внутри FCL, поскольку нет способа обеспечить записание сборок различных производителей, спорящих за доступ к ресурсу. Класс `Console` содержит статическое поле, внутри которого многие из его методов запираются и отпираются, гарантируя, что в каждый момент времени доступ к консоли будет только у одного потока.

Кстати, создание метода, безопасного в отношении потоков, не означает, что внутренне он реализует записание в рамках синхронизации потоков. Просто этот метод сохраняет данные неповрежденными при попытке одновременного доступа к ним со стороны нескольких потоков. Класс `System.Math` обладает статическим методом `Max`, который реализован следующим образом:

```
public static Int32 Max(Int32 val1, Int32 val2) {  
    return (val1 < val2) ? val2 : val1;  
}
```

Этот метод безопасен в отношении потоков, хотя в нем нет никакого кода записания. Так как тип `Int32` относится к значимым, два значения этого типа при передаче в переменную `Max` копируются, а значит, разные потоки могут одновременно обращаться к данной переменной. При этом каждый поток будет работать с собственными данными, изолированными от всех прочих потоков.

В то же время FCL не гарантирует безопасности в отношении потоков экземплярным методам, так как введение в них запирающего кода слишком сильно сказывается на производительности. Более того, если каждый экземплярный метод начнет выполнять записание и отпирание, все закончится тем, что в приложении в каждый момент времени будет исполняться только один поток, что еще больше снизит производительность. Как уже упоминалось, поток, конструирующий объект, является единственным, кто имеет к нему доступ. Другим потокам данный объект недоступен, а значит, при вызове экземплярных методов синхронизация не требуется. Однако если потом поток отдаст ссылку на объект, поместив ее в статическое поле, передав ее в качестве аргумента состояния методу `ThreadPool.QueueUserWorkItem` или объекту `Task` и т. п., то тут синхронизация уже понадобится, если разные потоки попытаются одновременно получить доступ к данным не только для чтения.

Собственные библиотеки классов рекомендуется строить по следующему эталону. Все статические методы следует сделать безопасными в отношении потоков, а экземплярные методы — нет. Впрочем, следует оговорить, что если целью экземплярного метода является координирование потоков, его тоже следует сделать безопасным в отношении потоков. К примеру, один поток может отменять операцию, вызывая метод `Cancel` класса `CancellationTokenSource`, а другой поток, делая запрос к соответствующему свойству `IsCancellationRequested` объекта `CancellationToken`, может обнаружить, что отмена на самом деле не нужна. Внутри этих экземплярных методов есть специальный код синхронизации потоков, гарантирующий их скоординированную работу¹.

¹ Поле, к которому осуществляют доступ оба члена, помечается ключевым словом `volatile`, о котором мы поговорим чуть позже.

Простейшие конструкции пользовательского режима и режима ядра

В этой главе рассмотрены примитивные конструкции для синхронизации потоков. Под «примитивными» я подразумеваю простейшие конструкции, которые доступны в коде. Они бывают двух видов: пользовательского режима и режима ядра. По возможности нужно задействовать первые, так как они значительно быстрее вторых и используют для координации потоков специальные директивы процессора. То есть координация имеет место уже на аппаратном уровне (и именно это обеспечивает быстроедействие). Однако одновременно это означает, что блокировка потоков на уровне примитивной конструкции пользовательского режима операционной системой Windows просто не распознается. А так как заблокированным таким способом поток пула не считается таковым, пул не создает дополнительных потоков для восполнения загрузки процессора. Кроме того, блокировка происходит на очень короткое время.

Звучит заманчиво, не правда ли? Более того, все действительно так, именно поэтому я рекомендую использовать эти конструкции как можно чаще. Впрочем, они не лишены недостатков. Только ядро операционной системы Windows может остановить выполнение потока, чтобы он перестал впустую расходовать ресурсы процессора. Запущенный в пользовательском режиме поток может быть прерван операционной системой, но довольно быстро снова будет готов к работе. В итоге поток, который пытается, но не может получить некоторый ресурс, начинает циклически существовать в пользовательском режиме. Потенциально это является пустым расходом времени процессора, которое лучше было бы потратить с пользой.

Это заставляет нас перейти к примитивным конструкциям режима ядра. Они предоставляются самой операционной системой Windows и требуют от потоков приложения вызова функций, реализованных в ядре. Переход потока между пользовательским режимом и режимом ядра значительно снижает производительность, поэтому конструкций режима ядра крайне желательно избегать¹. Однако и у них есть свои достоинства. Если один поток использует конструкцию режима ядра для получения доступа к ресурсу, с которым уже работает другой поток, Windows блокирует его, чтобы не тратить понапрасну время процессора. А затем, когда ресурс становится доступным, блокировка снимается, и поток получает доступ к ресурсу.

Если поток, использующий в данный момент конструкцию, не освободит ее, ожидающий конструкции поток может оказаться заблокированным навсегда. В этом случае в пользовательском режиме поток бесконечно исполняется процессором; этот вариант блокировки называется *живым затиранием* (livelock), или *зависанием*. В режиме ядра поток блокируется навсегда, этот тип блокировки называется *мертвым затиранием* (deadlock), или *клинчем*. Обе ситуации по-своему плохи, но если выбирать из двух зол, второй вариант видится более предпочтитель-

¹ Чуть дальше в этой главе показана программа, измеряющая производительность.

ным, потому что в первом случае впустую расходуются как время процессора, так и память (стек потока и т. п.), а во втором случае расходует только память¹.

В идеальном мире у нас были бы самые лучшие конструкции. Быстро работающие и не блокирующиеся (как конструкции пользовательского режима) в условиях отсутствия конкуренции. А если конструкции начинали бы соперничать друг другом, их блокировало бы ядро операционной системы. Описанные конструкции даже существуют в природе, я называю их *гибридными* (hybrid constructs), и мы рассмотрим их в следующей главе. Именно гибридные конструкции обычно используются в приложениях, так как в большинстве приложений несколько потоков крайне редко пытаются одновременно получить доступ к одним и тем же данным. Гибридные конструкции основное время поддерживают быструю работу приложения и периодически замедляются, блокируя поток. Однако это замедление в тот момент не имеет особого значения.

Многие из конструкций синхронизации потоков в CLR являются всего лишь объектно-ориентированными оболочками классов, построенных на базе конструкций синхронизации потоков Win32. В конце концов, CLR-потоки являются потоками операционной системы Windows, которая планирует и контролирует их синхронизацию. Конструкции синхронизации существуют с 1992 года и о них написано множество книг². Поэтому мы не будем подробно останавливаться на них в этой главе.

Конструкции пользовательского режима

Существует два типа примитивных конструкций синхронизации потоков пользовательского режима:

- ❑ *Волатильные конструкции* (volatile constructs) выполняют для переменной, содержащей данные простого типа, атомарную операцию чтения *или* записи.
- ❑ *Взаимозапирающие конструкции* (interlocked constructs) выполняют для переменной, содержащей данные простого типа, атомарную операцию чтения *и* записи.

Конструкции обоих типов требуют передачи ссылки (адреса в памяти) на переменную, принадлежащую к простому типу данных. Архитектура некоторых процессоров также требует корректного выравнивания этого адреса в памяти, в противном случае вбрасывается исключение `DataMisalignedException`.

Это означает, что переменные, содержащие 1-разрядные, 2-разрядные и 4-разрядные значения, должны располагаться по адресу, кратному 1, 2 или 4

¹ Я считаю, что выделенная потоку память расходует впустую, если поток не выполняет никакой полезной работы.

² В моей книге «Windows via C/C++» (Microsoft Press, 2007) этой теме посвящено несколько глав.

соответственно, а переменная, содержащая 8-разрядное значение, располагаться по адресу, допускающему атомарные манипуляции со стороны аппаратного обеспечения (кратному 4 или 8). А конкретно это означает, что переменные типа (S)Byte выравниваются по 1-байтной границе, типа (U)Int16 — по 2-байтной, типов (U)Int32 и Single — по 4-байтной, а типов (U)Int64 и Double — по 4- или 8-байтной. Все ссылочные переменные и переменные типа (U)IntPtr в 32-разрядном процессе имеют ширину 4 байта, в 64-разрядном процессе — 8 байт. То есть эти переменные всегда выровнены по 4- или 8-байтной границе в зависимости от типа процесса.

К счастью, CLR гарантирует автоматическое выравнивание полей, если, конечно, тип, которому принадлежат поля, не имеет атрибута [StructLayout(LayoutKind.Explicit)] либо атрибута [FieldOffset(...)], примененного к отдельным полям. Если вы не используете данные атрибуты, препятствующие выравниванию полей, проблем с применением конструкций пользовательского режима не будет.

Доступ к корректно выровненным переменным любого из упомянутых типов всегда является атомарным. Это означает одновременное чтение всех байтов внутри переменной или их одновременную запись. К примеру, рассмотрим такой класс:

```
internal static class SomeType {  
    public static Int32 x = 0;  
}
```

Если какой-то поток выполняет следующую строку кода, то переменная *x* сразу (атомарно) изменяется с 0x00000000 до 0x01234567:

```
SomeType.x = 0x01234567;
```

Атомарно означает, что посторонние потоки не могут видеть переменную в промежуточном состоянии. К примеру, другой поток не может сделать запрос к свойству *SomeType.x* и получить значение 0x01230000. Однако если чтение и запись в корректно выровненную переменную осуществляются атомарно, то из-за работы компилятора и оптимизации процессора вы не можете знать, *когда именно* это произойдет. Волатильные конструкции не просто гарантируют атомарность операций чтения и записи, но, что еще важнее, управляют временем их выполнения. Взаимозапирающие конструкции позволяют выполнять и более сложные операции, чем простые чтение и запись, и при этом тоже управляют временем их выполнения.

Предположим, что поле *x* класса *SomeType*, принадлежащее типу *Int64*, не было корректно выровнено. Если поток выполнит следующую строку кода, то другой поток сможет сделать запрос переменной *x* и получить значение 0x0123456700000000 или 0x0000000089abcdef, так как операции чтения и записи не являются атомарными:

```
SomeType.x = 0x0123456789abcdef;
```

Это пример так называемого *прерванного чтения* (torn read).

Волатильные конструкции

Когда компьютеры только появились, программное обеспечение писалось на языке ассемблер. Это было крайне утомительное занятие, так как программист должен был формулировать все в явном виде: использовать определенный регистр процессора, передать управление, осуществить неявный вызов и т. п. Для упрощения задачи были придуманы языки высокого уровня. Именно в них впервые были реализованы привычные конструкции `if/else`, `switch/case`, циклы, локальные переменные, аргументы, вызовы виртуальных методов, перегрузка операторов и многое другое. В конечном итоге компилятор преобразует конструкции высокого уровня в низкоуровневые, позволяющие компьютеру понять, что именно ему следует делать.

Другими словами, компилятор C# преобразует конструкции языка C# в команды промежуточного языка (Intermediate Language, IL), которые, в свою очередь, JIT-компилятор превращает в машинные директивы, обрабатываемые уже непосредственно процессором. При этом компилятор C#, JIT-компилятор и даже сам процессор могут оптимизировать ваш код. К примеру, после компиляции следующий нелепый метод в конечном итоге превратится в ничто:

```
private static void OptimizedAway() {  
    // Вычисленная во время компиляции константа равна нулю  
    Int32 value = (1 * 100) - (50 * 2);  
  
    // Если значение равно 0, цикл не выполняется  
    for (Int32 x = 0; x < value; x++) {  
        // Код цикла не нужно компилировать, так как он никогда не выполняется  
        Console.WriteLine("Jeff");  
    }  
}
```

В этом коде компилятор выясняет, что значение всегда равно 0, а значит, цикл никогда не будет выполнен и соответственно нет никакой нужды компилировать код внутри него. От метода в итоге может ничего не остаться. Далее, при компиляции метода, вызывающего метод `OptimizedAway`, JIT-компилятор попытается встроить туда код метода `OptimizedAway`, но так как этот код отсутствует, компилятор просто не будет вызывать данный метод. Разработчики очень любят это свойство компиляторов. Обычно код пытаются писать максимально осмысленно. Он должен быть простым для чтения, записи и редактирования. А затем компилятор переводит его на язык, понятный компьютеру. И мы хотим, чтобы компиляторы делали это максимально хорошо.

В процессе оптимизации кода компилятором C#, JIT-компилятором и процессором гарантируется сохранение его назначения. То есть с точки зрения одного потока метод делает то, зачем мы его написали, хотя способ реализации может отличаться от описанного в исходном коде. Однако при переходе к многопоточной конфигурации ситуация может измениться. Вот пример, в котором в результате оптимизации программа стала работать не так, как ожидалось:


```
internal static class StrangeBehavior {  
    // Далее вы увидите, что проблема решается.  
    // если сделать это поле волатильным  
    private static Boolean s_stopWorker = false;  
  
    public static void Main() {  
        Console.WriteLine("Main: letting worker run for 5 seconds");  
        Thread t = new Thread(Worker);  
        t.Start();  
        Thread.Sleep(5000);  
        s_stopWorker = true;  
        Console.WriteLine("Main: waiting for worker to stop");  
        t.Join();  
    }  
  
    private static void Worker(Object o) {  
        Int32 x = 0;  
        while (!s_stopWorker) x++;  
        Console.WriteLine("Worker: stopped when x={0}", x);  
    }  
}
```

Метод Main в этом фрагменте кода создает новый поток, исполняющий метод Worker, который считает по возрастающей, пока не получит команду остановиться. Метод Main позволяет потоку метода Worker работать 5 секунд, а затем останавливает его, присваивая статическому полю Boolean значение true. В этот момент поток метода Worker должен вывести результат счета, после чего он завершится. Метод Main ждет завершения метода Worker, вызывает метод Join, после чего поток метода Main возвращает управление, заставляя весь процесс прекратить работу.

Выглядит просто, не так ли? Но программа является потенциально проблемной в связи с ожидающей ее оптимизацией. При компиляции метода Worker компилятор обнаруживает, что переменная s_stopWorker может принимать значение true или false, но внутри метода это значение никогда не меняется. Поэтому компилятор может создать код, заранее проверяющий состояние переменной s_stopWorker. Если она имеет значение true, выводится результат "Worker: stopped when x=0". В противном случае компилятор создает код, входящий в бесконечный цикл и бесконечно увеличивающий значение переменной x. При этом оптимизация заставляет цикл работать крайне быстро, так как проверка переменной s_stopWorker осуществляется перед циклом, а проверки переменной на каждой итерации цикла не происходит.

Если вы хотите посмотреть, как это работает, поместите код в файл с расширением .cs и скомпилируйте его с использованием переключателей platform:x86 и /optimize+ языка C#. Запустите полученный исполняемый файл и вы убедитесь, что программа работает бесконечно. Обратите внимание, что вам нужен JIT-компилятор для платформы x86, который совершеннее компиляторов

x64 и IA64, а значит, обеспечивает более полную оптимизацию. Остальные JIT-компиляторы не выполняют оптимизацию столь тщательно, поэтому после их работы программа успешно завершится. Это подчеркивает еще один интересный аспект. Итоговое поведение вашей программы зависит от множества факторов, в частности от выбранной версии компилятора и используемых переключателей, от выбранного JIT-компилятора, от процессора, который будет выполнять код. Кроме того, программа не станет работать бесконечно, если запустить ее в отладчике, так как отладчик заставляет JIT-компилятор ограничиться неоптимизированным кодом, который проще поддается выполнению в пошаговом режиме.

Рассмотрим другой пример, в котором пара потоков осуществляет доступ к двум полям:

```
internal sealed class ThreadsSharingData {
    private Int32 m_flag = 0;
    private Int32 m_value = 0;

    // Этот метод исполняется одним потоком
    public void Thread1() {
        // ПРИМЕЧАНИЕ. Они могут выполняться в обратном порядке
        m_value = 5;
        m_flag = 1;
    }

    // Этот метод исполняется другим потоком
    public void Thread2() {
        // ПРИМЕЧАНИЕ. поле m_value может быть прочитано раньше, чем m_flag
        if (m_flag == 1)
            Console.WriteLine(m_value);
    }
}
```

В данном случае проблема в том, что компиляторы и процессор могут оттранслировать код таким образом, что две строки в методе Thread1 поменяются местами. Разумеется, это не изменит предназначения метода. Метод должен получить значение 5 в переменной m_value и значение 1 в переменной m_flag. С точки зрения однопоточного приложения порядок выполнения строк кода не имеет значения. Если же поменять указанные строки местами, другой поток, выполняющий метод Thread2, *может* обнаружить, что переменная m_flag имеет значение 1, и выведет значение 0.

Рассмотрим этот код с другой точки зрения. Предположим, что код метода Thread1 выполняется так, как *предусмотрено программой* (то есть так, как он написан). Обработывая код метода Thread2, компилятор должен сгенерировать код, читающий значения переменных m_flag и m_value из оперативной памяти в регистрах процессора. И возможно, что память первой выдаст значение переменной m_value, равное 0. Затем можно выполнить метод Thread1, меняющий

значение переменной `m_value` на 5, а переменной `m_flag` — на 1. Но регистр процессора метода `Thread2` не видит, что значение переменной `m_value` было изменено другим потоком на 5. После этого из оперативной памяти в регистре процессора может быть считано значение переменной `m_flag`, ставшее равным 1. В результате метод `Thread2` снова выведет значение 0.

Все эти захватывающие события, скорее всего, приведут к проблемам при окончательной компоновке программы, а не при компоновке на этапе отладки. В результате задача выявления проблемы и исправления кода становится не тривиальной. Поэтому сейчас давайте поговорим о том, как исправить код.

Класс `System.Threading.Thread` содержит три статических метода, которые выглядят следующим образом¹:

```
public sealed class Thread {  
    public static void VolatileWrite(ref Int32 address, Int32 value);  
    public static Int32 VolatileRead(ref Int32 address);  
    public static void MemoryBarrier();  
}
```

Это специальные методы. Они отключают оптимизацию, выполняемую компилятором C#, JIT-компилятором и собственно процессором. Вот как они работают:

- ❑ Метод `VolatileWrite` заставляет записать значение в параметр `address` непосредственно в момент обращения. *Более ранние* загрузки и сохранения программы должны происходить *до вызова* этого метода.
- ❑ Метод `VolatileRead` заставляет считать значение параметра `address` непосредственно в момент обращения. *Более поздние* загрузки и сохранения программы должны происходить *после вызова* этого метода.
- ❑ Метод `MemoryBarrier` не имеет доступа к памяти, но он заставляет все *более ранние* загрузки и сохранения программы завершиться *до своего вызова*. Все *более поздние* загрузки и сохранения программы завершаются *после вызова* данного метода. Метод `MemoryBarrier` используется не так часто, как первые два метода.

ВНИМАНИЕ

Чтобы не запутаться, приведу простое правило: при взаимодействии потоков друг с другом через общую память записывайте последнее значение методом `VolatileWrite`, а первое значение читайте методом `VolatileRead`.

Теперь при помощи указанных методов можно исправить класс `ThreadsSharingData`:

```
internal sealed class ThreadsSharingData {  
    private Int32 m_flag = 0;  
    private Int32 m_value = 0;
```

¹ Существуют также перегруженные версии методов `VolatileRead` и `VolatileWrite`, работающие с типами: `(S)Byte`, `(U)Int16`, `(U)Int32`, `(U)Int64`, `(U)IntPtr`, `Single`, `Double` и `Object`.

```
// Этот метод выполняется одним потоком
public void Thread1() {
    // ПРИМЕЧАНИЕ. 5 нужно записать в m_value до записи 1 в m_flag
    m_value = 5;
    Thread.VolatileWrite(ref m_flag, 1);
}

// Этот метод выполняется вторым потоком
public void Thread2() {
    // ПРИМЕЧАНИЕ. Поле m_value должно быть прочитано после m_flag
    if (Thread.VolatileRead(ref m_flag) == 1)
        Console.WriteLine(m_value);
}
}
```

Обращаю ваше внимание, что мы следуем правилу. Метод `Thread1` записывает два значения в поля, к которым имеют доступ несколько потоков. Последнее значение, которое мы хотим записать (присвоение переменной `m_flag` значения 1), записывается методом `VolatileWrite`. Метод `Thread2` читает оба значения из полей общего доступа, причем чтение первого из них (`m_flag`) выполняется методом `VolatileRead`.

Но что здесь происходит на самом деле? Для метода `Thread1` вызов метода `VolatileWrite` гарантирует, что все записи в переменные будут завершены до записи значения 1 в переменную `m_flag`. Так как операция `m_value = 5` расположена до вызова метода `VolatileWrite`, она сначала должна завершиться. Более того, значения скольких бы переменных ни редактировались до вызова метода `VolatileWrite`, все эти операции следует завершить до записи значения 1 в переменную `m_flag`. При этом все эти операции можно оптимизировать, сделав порядок их выполнения несущественным; главное, чтобы все они закончились до вызова метода `VolatileWrite`.

Вызов метода `VolatileRead` для метода `Thread2` гарантирует, что значения всех переменных будут прочитаны после значения переменной `m_flag`. Так как чтение переменной `m_value` происходит после вызова метода `VolatileRead`, оно должно осуществляться только после чтения значения переменной `m_flag`. То же самое касается чтения всех остальных переменных, расположенных после вызова метода `VolatileRead`. При этом операции чтения после метода `VolatileRead` можно оптимизировать, сделав порядок их выполнения несущественным; чтение просто станет невозможным до вызова метода `VolatileRead`.

Поддержка волатильных полей в C#

Как гарантировать, что программисты будут корректно вызывать методы `VolatileRead` и `VolatileWrite`? Сложно продумать все, в частности, представить, что именно могут делать с общими данными другие потоки в фоновом режиме. Чтобы упростить ситуацию, в C# появилось ключевое слово `volatile`, применяемое к статическим или экземплярным полям типов `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Char`, `Single` и `Boolean`. Также оно применяется к ссылочным типам

и любым перечислимым полям, если в основе последних лежит тип `Byte`, `SByte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Single` или `Boolean`. JIT-компилятор гарантирует, что доступ к полям, помеченным данным ключевым словом, будет происходить в режиме волатильного чтения или записи, поэтому в явном виде вызывать статический метод `VolatileRead` или `VolatileWrite` класса `Thread` больше не требуется. Более того, ключевое слово `volatile` запрещает компилятору C# и JIT-компилятору кэшировать содержимое поля в регистры процессора. Это гарантирует, что при всех операциях чтения и записи манипуляции будут производиться непосредственно с памятью.

Ключевое слово `volatile` позволяет переписать класс `ThreadsSharingData` следующим образом:

```
internal sealed class ThreadsSharingData {
    private volatile Int32 m_flag = 0;
    private Int32 m_value = 0;

    // Этот метод выполняется одним потоком
    public void Thread1() {
        // ПРИМЕЧАНИЕ. Значение 5 должно быть записано в m_value
        // перед записью 1 в m_flag
        m_value = 5;
        m_flag = 1;
    }

    // Этот метод выполняется другим потоком
    public void Thread2() {
        // ПРИМЕЧАНИЕ. Поле m_value должно быть прочитано после m_flag
        if (m_flag == 1)
            Console.WriteLine(m_value);
    }
}
```

Некоторые разработчики (в том числе я) не любят ключевое слово `volatile` и считают, что не стоило вводить его в C#. Мы считаем, что большинству алгоритмов не нужен одновременный доступ на чтение и запись поля со стороны нескольких потоков. А в большинстве оставшихся алгоритмов можно ограничиться обычным доступом к полю, повышающему производительность. А доступ к полю, помеченному как волатильное, требуется крайне редко. К примеру, сложно объяснить, как применить операцию волатильного чтения к такому вот алгоритму:

```
m_amount = m_amount + m_amount; // Предполагаем, что поле m_amount
                                // определено как волатильное
```

Обычно целое число может быть удвоено простым сдвигом всех битов на единицу влево, и многие компиляторы могут обработать такой код и выполнить указанную оптимизацию. Но если пометить поле `m_amount` ключевым словом

volatile, оптимизация станет невозможной. Компилятору придется создать код, читающий переменную `m_amount` из регистра, затем читающий ее еще раз из другого регистра, складывающий два значения и записывающий результат обратно в поле `m_amount`. Неоптимизированный код определенно занимает больше места и медленнее работает; вряд ли было бы уместно помещать такой код внутри цикла.

К тому же C# не поддерживает передачу волатильного поля по ссылке в метод. К примеру, если принадлежащее типу `Int32` волатильное поле `m_amount` попытается вызвать метод `Int32.TryParse`, компилятор сгенерирует предупреждение:

```
Boolean success = Int32.TryParse("123", out m_amount);  
// Эта строка приводит к сообщению от компилятора:  
// CS0420: ссылка на волатильное поле не будет трактоваться как волатильная
```

Взаимозапирающие конструкции

Как мы выяснили, метод `VolatileRead` класса `Thread` выполняет атомарную операцию чтения, а метод `VolatileWrite` этого же класса осуществляет атомарную операцию записи. То есть для чтения и записи существуют отдельные методы. В этом разделе мы поговорим о статических методах класса `System.Threading.Interlocked`. Каждый из этих методов выполняет как атомарное чтение, так и атомарную запись. Кроме того, все методы класса `Interlocked` ставят барьер в памяти. То есть любая запись переменной перед вызовом метода класса `Interlocked` выполняется до этого метода, а все чтения переменных после вызова метода выполняются после него.

Статические методы, работающие с переменными типа `Int32`, безоговорочно относятся к наиболее часто используемым. Продемонстрируем их:

```
public static class Interlocked {  
    // Возвращает (++location)  
    public static Int32 Increment(ref Int32 location);  
  
    // Возвращает (--location)  
    public static Int32 Decrement(ref Int32 location);  
  
    // Возвращает (location1 += value)  
    // ПРИМЕЧАНИЕ. Значение может быть отрицательным,  
    // что позволяет выполнить вычитание  
    public static Int32 Add(ref Int32 location1, Int32 value);  
  
    // Int32 old = location1; location1 = value; возвращает old;  
    public static Int32 Exchange(ref Int32 location1, Int32 value);  
  
    // Int32 old = location1;
```

продолжение ➤

```
// если (location1 == comparand) location1 = value;
// возвращает old;
public static Int32 CompareExchange(ref Int32 location1,
    Int32 value, Int32 comparand);
...
}
```

Существуют и перегруженные версии этих методов, работающие со значениями типа Int64. Кроме того, в классе Interlocked существуют методы Exchange и CompareExchange, принимающие параметры Object, IntPtr, Single и Double. Есть и обобщенная версия, в которой обобщенный тип ограничен типом class (любой ссылочный тип).

Лично мне очень нравятся методы взаимного записания, потому что они работают относительно быстро и позволяют многого добиться. Давайте рассмотрим код, использующий данные методы для асинхронного запроса данных с различных веб-серверов. Это короткий код, не блокирующий никаких потоков, автоматически масштабирующий потоки пула и использующий все доступные процессоры, если их загрузка может пойти ему на пользу. Кроме того, количество серверов, на которые код в исходном виде поддерживает доступ, достигает 2 147 483 647 (Int32.MaxValue). Другими словами, это превосходная основа для написания собственных сценариев.

```
internal sealed class MultiWebRequests {
    // Этот класс Helper координирует все асинхронные операции
    private AsyncCoordinator m_ac = new AsyncCoordinator();

    // Набор веб-серверов, к которым будут посылаться запросы
    private WebRequest[] m_requests = new WebRequest[] {
        WebRequest.Create("http://wintellect.com/"),
        WebRequest.Create("http://Microsoft.com/")
    };

    // Создание массива ответов: по одному ответу на каждый запрос
    private WebResponse[] m_results = new WebResponse[2];

    public MultiWebRequests(Int32 timeout = Timeout.Infinite) {
        // Одновременная асинхронная инициализация всех запросов
        for (Int32 n = 0; n < m_requests.Length; n++) {
            m_ac.AboutToBegin(1);
            m_requests[n].BeginGetResponse(EndGetResponse, n);
        }

        // Сообщаем классу Helper, что все операции инициализированы,
        // вызываем AllDone после завершения всех операций
        // или вызывается метод Cancel, или заканчивается время ожидания
        m_ac.AllBegin(AllDone, timeout);
    }
}
```

```
// Вызов этого метода показывает, что результат не имеет значения
public void Cancel() { m_ac.Cancel(); }

// Этот метод вызывается при ответе каждого сервера
private void EndGetResponse(IAsyncResult result) {
    // Получение индекса запроса
    Int32 n = (Int32)result.AsyncState;

    // Сохраняем ответ под тем же самым индексом
    m_results[n] = m_requests[n].EndGetResponse(result);

    // Сообщаем классу Helper, что сервер ответил
    m_ac.JustEnded();
}

// Этот метод вызывается после получения ответа от всех серверов,
// или вызывается метод Cancel, или заканчивается время ожидания
private void AllDone(CoordinationStatus status) {
    switch (status) {
        case CoordinationStatus.Cancel:
            Console.WriteLine("The operation was canceled"); break;

        case CoordinationStatus.Timeout:
            Console.WriteLine("The operation timed-out"); break;

        case CoordinationStatus.AllDone:
            Console.WriteLine("Here are the results from all the Web servers");
            for (Int32 n = 0; n < m_requests.Length; n++) {
                Console.WriteLine("{0} returned {1} bytes.",
                    m_results[n].ResponseUri, m_results[n].ContentLength);
            }
            break;
    }
}
```

Этот код непосредственно не задействует методы взаимного записания, так как весь координирующий код инкапсулирован в класс `AsyncCoordinator`, предназначенный для многократного использования. В процессе конструирования класс `MultiWebRequest` инициализирует класс `AsyncCoordinator`, массив объектов `WebRequest` и массив объектов `WebResponse`. Затем он асинхронно выполняет все веб-запросы, вызвав метод `BeginGetResponse`. Но перед каждым запросом вызывается метод `AboutToBegin` класса `AsyncCoordinator`, которому передается количество запланированных запросов¹.

¹ Код будет работать корректно, даже если вызвать метод `m_ac>AboutToBegin(m_requests.Length)` всего один раз перед циклом вместо вызова метода `AboutToBegin` внутри цикла.

После завершения всех запросов к веб-серверам методу `AllBegun` класса `AsyncCoordinator` будет передано, во-первых, имя метода, который следует запустить после выполнения всех операций (`AllDone`), а во-вторых, время задержки. После ответа каждого сервера различные потоки пула будут вызывать метод `EndGetResponse` класса `MultiWebRequests`. Этот метод выясняет, какой из запросов он обрабатывает (при помощи свойства `AsyncState` класса `IAAsyncResult`), а затем сохраняет объект `WebResponse` в массиве `m_results`. После сохранения каждого результата вызывается метод `JustEnded` класса `AsyncCoordinator`, позволяющий объекту `AsyncCoordinator` узнать о завершении операции.

После завершения всех операций объект `AsyncCoordinator` вызывает метод `AllDone` для обработки результатов, полученных со всех веб-серверов. Этот метод будет выполняться потоком пула, последним получившим ответ с сервера. В случае завершения времени ожидания или отмены операции метод `AllDone` будет вызван либо потоком пула, уведомляющим объект `AsyncCoordinator` о том, что время закончилось, либо потоком, вызвавшим метод `Cancel`. Существует также вероятность, что поток, выполняющий запрос к серверу, сам вызовет метод `AllDone`, если последний запрос завершится до вызова метода `AllBegin`.

Имейте в виду, что в данном случае имеет место ситуация гонки, так как возможно одновременное завершение всех запросов к серверам, вызов метода `AllBegun`, завершение времени ожидания и вызов метода `Cancel`. Если такое произойдет, объект `AsyncCoordinator` выберет победителя, гарантируя, что метод `AllDone` будет вызван не более одного раза. Победитель указывается передачей в метод `AllDone` аргумента состояния, роль которого может играть один из символов, определенных в типе `CoordinationStatus`:

```
internal enum CoordinationStatus { AllDone, Timeout, Cancel };
```

Теперь, когда вы получили представление о том, что происходит, посмотрим, как это работает. Класс `AsyncCoordinator` содержит всю логику координации потоков. Во всех случаях он использует методы `Interlocked`, гарантируя быстрое выполнение кода и отсутствие блокировки потоков. Вот код для этого класса:

```
internal sealed class AsyncCoordinator {
    private Int32 m_opCount = 1;           // Уменьшается на 1 методом AllBegun
    private Int32 m_statusReported = 0;    // 0=false, 1=true
    private Action<CoordinationStatus> m_callback;
    private Timer m_timer;

    // Этот метод СЛЕДУЕТ вызвать ДО метода BeginXxx
    public void AboutToBegin(Int32 opsToAdd = 1) {
        Interlocked.Add(ref m_opCount, opsToAdd);
    }

    // Этот метод СЛЕДУЕТ вызвать ПОСЛЕ метода EndXxx
    public void JustEnded() {
        if (Interlocked.Decrement(ref m_opCount) == 0)
            ReportStatus(CoordinationStatus.AllDone);
    }
}
```

```
}

// Этот метод СЛЕДУЕТ вызывать ПОСЛЕ вызова ВСЕХ методов BeginXxx
public void AllBegan(Action<CoordinationStatus> callback,
    Int32 timeout = Timeout.Infinite) {

    m_callback = callback;
    if (timeout != Timeout.Infinite)
        m_timer = new Timer(TimeExpired, null, timeout, Timeout.Infinite);
    JustEnded();
}

private void TimeExpired(Object o) {
    ReportStatus(CoordinationStatus.Timeout);
}

public void Cancel() { ReportStatus(CoordinationStatus.Cancel); }

private void ReportStatus(CoordinationStatus status) {
    // Если состояние ни разу не передавалось, передайте его.
    // в противном случае игнорируйте его
    if (Interlocked.Exchange(ref m_statusReported, 1) == 0)
        m_callback(status);
}
}
```

Самым важным в этом классе является поле `m_opCount`. Оно отслеживает количество асинхронных операций, ожидающих выполнения. Перед началом каждой такой операции вызывается метод `AboutToBegin`. Он вызывает метод `Interlocked.Add`, чтобы атомарно добавить к полю `m_opCount` переданное в него число. Операция суммирования должна осуществляться атомарно, так как веб-серверы могут отвечать потокам пула в процессе начала дополнительных операций. При каждом ответе сервера вызывается метод `JustEnded`. Он вызывает метод `Interlocked.Decrement` и атомарно вычитает из переменной `m_opCount` единицу. Поток, присвоивший переменной `m_opCount` значение 0, вызывает метод `ReportStatus`.

ПРИМЕЧАНИЕ

Полю `m_opCount` присваивается начальное значение 1 (не 0); это крайне важно, так как гарантирует, что метод `AllDone` не будет вызван во время запроса к серверу потоком, исполняющим метод конструктора. До вызова конструктором метода `AllBegan` переменная `m_opCount` не может получить значение 0. Вызванный же конструктором метод `AllBegan`, в свою очередь, вызывает метод `JustEnded`, который последовательно уменьшает значение переменной `m_opCount` на 1 и фактически отменяет эффект присвоения ей начального значения 1. В результате переменная `m_opCount` может достичь значения 0, но только после того как мы получим информацию об отправке всех запросов к веб-серверам.

Метод `ReportStatus` является судьей в гонках, которые могут возникнуть между завершающимися операциями, истечением времени ожидания и вызовом метода `Cancel`. Он должен убедиться, что только одно из условий рассматривается в качестве победителя, и метод `m_callback` будет вызван всего один раз. Выбор победителя осуществляется передачей методу `Interlocked.Exchange` ссылки на поле `m_statusReported`. Это поле рассматривается как переменная типа `Boolean`; но на самом деле подобное невозможно, так как методы класса `Interlocked` не принимают переменных типа `Boolean`. Поэтому мы используем переменную типа `Int32`, значение `0` которой является эквивалентом `false`, а значение `1` — эквивалентом `true`.

Внутри метода `ReportStatus` вызов метода `Interlocked.Exchange` меняет значение переменной `m_statusReported` на `1`. Но только первый проделавший это поток увидит, как метод `Interlocked.Exchange` возвращает значение `0`, и только он активизирует метод обратного вызова. Все остальные потоки, вызвавшие метод `Interlocked.Exchange`, получают значение `1`, по сути, уведомляющее их, что метод обратного вызова уже активизирован и больше этого делать не нужно.

Реализация простого записывания с заикливанием

Методы взаимного записывания прекрасно работают, но в основном со значениями типа `Int32`. А что делать, если возникла необходимость атомарного манипулирования набором полей объекта? Нам потребуется предотвратить проникновение всех потоков кроме одного в область кода, управляющую полями. Методы взаимного записывания позволяют выполнить записывание в рамках синхронизации потоков:

```
internal struct SimpleSpinLock {
    private Int32 m_ResourceInUse; // 0=false (по умолчанию), 1=true

    public void Enter() {
        // Указываем, что ресурс используется, и если этот поток
        // переводит его из свободного состояния, возвращаем управление
        while (Interlocked.Exchange(ref m_ResourceInUse, 1) != 0) {
            /* Здесь что-то происходит... */
        }
    }

    public void Leave() {
        // Помечаем ресурс, как свободный
        Thread.VolatileWrite(ref m_ResourceInUse, 0);
    }
}
```

А вот класс, демонстрирующий использование метода `SimpleSpinLock`:

```
public sealed class SomeResource {
    private SimpleSpinLock m_sl = new SimpleSpinLock();
```

```
public void AccessResource() {  
    m_sl.Enter();  
    // Доступ к ресурсу в каждый момент времени имеет только один поток...  
    m_sl.Leave();  
}  
}
```

Реализация метода `SimpleSpinLock` весьма проста. Если два потока одновременно вызывают метод `Enter`, метод `Interlocked.Exchange` гарантирует, что один поток изменит значение переменной `m_resourceInUse` с 0 на 1. Когда он видит, что переменная `m_resourceInUse` равна 0, он заставляет метод `Enter` возвратить управление, чтобы продолжить выполнение кода метода `AccessResource`. Второй поток тоже попытается заменить значение переменной `m_resourceInUse` на 1. Но этот поток увидит, что переменная уже не равна 0, и заикнется, начав непрерывно вызывать метод `Exchange` до тех пор, пока первый поток не вызовет метод `Leave`.

После того как первый поток завершит манипуляции полями объекта `SomeResource`, он вызовет метод `Leave`, который, в свою очередь, вызовет метод `Thread.VolatileWrite` и вернет переменной `m_resourceInUse` значение 0. Это заставит заикнувшийся поток поменять значение переменной `m_resourceInUse` с 0 на 1 и, наконец, получить управление от метода `Enter`, предоставляя последнему доступ к полям объекта `SomeResource`.

Итак, все готово. Это простая реализация записи в рамках синхронизации потоков. Правда, ее серьезная потенциальная проблема состоит в том, что при наличии конкуренции за право на запись потоки вынуждены ожидать записи в цикле, и это заикливание приводит к пустому расходованию бесценного процессорного времени. Соответственно, запись с заикливанием имеет смысл использовать только для защиты очень быстро выполняемого кода.

Запись с заикливанием обычно не применяется на машинах с одним процессором, так как запертый поток не сможет быстро отпереться, если другой поток, претендующий на запись, заикнется. Ситуация осложняется, если запертый поток имеет более низкий приоритет, чем поток, претендующий на запись. Низкоприоритетный поток может вообще не получить шансов на выполнение, то есть просто зависнуть. Windows иногда на короткое время динамически повышает приоритет потоков. Для потоков, использующих запись с заикливанием, данный режим следует отключить. Это делается при помощи свойств `PriorityBoostEnabled` классов `System.Diagnostics.Process` и `System.Diagnostics.ProcessThread`. Запись с заикливанием на гиперпоточных машинах также связано с проблемами. Для их решения код записи с заикливанием часто наделяется дополнительной логикой. Однако я не хотел бы вдаваться в детали, так как эта логика довольно быстро меняется. Могу сказать только, что FCL поставляется вместе со структурой `System.Threading.SpinWait`, которая включает в себя всю необходимую логику.

В FCL существует также структура `System.Threading.SpinLock`, сходная с показанным ранее классом `SimpleSpinLock`. Она отличается использованием структуры `SpinWait` с целью повышения производительности. Структура `SpinLock` поддерживает время ожидания. Интересно отметить, что обе структуры — моя `SimpleSpinLock` и `SpinLock` в FCL — относятся к значимым типам. То есть они являются облегченными, легко запоминающимися объектами. Перечислением `SpinLock` имеет смысл пользоваться, если вам нужно, к примеру, связать записание с каждым элементом коллекции. Но при этом нужно следить за тем, чтобы экземпляры `SpinLock` никуда не передавались, потому что они при этом копируются, из-за чего вся синхронизация сойдет «на нет». Экземплярные поля `SpinLock` не следует помечать, как предназначенные только для чтения, поскольку при манипуляциях с записанием их внутреннее значение должно меняться.

Задержка в исполнении потока

Хитрость состоит в том, чтобы иметь поток, умеющий заставить ресурс на время приостановить исполнение этого потока, чтобы другой поток, обладающий в данный момент ресурсом, завершился и освободил место. Для этого структура `SpinWait` вызывает методы `Sleep`, `Yield` и `SpinWait` класса `Thread`. Коротко опишем данные методы.

Поток может сообщить системе, что в течение некоторого времени его не нужно планировать на исполнение. Эта задача решается статическим методом `Sleep`:

```
public static void Sleep(Int32 millisecondsTimeout);  
public static void Sleep(TimeSpan timeout);
```

Поток заставляет метод приостановить работу на указанное время. Вызов метода `Sleep` позволяет потоку добровольно убрать напоминание о времени своего исполнения. Система забирает поток у планировщика *примерно* на указанное время. То есть если вы говорите системе, что метод хочет приостановить работу на 100 мс, он будет приостановлен примерно на это время, но возможно пробудет в состоянии покоя на несколько секунд меньше или больше. Надеюсь, вы помните, что Windows не является операционной системой реального времени. Поток, скорее всего, пробудится в указанное время, но по большому счету время его пробуждения зависит от остальных происходящих в системе процессов.

Можно передать параметру `millisecondsTimeout` метода `Sleep` значение `System.Threading.Timeout.Infinite` (определенное как `-1`). В результате поток окажется заблокированным на неограниченное время. При этом он будет существовать, и вы в любой момент сможете восстановить его стек и ядро. Передача в метод `Sleep` значения `0` скажет системе, что вызывающий поток освобождает ее от его исполнения и заставляет запланировать другой поток. Впрочем, система при отсутствии доступных для планирования потоков такого же или более высокого приоритета может снова запланировать исполнение потока, только что вызвавшего метод `Sleep`.

Поток может также попросить Windows запланировать для текущего процессора другой поток, вызвав метод `Yield` класса `Thread`:

```
public static Boolean Yield();
```

При наличии другого потока, готового работать на данном процессоре, метод возвращает значение `true`, время жизни вызвавшего его потока считается завершенным, а в течение одного такта используется другой выбранный поток. После этого вызвавший метод `Yield` поток снова попадает в расписание и начинает работать в течение следующего такта. При отсутствии потоков, на которые можно переключиться, метод `Yield` возвращает значение `false`, и вызвавший его поток продолжает исполняться.

Метод `Yield` дает шанс исполнить ожидающие своего процессорного времени потоки равного или более низкого приоритета. Поток вызывает данный метод, если ему требуется ресурс, которым в настоящее время владеет другой поток. Он *надеется* на то, что Windows поставит обладающий ресурсом в данный момент поток в очередь планировщика, освободив тем самым доступ. В результате, когда вызвавший метод `Yield` поток снова начнет исполняться, доступ к ресурсу может получить уже он.

Существует выбор между вызовом методов `Thread.Sleep(0)` и `Thread.Sleep(1)`. В первом случае потокам с низким приоритетом не дают исполняться, в то время как метод `Thread.Sleep(1)` включает принудительное переключение контекста, и Windows погружает поток в спящее состояние более чем на 1 мс, что обусловлено разрешением внутреннего системного таймера.

Гиперпотоковые процессоры могут оставить в каждый момент времени один поток. И когда отсчеты начинают совершаться в цикле, нужно принудительно остановить текущий поток, позволив исполняться другому. Поток может остановиться сам, дав гиперпотоковому процессору возможность переключиться на другие потоки. Для этого он вызывает метод `SpinWait` класса `Thread`:

```
public static void SpinWait(Int32 iterations);
```

Вызывая этот метод, вы фактически выполняете специальную инструкцию процессора. Она не заставляет Windows делать какую-либо работу (операционная система уверена, что она уже запланировала для процессора два потока). В случае если гиперпотоковый процессор не используется, эта специальная инструкция просто игнорируется.

ПРИМЕЧАНИЕ

Чтобы лучше познакомиться с данными методами, почитайте про их Win32-эквиваленты: `Sleep`, `SwitchToThread` и `YieldProcessor`. Дополнительные сведения о настройке разрешения системного таймера вы получите при знакомстве с Win32-функциями `timeBeginPeriod` и `timeEndPeriod`.

Универсальный эталон взаимного записи

Многие пользователи, познакомившись с методами взаимного записи, удивляются, почему специалисты Microsoft не разработали дополнительных методов подобного рода, подходящих для большего количества сценариев. К примеру, в классе `Interlocked` были бы полезны методы `Multiple`, `Divide`, `Minimum`, `Maximum`, `And`, `Or`, `Xor` и многие другие. Однако вместо этих методов можно использовать хорошо известный эталон, позволяющий методом `Interlocked.CompareExchange` атомарно выполнять любые операции со значениями типа `Int32`. А так как существуют перегруженные версии этого метода для типов `Int64`, `Single`, `Double`, `Object`, а также для обобщенного ссылочного типа, эталон может работать и со всеми этими типами.

Вот пример создания на основе эталона атомарного метода `Maximum`:

```
public static Int32 Maximum(ref Int32 target, Int32 value) {
    Int32 currentVal = target, startVal, desiredVal;

    // Параметр target может использоваться другим потоком.
    // его трогать не стоит
    do {
        // Запись начального значения этой итерации
        startVal = currentVal;

        // Вычисление желаемого значения в терминах startVal и value
        desiredVal = Math.Max(startVal, value);

        // ПРИМЕЧАНИЕ. Здесь поток может быть прерван!

        // if (target == startVal) target = desiredVal
        // Возвращение значения, предшествующего потенциальным изменениям
        currentVal = Interlocked.CompareExchange(
            ref target, desiredVal, startVal);

        // Если начальное значение на этой итерации изменилось, повторить
    } while (startVal != currentVal);

    // Возвращаем максимальное значение, когда поток пытается его присвоить
    return desiredVal;
}
```

Давайте посмотрим, что здесь происходит. В момент, когда метод начинает выполняться, переменная `currentVal` инициализируется значением параметра `target`. Затем внутри цикла то же самое начальное значение получает переменная `startVal`. При помощи этой последней переменной вы можете выполнять любые нужные вам операции. Они могут быть крайне сложными и состоять из тысяч строк кода. Но в итоге вы получите результат, который помещается в переменную `desiredVal`. В моем примере просто сравниваются переменные `startVal` и `value`.

Пока операция выполняется, значение параметра `target` может поменять другой поток. Вряд ли такое случится, но в принципе подобная ситуация возможна. Если это произойдет, значение переменной `derivedVal` окажется основанным на старом значении переменной `startVal`, а не на текущем значении параметра `target`, а следовательно, мы не должны менять этот параметр. Гарантировать, что значение параметра `target` поменяется на значение переменной `desiredVal` при условии, что никакой другой поток не поменяет его за спиной нашего потока, можно с помощью метода `Interlocked.CompareExchange`. Он проверяет, совпадает ли значение параметра `target` со значением переменной `startVal` (а именно его мы предполагаем у параметра `target` перед началом выполнения операции). Если значение параметра `target` не поменялось, метод `CompareExchange` заменяет его новым значением переменной `desiredVal`. Если же изменения произошли, метод `CompareExchange` не трогает параметр `target`.

Метод `CompareExchange` возвращает значение параметра `target` на момент своего вызова, которое мы помещаем в переменную `currentVal`. Затем мы сравниваем переменную `startVal` с новым значением переменной `currentVal`. В случае совпадения поток не меняет параметр `target` за нашей спиной, этот параметр содержит значение переменной `desiredVal`, цикл `while` прекращает свою работу, и метод возвращает управление. Если же значения не совпадают, значит, другой поток поменял значение параметра `target`, поэтому параметру не было присвоено значение переменной `desiredVal`, цикл переходит к следующей итерации и пробует снова выполнить операцию на этот раз с новым значением переменной `currentVal`, отражающей изменения, внесенные посторонним потоком.

Лично я использовал данный эталон очень часто и даже создал инкапсулировавший его обобщенный метод `Morph`!

```
delegate Int32 Morpher<TResult, TArgument>(
    Int32 startValue, TArgument argument,
    out TResult morphResult);

static TResult Morph<TResult, TArgument>(
    ref Int32 target, TArgument argument,
    Morpher<TResult, TArgument> morpher) {

    TResult morphResult;
    Int32 currentVal = target, startVal, desiredVal;
    do {
        startVal = currentVal;
        desiredVal = morpher(startVal, argument, out morphResult);
        currentVal = Interlocked.CompareExchange(
            ref target, desiredVal, startVal);
    } while (startVal != currentVal);
    return morphResult;
}
```

¹ Очевидно, что из-за метода обратного вызова `morpher` метод `Morph` хуже в плане производительности. Чтобы исправить ситуацию, сделайте код подставляемым (`inline`), как в примере `Maximum`.

Конструкции режима ядра

Для синхронизации потоков в Windows существует несколько конструкций режима ядра. Они работают намного медленнее конструкций пользовательского режима, так как требуют координации со стороны операционной системы. Кроме того, каждый метод обращается к ядру, заставляя вызывающий поток перейти из управляемого в машинный код, а затем и в код режима ядра, после чего возвращается назад. Такие переходы требуют много процессорного времени и их частое выполнение значительно снижает производительность приложения.

Впрочем, у конструкций режима ядра есть и ряд преимуществ:

- ❑ Если конструкция режима ядра выявляет конкуренцию за ресурс, Windows блокирует проигравший поток, останавливая заикливание, которое ведет к напрасной трате ресурсов процессора.
- ❑ Конструкции режима ядра могут синхронизировать дуг с другом машинные и управляемые потоки.
- ❑ Конструкции режима ядра умеют синхронизировать потоки различных процессов, запущенных на одной машине.
- ❑ Конструкции режима ядра можно наделить защитой, запретив доступ со стороны неавторизованных учетных записей.
- ❑ Поток можно заблокировать, пока не станут доступны все конструкции режима ядра или пока не станет доступна хотя бы одна такая конструкция.
- ❑ Поток можно заблокировать конструкцией режима ядра, указав время ожидания; если за указанное время поток не получит доступа к нужному ему ресурсу, он будет разблокирован и сможет выполнять другие задания.

К примитивным конструкциям синхронизации потоков в режиме ядра относятся *события* (events) и *семафоры* (semaphores). На их основе строятся более сложные конструкции аналогичного назначения, например *мьютексы* (mutex). Более полную информацию о них вы найдете в моей книге «Windows via C/C++» (Microsoft Press, 2007).

В пространстве имен System.Threading существует абстрактный базовый класс WaitHandle. Он играет роль оболочки для дескриптора ядра Windows. В FCL имеется несколько производных от него классов. Все они определены в пространстве имен System.Threading и реализуются библиотекой MSCorLib.dll. Исключением является класс Semaphore, реализованный в библиотеке System.dll. Вот как выглядит иерархия этих классов:

```
WaitHandle
  EventWaitHandle
    AutoResetEvent
    ManualResetEvent
Semaphore
Mutex
```

В базовом классе `WaitHandle` имеется поле `SafeWaitHandle`, содержащее дескриптор ядра `Win32`. Это поле инициализируется в момент создания класса, производного от `WaitHandle`. Кроме того, класс `WaitHandle` открыто предлагает методы, которые наследуются всеми производными классами. Каждый из вызываемых конструкциями режима ядра методов обеспечивает полную защиту памяти. Вот наиболее интересные открытые методы класса `WaitHandle` (перегруженные версии некоторых методов не показаны):

```
public abstract class WaitHandle : MarshalByRefObject, IDisposable {
    // Close и Dispose вызывают Win32-функцию CloseHandle
    public virtual void Close();
    public void Dispose();

    // WaitOne вызывает Win32-функцию WaitForSingleObjectEx
    public virtual Boolean WaitOne();
    public virtual Boolean WaitOne(Int32 millisecondsTimeout);

    // WaitAny вызывает Win32-функцию WaitForMultipleObjectsEx
    public static Int32 WaitAny(WaitHandle[] waitHandles);
    public static Int32 WaitAny(
        WaitHandle[] waitHandles, Int32 millisecondsTimeout);

    // WaitAll вызывает Win32-функцию WaitForMultipleObjectsEx
    public static Boolean WaitAll(WaitHandle[] waitHandles);
    public static Boolean WaitAll(
        WaitHandle[] waitHandles, Int32 millisecondsTimeout);

    // SignalAndWait вызывает Win32-функцию SignalObjectAndWait
    public static Boolean SignalAndWait(
        WaitHandle toSignal, WaitHandle toWaitOn);
    public static Boolean SignalAndWait(
        WaitHandle toSignal, WaitHandle toWaitOn,
        Int32 millisecondsTimeout, Boolean exitContext)

    // Используйте для доступа к необработанному Win32-дескриптору
    public SafeWaitHandle SafeWaitHandle { get; set; }

    // Возвращает управление от WaitAny после истечения времени ожидания
    public const Int32 WaitTimeout = 0x102;
}
```

Здесь следует сделать несколько замечаний:

- ❑ Метод `Close` класса `WaitHandle` (или непараметрический метод `Dispose` интерфейса `IDisposable`) закрывает дескриптор базового ядра. Оба эти метода вызывают `Win32-функцию CloseHandle`.
- ❑ Метод `WaitOne` класса `WaitHandle` блокирует текущий поток до получения сигнала объектом ядра. Он вызывает `Win32-функцию WaitForSingleObjectEx`.

Значение `true` возвращается, если объект получил сигнал. Если же время ожидания истекло, возвращается значение `false`.

- ❑ Статический метод `WaitAny` класса `WaitHandle` заставляет вызывающий поток ждать получения сигнала одним из объектов ядра, указанным в массиве `WaitHandle[]`. Возвращенное значение типа `Int32` является индексом элемента массива, получившего сигнал. Если в процессе ожидания сигнала не поступило, возвращается значение `WaitHandle.WaitTimeout`. Данный метод вызывает Win32-функцию `WaitForMultipleObjectsEx`, передавая параметру `bWaitAll` значение `FALSE`.
- ❑ Статический метод `WaitAll` класса `WaitHandle` заставляет вызывающий поток ждать получения сигнала всеми объектами ядра, указанными в массиве `WaitHandle[]`. Если сигналы благополучно получены, возвращается значение `true`, в случае же окончания времени ожидания возвращается значение `false`. Данный метод вызывает Win32-функцию `WaitForMultipleObjectsEx`, передавая параметру `bWaitAll` значение `TRUE`.
- ❑ Массив, передаваемый методам `WaitAny` и `WaitAll`, должен содержать не более 64 элементов. В противном случае метод вбрасывает исключение `System.NotSupportedException`.
- ❑ Статический метод `SignalAndWait` класса `WaitHandle` атомарно сигнализирует одному ядру и ждет получения сигнала вторым. Если объект получил сигнал, возвращается значение `true`, в случае же истечения времени ожидания возвращается значение `false`. Данный метод вызывает Win32-функцию `SignalObjectAndWait`.

ПРИМЕЧАНИЕ

В некоторых случаях при блокировке потока из однопоточного отделения (`apartment`) возможно пробуждение потока для отправки сообщений. Например, заблокированный поток может проснуться для обработки Windows-сообщения, отправленного другим потоком. Это делается для совместимости с моделью СОМ. Для большинства приложений это не проблема. Но если ваш код в процессе обработки сообщения запрет другой поток, может случиться клинч. Как вы увидите в главе 29, все гибридные конструкции записывания тоже могут вызывать данные методы, так что вышесказанное верно и для них.

Версии методов `WaitOne`, `WaitAll` и `SignalAndWait`, не принимающие параметр `timeout`, должны иметь прототип как возвращающие значение типа `void`, а не `Boolean`. В противном случае методы бы всегда возвращали значение `true` из-за предполагаемого бесконечного времени ожидания (`System.Threading.Timeout.Infinite`). Так что при вызове любого из этих методов нет нужды проверять возвращаемое им значение.

Как уже упоминалось, классы `AutoResetEvent`, `ManualResetEvent`, `Semaphore` и `Mutex` являются производными от класса `WaitHandle`, то есть наследуют методы этого класса и их поведение. Впрочем, эти классы обладают и собственными методами, о которых мы сейчас и поговорим.

Во-первых, конструкторы всех этих классов вызывают Win32-функцию `CreateEvent` (передавая параметру `bManualReset` значение `FALSE`), `CreateEvent` (передавая параметру `bManualReset` значение `TRUE`), `CreateSemaphore` или `CreateMutex`. Значение дескриптора, возвращаемого при таких вызовах, сохраняется в закрытом поле `SafeWaitHandle`, определенном в базовом классе `WaitHandle`.

Во-вторых, классы `EventWaitHandle`, `Semaphore` и `Mutex` предлагают статические методы `OpenExisting`, вызывающие Win32-функцию `OpenEvent`, `OpenSemaphore` или `OpenMutex`, передавая ей аргумент типа `String` с именем существующего ядра. Значение дескриптора, возвращаемого при таких вызовах, сохраняется во вновь созданном объекте, возвращаемым методом `OpenExisting`. При отсутствии ядра с указанным именем вбрасывается исключение `WaitHandleCannotBeOpenedException`.

Конструкции режима ядра предназначены для создания приложений, позволяющих в каждый момент времени выполняться только одной своей копии. Примерами таких приложений являются `Microsoft Office Outlook`, `Windows Live Messenger`, `Windows Media Player` `Windows Media Center`. Вот как реализовать такое приложение:

```
using System;
```

```
using System.Threading;
```

```
public static class Program {  
    public static void Main() {  
        Boolean createdNew;  
  
        // Пытаемся создать объект ядра с указанным именем  
        using (new Semaphore(0, 1, "SomeUniqueStringIdentifyingMyApp",  
            out createdNew)) {  
            if (createdNew) {  
                // Этот поток создает ядро, так что другие копии приложения  
                // не могут запускаться. Выполняем остальную часть приложения...  
            } else {  
                // Этот поток открывает существующее ядро с тем же именем;  
                // должна запуститься другая копия приложения.  
                // Ничего не делаем, ждем возвращения управления от метода Main,  
                // чтобы завершить вторую копию приложения  
            }  
        }  
    }  
}
```

В этом фрагменте кода фигурирует класс `Semaphore`, но с таким же успехом можно было воспользоваться классом `EventWaitHandle` или `Mutex`, так как пред-

лагаемое объектом поведение не требует синхронизации потока. Однако я использую преимущество такого поведения при создании объектов ядра. Давайте посмотрим, как работает показанный код. Представим, что две копии процесса запустились одновременно. Каждому процессу соответствует его собственный поток, и оба потока попытаются создать объект Semaphore с одним и тем же именем (в моем примере SomeUniqueStringIdentifyingMyApp). Ядро Windows гарантирует создание объекта ядра с указанным именем только одним потоком; переменной `createdNew` этого потока будет присвоено значение `true`.

В случае со вторым потоком Windows обнаруживает, что объект ядра с указанным именем уже существует; соответственно, потоку не позволено создать еще один объект. Впрочем, продолжив работу, этот поток может получить доступ к тому же объекту ядра, что и поток первого процесса. Таким способом потоки из различных процессов взаимодействуют друг с другом через единое ядро. Но в данном случае поток второго процесса видит, что его переменной `createdNew` присвоено значение `false`. Таким образом он узнает о том, что первая копия процесса запущена, поэтому вторая копия немедленно завершает свою работу.

События

Событиями (events) называются переменные типа `Boolean`, управляемые ядром. Ожидающий события поток блокируется, если оно имеет значение `false`, и освобождается в случае значения `true`. Существует два вида событий. Когда событие с автосбросом имеет значение `true`, оно освобождает всего один заблокированный поток, так как после освобождения первого потока ядро автоматически возвращает событию значение `false`. Если же значение `true` имеет событие с ручным сбросом, оно освобождает все ожидающие этого потоки, так как в данном случае ядро не присваивает ему значение `false` автоматически, в коде это должно быть сделано в явном виде. Вот как выглядят классы, связанные с событиями:

```
public class EventWaitHandle : WaitHandle {
    public Boolean Set();    // Boolean присваивается true;
                           // всегда возвращает true
    public Boolean Reset(); // Boolean присваивается false;
                           // всегда возвращает true
}

public sealed class AutoResetEvent : EventWaitHandle {
    public AutoResetEvent(Boolean initialState);
}

public sealed class ManualResetEvent : EventWaitHandle {
    public ManualResetEvent(Boolean initialState);
}
```

С помощью события с автосбросом можно легко реализовать записание в рамках синхронизации потоков, поведение которого сходно с поведением ранее показанного класса SimpleSpinLock:

```
internal sealed class SimpleWaitLock : IDisposable {
    private AutoResetEvent m_ResourceFree = new AutoResetEvent(true);
                                                // Изначально свободно

    public void Enter() {
        // Блокируем в ядре ресурсы, которые должны быть свободны,
        // и возвращаем управление
        m_ResourceFree.WaitOne();
    }

    public void Leave() {
        m_ResourceFree.Set(); // Помечаем ресурс, как свободный
    }

    public void Dispose() { m_ResourceFree.Dispose(); }
}
```

Класс SimpleWaitLock применяется так же, как мы использовали бы класс SimpleSpinLock. Более того, внешне он ведет себя совершенно так же; а вот производительность двух вариантов запириания отличается кардинальным образом. Несмотря на отсутствие конкуренции за право на запириание, класс SimpleWaitLock работает намного медленнее класса SimpleSpinLock, поскольку каждый вызов его методов Enter и Leave заставляет поток совершить переход из управляемого кода в ядро и обратно. Тем не менее при наличии конкуренции проигравший поток блокируется ядром и не зацикливается, не давая впустую тратить ресурсы процессора. Имейте в виду, что переходами из управляемого кода в ядро и обратно сопровождается также создание объекта AutoResetEvent и вызов для него метода Dispose, что отрицательно сказывается на производительности. Впрочем, эти вызовы совершаются редко, так что не стоит слишком сильно беспокоиться по этому поводу.

Чтобы продемонстрировать разницу в производительности, я написал следующий код:

```
public static void Main() {
    Int32 x = 0;
    const Int32 iterations = 10000000; // 10 миллионов

    // Сколько времени займет инкремент x 10 миллионов раз?
    Stopwatch sw = Stopwatch.StartNew();
    for (Int32 i = 0; i < iterations; i++) {
        x++;
    }
    Console.WriteLine("Incrementing x: {0:N0}", sw.ElapsedMilliseconds);
}
```

```

// Сколько времени займет инкремент x 10 миллионов раз, если
// добавить вызов ничего не делающего метода?
sw.Restart();
for (Int32 i = 0; i < iterations; i++) {
    M(); x++; M();
}
Console.WriteLine("Incrementing x in M: {0:N0}", sw.ElapsedMilliseconds);

// Сколько времени займет инкремент x 10 миллионов раз, если
// добавить вызов неконкурирующего объекта SimpleSpinLock?
SimpleSpinLock ssl = new SimpleSpinLock();
sw.Restart();
for (Int32 i = 0; i < iterations; i++) {
    ssl.Enter(); x++; ssl.Leave();
}
Console.WriteLine(
    "Incrementing x in SimpleSpinLock: {0:N0}", sw.ElapsedMilliseconds);

// Сколько времени займет инкремент x 10 миллионов раз, если
// добавить вызов неконкурирующего объекта SimpleWaitLock?
using (SimpleWaitLock swl = new SimpleWaitLock()) {
    sw.Restart();
    for (Int32 i = 0; i < iterations; i++) {
        swl.Enter(); x++; swl.Leave();
    }
    Console.WriteLine(
        "Incrementing x in SimpleWaitLock: {0:N0}", sw.ElapsedMilliseconds);
}
}

[MethodImpl(MethodImplOptions.NoInlining)]
private static void M() { /* Этот метод только возвращает управление */ }

```

Запустив этот код, я получил следующий результат:

```

Incrementing x: 8
Incrementing x in M: 50
Incrementing x in SimpleSpinLock: 219
Incrementing x in SimpleWaitLock: 17,615

```

Как легко заметить, простой инкремент *x* занимает всего 8 мс. Параллельный вызов метода увеличивает время до 42 мс. Выполнение кода в методе, который использует конструкции в режиме пользователя, заставило код работать в 27 (219/8) раз медленней. А теперь обратите внимание, на сколько замедлилась программа при вставке в нее конструкций режима ядра. Результат достигается в 2201 (17 615/8) раз медленней! Поэтому если можете избежать синхронизации потоков, избегайте ее. Если без нее не обойтись, задействуйте конструкции пользовательского режима. И уж совсем не стоит прибегать к конструкциям режима ядра, замедляющим работу кода в 80 (17 615/219) раз по сравнению с конструкциями пользовательского режима.

Семафоры

Семафорами (semaphores) также называют обычные переменные типа `Int32`, управляемые ядром. Ожидающий семафора поток блокируется при значении 0 и освобождается при значениях больше 0. При снятии блокировки с ожидающего семафора потока ядро автоматически вычитает единицу из счетчика. С семафорами связано максимальное значение типа `Int32`, которое ни при каких обстоятельствах не могут превысить текущие показания счетчика. Вот как выглядит класс `Semaphore`:

```
public sealed class Semaphore : WaitHandle {
    public Semaphore(Int32 initialCount, Int32 maximumCount);
    public Int32 Release(); // Вызывает Release(1);
                           // возвращает предыдущее
                           // значение счетчика
    public Int32 Release(Int32 releaseCount); // Возвращает предыдущее
                                              // значение счетчика
}
```

Подытожим, каким образом ведут себя эти три примитива режима ядра:

- ❑ При наличии нескольких потоков в режиме ожидания событие с автосбросом освобождает только один из них.
- ❑ Событие с ручным сбросом снимает блокировку со всех ожидающих его потоков.
- ❑ При наличии нескольких потоков, ожидающих семафора, его появление снимает блокировку с потоков `releaseCount` (здесь `releaseCount` — это аргумент, переданный методу `Release` класса `Semaphore`).

То есть получается, что событие с автосбросом эквивалентно семафору, максимальное значение счетчика которого равно единице. Разница между ними состоит в том, что метод `Set` для события с автосбросом можно вызвать много раз, но каждый раз освобождаться будет всего один поток, в то время как многократный вызов метода `Release` каждый раз увеличивает на единицу внутренний счетчик семафора, давая возможность снять блокировку с большего количества потоков. Однако следует помнить, что при вызове метода `Release` для семафора, показание счетчика которого уже равно максимальному, выбрасывается исключение `SemaphoreFullException`.

При помощи семафоров можно повторно реализовать класс `SimpleWaitLock` таким образом, что нескольким потокам будет предоставлен одновременный доступ к ресурсу (что безопасно только при условии, что все потоки используют ресурс в режиме только для чтения):

```
public sealed class SimpleWaitLock : IDisposable {
    private Semaphore m_AvailableResources;

    public SimpleWaitLock(Int32 maximumConcurrentThreads) {
```

продолжение ➤


```
m_AvailableResources =  
    new Semaphore(maximumConcurrentThreads, maximumConcurrentThreads);  
}  
  
public void Enter() {  
    // Ожидаем в ядре доступа к ресурсу и возвращаем управление  
    m_AvailableResources.WaitOne();  
}  
  
public void Leave() {  
    // Этому потоку доступ больше не нужен; его может получить другой поток  
    m_AvailableResources.Release();  
}  
  
public void Dispose() { m_AvailableResources.Close(); }  
}
```

Мьютексы

Мьютекс (mutex) предоставляет взаимно исключающее записание. Он функционирует аналогично объекту `AutoResetEvent` (или объекту `Semaphore` со значением счетчика 1), так как все три конструкции за один раз освобождают всего один ожидающий поток. Вот как выглядит класс `Mutex`:

```
public sealed class Mutex : WaitHandle {  
    public Mutex();  
    public void ReleaseMutex();  
}
```

Мьютексы снабжены дополнительной логикой, что делает их более сложными по сравнению с другими конструкциями. Во-первых, объекты `Mutex` записывают, какие потоки ими владеют. Для этого делается запрос к идентификатору потока. Если поток вызывает метод `ReleaseMutex`, объект `Mutex` сначала убеждается, что это именно владеющий им поток. Если это не так, состояние объекта `Mutex` не меняется, а метод `ReleaseMutex` вбрасывает исключение `System.ApplicationException`. Если владеющий объектом `Mutex` поток по какой-то причине завершается, пробуждается другой поток, ожидающий мьютекса, и вбрасывает исключение `System.Threading.AbandonedMutexException`. Обычно это исключение остается необработанным, финализируя весь процесс. И это хорошо, ведь новый поток получает объект `Mutex`, старый владелец которого вполне мог быть финализирован перед завершением обновления защищаемых мьютексом данных. Если новый поток перехватит исключение `AbandonedMutexException`, он может попытаться получить доступ к поврежденным данным, что приведет к непредсказуемым результатам и проблемам безопасности.

Кроме того, объекты `Mutex` управляют рекурсивным счетчиком, указывающим, сколько раз поток-владелец уже владел объектом. Если поток владеет

мьютексом в настоящий момент и ожидает его еще раз, рекурсивный счетчик увеличивается на единицу, и потоку разрешается продолжить выполнение. При вызове потоком метода `ReleaseMutex` рекурсивный счетчик уменьшается на единицу. И только после того, как его значение достигнет 0, владельцем мьютекса может стать другой поток.

Большинство пользователей не в восторге от этой дополнительной логики. Проблема в том, что эти «возможности» имеют свою цену. Объекту `Mutex` требуется дополнительная память для хранения идентификатора потока и рекурсивного счетчика. И главное, код объекта `Mutex` должен управлять этой информацией, что тормозит записание. Если приложению понадобятся эти дополнительные возможности, его код сможет реализовать их самостоятельно, не встраивая в объект `Mutex`. Поэтому многие избегают пользоваться мьютексами.

Обычно рекурсивное записание имеет место, если запертый метод вызывает другой метод, также требующий записания. Это демонстрирует следующий код:

```
internal class SomeClass : IDisposable {
    private readonly Mutex m_lock = new Mutex();

    public void Method1() {
        m_lock.WaitOne();
        // Делаем что-то...
        Method2(); // Метод Method2, рекурсивно получающий право на записание
        m_lock.ReleaseMutex();
    }

    public void Method2() {
        m_lock.WaitOne();
        // Делаем что-то...
        m_lock.ReleaseMutex();
    }

    public void Dispose() { m_lock.Dispose(); }
}
```

В приведенном фрагменте код, использующий объект `SomeClass`, может вызвать метод `Method1`, получающий объект `Mutex`. Этот код выполняет какую-то безопасную в отношении потоков операцию, а затем вызывает метод `Method2`, также выполняющий какую-то безопасную в отношении потоков операцию. Благодаря поддержке рекурсии объектом `Mutex` поток сначала дважды запирается, а потом дважды отпирается, и только после этого мьютекс может перейти к новому потоку. Если бы класс `SomeClass` использовал вместо мьютекса объект `AutoResetEvent`, при вызове метода `WaitOne` поток был бы заблокирован.

Рекурсивное записание можно легко организовать при помощи объекта `AutoResetEvent`:

```
internal sealed class RecursiveAutoResetEvent : IDisposable {
    private AutoResetEvent m_lock = new AutoResetEvent(true);
    private Int32 m_owningThreadId = 0;
    private Int32 m_recursionCount = 0;

    public void Enter() {
        // Получаем идентификатор вызывающего потока
        Int32 currentThreadId = Thread.CurrentThread.ManagedThreadId;

        // Если вызывающий поток запирается,
        // увеличиваем рекурсивный счетчик
        if (m_owningThreadId == currentThreadId) {
            m_recursionCount++;
            return;
        }

        // Вызывающий поток не заперт, ожидаем записания
        m_lock.WaitOne();

        // Теперь вызывающий поток запирается, инициализируем
        // идентификатор этого потока и рекурсивный счетчик
        m_owningThreadId = currentThreadId;
        m_recursionCount--;
    }

    public void Leave() {
        // Если вызывающий поток не заперт, ошибка
        if (m_owningThreadId != Thread.CurrentThread.ManagedThreadId)
            throw new InvalidOperationException();

        // Вычитаем единицу из рекурсивного счетчика
        if (--m_recursionCount == 0) {
            // Если рекурсивный счетчик равен 0,
            // не заперт ни один поток
            m_owningThreadId = 0;
            m_lock.Set(); // Пробуждаем первый ожидающий поток (если такие есть)
        }
    }

    public void Dispose() { m_lock.Dispose(); }
}
```

Если поведение классов `RecursiveAutoResetEvent` и `Mutex` идентично, то объект `RecursiveAutoResetEvent` при попытке потока получить право на рекурсивное записание будет иметь потрясающую производительность, так как процедуры отслеживания потока-владельца и рекурсии теперь находятся в управляемом коде. Поток осуществляет переход в ядро Windows только при первом получении объекта `AutoResetEvent` или при окончательной передаче его другому потоку.

Вызов метода при получении доступа к конструкции режима ядра

Блокировка потоков, бесконечно ожидающих доступа к объекту ядра, является нерациональной тратой памяти. Пул потоков предлагает способ вызова вашего метода при освобождении объекта ядра при помощи статического метода `RegisterWaitForSingleObject` класса `System.Threading.ThreadPool`. Есть несколько перегруженных версий этого метода, но все они очень похожи. Вот прототип одной из наиболее часто используемых версий:

```
public static RegisteredWaitHandle RegisterWaitForSingleObject(
    WaitHandle waitObject, WaitOrTimerCallback callback, Object state,
    Int32 millisecondsTimeoutInterval, Boolean executeOnlyOnce);
```

При вызове этого метода аргумент `waitObject` указывает объект ядра, освобождения которого должен ожидать пул потоков. Поскольку этот параметр унаследован от абстрактного класса `WaitHandle`, можно задать любой производный от него класс. В частности, можно передать ссылку на объект `Semaphore`, `Mutex`, `AutoResetEvent` или `ManualResetEvent`. Второй параметр, `callback`, указывает метод, который должен вызвать поток из пула. Вызываемый метод должен соответствовать делегату `System.Threading.WaitOrTimerCallback`, который определяется следующим образом:

```
public delegate void WaitOrTimerCallback(Object state, Boolean timedOut);
```

Параметр `state` метода `RegisterWaitForSingleObject` позволяет задать некоторые данные состояния, которые нужно передать в метод обратного вызова в момент активизации его потоком из пула. При отсутствии данных передается значение `null`. Четвертый параметр, `millisecondsTimeoutInterval`, указывает пулу, как долго он должен ожидать освобождения объекта ядра. Чтобы задать неограниченное время ожидания, обычно передают значение `Timeout.Infinite` (`-1`). Если последний параметр `executeOnlyOnce` имеет значение `true`, пул потоков выполнит метод обратного вызова только раз. В противном случае поток из пула будет выполнять метод обратного вызова при каждом получении сигнала объектом ядра. Это исключительно удобно при ожидании освобождения объекта `AutoResetEvent`.

Методу обратного вызова передаются данные состояния и логическое значение `timedOut`. Если это значение равно `false`, метод узнает, что он вызван после получения сигнала объектом ядра. При значении `true` метод узнает, что он вызывается потому, что объект ядра не получил сигнала за означенное время. Метод обратного вызова может выполнять любые операции, причем выбор операции может зависеть от значения, полученного в аргументе `timedOut`.

Метод `RegisterWaitForSingleObject` возвращает ссылку на объект `RegisteredWaitHandle`. Последний определяет объект ядра, освобождения которого ожидает пул потоков. Если по какой-то причине приложение должно сообщить пулу о необходимости прекратить наблюдение за зарегистрированным дескриптором ожидания, приложение может вызвать метод `Unregister` класса `RegisteredWaitHandle`:

```
public Boolean Unregister(WaitHandle waitObject);
```

Параметр `waitObject` указывает способ уведомления об обслуживании всех рабочих элементов в очереди зарегистрированного объекта ожидания. Если уведомление не требуется, передайте значение `null`. При передаче ссылки на объект, производный от класса `WaitHandle`, пул потоков сообщит объекту об обслуживании всех рабочих элементов.

В следующем коде показано, как поток из пула потоков вызывает метод при каждом получении сигнала объектом `AutoResetEvent`:

```
internal static class RegisteredWaitHandleDemo {
    public static void Main() {
        // Создание объекта AutoResetEvent (изначально несвободного)
        AutoResetEvent are = new AutoResetEvent(false);

        // Заставляем пул ждать AutoResetEvent
        RegisteredWaitHandle rwh = ThreadPool.RegisterWaitForSingleObject(
            are,           // Ждем этого объекта AutoResetEvent
            EventOperation, // При получении доступа вызываем метод EventOperation
            null,          // Передаем null в EventOperation
            5000,          // Ждем освобождения 5 с
            false);         // Вызываем EventOperation при каждом освобождении

        // Начало цикла
        Char operation = (Char) 0;
        while (operation != 'Q') {
            Console.WriteLine("S=Signal, Q=Quit?");
            operation = Char.ToUpper(Console.ReadKey(true).KeyChar);
            if (operation == 'S') are.Set(); // Пользователь хочет задать событие
        }

        // Заставляем пул прекратить ожидание события
        rwh.Unregister(null);
    }

    // Этот метод вызывается при каждом освобождении события
    // или через 5 с после обратного вызова/завершения времени ожидания
    private static void EventOperation(Object state, Boolean timedOut) {
        Console.WriteLine(timedOut ? "Timeout" : "Event became true");
    }
}
```

Глава 29. Гибридные конструкции синхронизации потоков

В главе 28 были рассмотрены простейшие конструкции синхронизации потоков пользовательского режима и режима ядра. На их основе можно строить более сложные конструкции синхронизации. Обычно конструкции пользовательского режима и режима ядра комбинируются, а то что при этом получается, я называю *гибридными конструкциями синхронизации потоков* (hybrid thread synchronization constructs). При отсутствии конкуренции потоков гибридные конструкции дают даже более высокую производительность, чем простейшие конструкции пользовательского режима. В них также применяются простейшие конструкции режима ядра, что позволяет избежать заикливания (пустой траты процессорного времени) при попытке нескольких потоков одновременно получить доступ к процессору. Так как в большинстве приложений потоки практически никогда не устраивают соревнование за доступ к конструкции, повышение производительности помогает ускорить работу приложения.

В этой главе рассматриваются вопросы создания гибридных конструкций на базе простейших конструкций. В частности, вы узнаете, какие именно гибридные конструкции поставляются вместе с FCL, познакомитесь с их поведением и получите представление о том, как правильно с ними работать. Упомянутся и созданные лично мною конструкции, которые доступны для загрузки (<http://Wintellect.com/>).

Ближе к концу главы я покажу, как минимизировать потребление ресурсов и повысить производительность с помощью безопасных в отношении потоков классов коллекций из FCL, являющихся альтернативой гибридным конструкциям. Ну и напоследок мы обсудим классы ReaderWriterGate и SyncGate из библиотеки Power Threading, предлагающие семантику чтения/записи без блокировки потоков и за счет этого сокращающие потребление ресурсов и повышающие производительность.

Простое гибридное записание

Ну что ж, начнем с демонстрации примера гибридного записания в рамках синхронизации потоков:

```
internal sealed class SimpleHybridLock : IDisposable {  
    // Int32 используется примитивными конструкциями  
    // пользовательского режима (методами взаимного записания)  
    private Int32 m_waiters = 0;
```

продолжение ➤

```
// AutoResetEvent - примитивная конструкция режима ядра
private AutoResetEvent m_waiterLock = new AutoResetEvent(false);

public void Enter() {
    // Указываем, что этот поток нуждается в записании
    if (Interlocked.Increment(ref m_waiters) == 1)
        return; // Запираем доступно, конкуренции нет, возвращаем управление

    // Ожидаем другой поток. Блокируем его, так как наличествует конкуренция
    m_waiterLock.WaitOne(); // Значительное снижение производительности
    // Когда WaitOne возвращает управление, этот поток запирается
}

public void Leave() {
    // Этот поток отпирается
    if (Interlocked.Decrement(ref m_waiters) == 0)
        return; // Другие потоки не заблокированы, возвращаем управление

    // Другие потоки заблокированы, пробуждаем один из них
    m_waiterLock.Set(); // Значительное снижение производительности
}

public void Dispose() { m_waiterLock.Dispose(); }
}
```

Класс SimpleHybridLock содержит два поля: одно типа Int32, управляемое примитивными конструкциями пользовательского режима, и второе типа AutoResetEvent, являющееся примитивной конструкцией режима ядра. Чтобы добиться более высокой производительности, при записании нужно пытаться использовать поле Int32 и по возможности не использовать поле AutoResetEvent. Поле AutoResetEvent создается при конструировании объекта SimpleHybridLock и является причиной значительного снижения производительности, особенно по сравнению с полем Int32. Далее в этой главе рассматривается еще одна гибридная конструкция (AutoResetEventSlim), которая не создает поля AutoResetEvent до возникновения конкуренции со стороны потоков, одновременно пытающихся добиться права на записывание. Закрывающий поле AutoResetEvent метод Dispose также значительно снижает производительность, особенно в сравнении с издержками на ликвидацию поля Int32.

Как ни заманчиво выглядит задача повышения производительности при создании и освобождении объекта SimpleHybridLock, лучше сфокусироваться на его методах Enter и Leave, вызываемых за время жизни объекта бесчисленное количество раз. Давайте рассмотрим их подробно.

Первый вызвавший метод Enter поток заставляет метод Interlocked.Increment добавить к полю m_waiters единицу, сделав его значение равным единице. Поток обнаруживает, что прежде потоков, ожидающих права на данное записывание, не было, поэтому после вызова метода Enter он возвращает управление. Здесь важно то, что поток очень быстро запирается. Если теперь появится второй поток и вызовет метод Enter, он увеличит значение поля m_waiters уже до двух и обнаружит

присутствие уже запертого потока, поэтому он блокируется, вызывая метод `WaitOne`, использующий поле `AutoResetEvent`. Вызов метода `WaitOne` заставит поток перейти в ядро Windows, и именно эта процедура приводит к значительному снижению производительности. Однако этот поток в любом случае должен прекратить свою работу, поэтому тот факт, что полная остановка требует лишних временных затрат, не является слишком критичным. В итоге поток блокируется и перестает впустую расходовать процессорное время из-за заикливания. Именно для этого и нужен продемонстрированный еще в главе 28 метод `Enter` класса `SimpleSpinLock`.

Теперь перейдем к методу `Leave`. Его вызов потоком сопровождается вызовом метода `Interlocked.Decrement`, вычитающего из поля `m_waiters` единицу. Равенство этого поля нулю означает отсутствие заблокированных потоков внутри вызова метода `Enter`, поэтому поток, который вызвал метод `Leave`, может просто вернуть управление. И снова посмотрим, насколько быстро все это происходит. Отпирание означает, что поток вычитает единицу из поля `Int32`, выполняет быструю проверку условия и возвращает управление! В то же время, если вызывающий метод `Leave` поток обнаруживает отличное от единицы значение поля `m_waiters`, он узнает о наличии конкуренции и о том, что, по крайней мере, один заблокированный поток в ядре уже имеется. Поток, вызывающий метод `Leave`, должен разбудить один (и только один) из заблокированных потоков. Для этого он вызывает метод `Set` объекта `AutoResetEvent`. Данная операция ведет к снижению производительности, так как потоку приходится совершать переходы к ядру и обратно. К счастью, подобный переход осуществляется только при наличии конкуренции. Разумеется, объект `AutoResetEvent` гарантирует пробуждение только одного из заблокированных потоков; все прочие заблокированные объектом `AutoResetEvent` потоки останутся в таком состоянии, пока новый незаблокированный поток не вызовет метод `Leave`.

ПРИМЕЧАНИЕ

В реальной жизни метод `Leave` может вызвать любой поток в любой момент времени, потому что метод `Enter` не сохраняет информацию о том, какому потоку удалось успешно запереться. Добавить для этого поле и управляющий код несложно, но это увеличивает объем памяти, необходимой для самого объекта записания, и снижает производительность выполнения методов `Enter` и `Leave`, ведь им в результате приходится работать с этим новым полем. Я предпочитаю иметь быстроедействующее записание и корректно использующий его код. С информацией подобного рода не умеют работать ни события, ни семафоры; это могут делать только мьютексы.

Заикливание, владение потоком и рекурсия

Так как переходы в ядро значительно снижают производительность, а потоки остаются запертыми очень короткое время, общую производительность при-

ложения можно повесить, заставив поток перед переходом в режим ядра на некоторое время заикнуться в пользовательском режиме. Если в это время записание, которого ожидает поток, станет возможным, переход в режим ядра не понадобится.

Кроме того, некоторые варианты записания налагают ограничение, в соответствие с которым получить право на записание может только отпираемый поток. Другие варианты записания допускают рекурсивный захват ресурса потоком. Именно такое поведение демонстрирует объект `Mutex`¹. С помощью изощренной логики можно получить гибридное записание, предполагающее одновременно заикливание, владение потоком и рекурсию. Вот пример подобного кода:

```
internal sealed class AnotherHybridLock : IDisposable {
    // Int32 используется примитивом в пользовательском режиме
    // (методы Interlocked)
    private Int32 m_waiters = 0;

    // AutoResetEvent – примитивная конструкция режима ядра
    private AutoResetEvent m_waiterLock = new AutoResetEvent(false);

    // Это поле контролирует заикливание с целью поднять производительность
    private Int32 m_spincount = 4000; // Произвольно выбранный отсчет

    // Эти поля указывают, какой поток и сколько раз запирается
    private Int32 m_owningThreadId = 0, m_recursion = 0;

    public void Enter() {
        // Если вызывающий поток уже заперт, увеличим рекурсивный
        // счетчик на единицу и вернем управление
        Int32 threadId = Thread.CurrentThread.ManagedThreadId;
        if (threadId == m_owningThreadId) { m_recursion++; return; }

        // Вызывающий поток не заперт, пытаемся получить право на записание
        SpinWait spinwait = new SpinWait();
        for (Int32 spinCount = 0; spinCount < m_spincount; spinCount++) {
            // Если записание возможно, этот поток запирается
            // Задаем некоторое состояние и возвращаем управление
            if (Interlocked.CompareExchange(
                ref m_waiters, 1, 0) == 0) goto GotLock;

            // Черная магия: даем остальным потокам шанс
            // запуститься в надежде на отпирание
            spinwait.SpinOnce();
        }

        // Время заикливания истекает, а отпираения еще не случилось.
```

¹ При ожидании объекта `Mutex` поток не заикливается, потому что код этого объекта находится в ядре. То есть проверка состояния объекта `Mutex` возможна только после пересхода потока в ядро.

```
// пытаемся еще раз
if (Interlocked.Increment(ref m_waiters) > 1) {
    // Остальные потоки заблокированы
    // и этот также должен быть заблокирован
    m_waiterLock.WaitOne(); // Ожидаем возможности записи;
                           // производительность падает
    // Проснувшись, этот поток получает право на запись
    // Задаем некоторое состояние и возвращаем управление
}

GotLock:
// Когда поток запирается, записываем его идентификатор
// и указываем, что он получил право на запись впервые
m_owningThreadId = threadId; m_recursion = 1;
}

public void Leave() {
    // Если вызывающий поток не заперт, ошибка
    Int32 threadId = Thread.CurrentThread.ManagedThreadId;
    if (threadId != m_owningThreadId)
        throw new SynchronizationLockException(
            "Lock not owned by calling thread");

    // Уменьшаем на единицу рекурсивный счетчик. Если поток все еще
    // заперт, просто возвращаем управление
    if (--m_recursion > 0) return;

    m_owningThreadId = 0; // Запертых потоков больше нет

    // Если нет других заблокированных потоков, возвращаем управление
    if (Interlocked.Decrement(ref m_waiters) == 0)
        return;

    // Остальные потоки заблокированы, пробуждаем один из них
    m_waiterLock.Set(); // Значительное падение производительности
}

public void Dispose() { m_waiterLock.Dispose(); }
}
```

Как видите, оснащение кода записи дополнительной логикой увеличивает количество имеющихся полей, а значит, и потребление памяти. Код, который должен выполняться, становится сложнее, что также снижает производительность записи. В первом разделе главы 28 сравнивалась производительность конструкции, где увеличивалось на единицу значение типа `Int32` без записи, а также примитивной конструкции пользовательского режима и конструкции режима ядра. Я воспроизведу здесь результаты теста, добавив к ним результаты использования классов `SimpleHybridlock` и `AnotherHybridLock`. Вот они от самого быстрого к самому медленному:

Приращение x	Быстродействие
8	Максимально быстро
B M: 50	В 6 раз медленней
B SimpleSpinLock : 210	В 26 раз медленней
B SimpleHybridLock: 211	В 26 раз медленней (аналогично SimpleSpinLock)
B AnotherHybridLock: 415	В 52 раз медленней (из-за владения/рекурсии)
B SimpleWaitLock: 17615	В 2201 раз медленней

Не имеет значения, что блокировка `AnotherHybridLock` отнимает в два раза больше времени, чем `SimpleHybridLock`. Это обусловлено дополнительной логикой и проверкой ошибок, необходимой, чтобы контролировать владение потоком и рекурсию. Как видите, на производительности отрицательно сказывается любая логика, добавляемая в код записания.

Различные гибридные конструкции

В FCL существует множество гибридных конструкций, изощренная логика которых не выпускает потоки из пользовательского режима, повышая производительность приложения. В некоторых из них до возникновения конкуренции между потоками обращения к конструкциям режима ядра также не происходит. В результате, если конкуренция так и не возникает, приложению не приходится сталкиваться с падением производительности и необходимостью выделять память для объекта. Некоторые конструкции поддерживают объект `CancellationToken` (он рассматривался в главе 26), а значит, поток получает возможность принудительно разблокировать другие потоки, которые могут находиться в режиме ожидания. В этом разделе мы рассмотрим различные типы гибридных конструкций.

Классы `ManualResetEventSlim` и `SemaphoreSlim`

Первые две гибридные конструкции — это классы `System.Threading.ManualResetEventSlim` и `System.Threading.SemaphoreSlim`¹. Они функционируют точно так же, как их аналоги режима ядра, отличаясь только заикливанием в пользовательском режиме, а также тем, что они не создают конструкций режима ядра до возникновения конкуренции. Их методы `Wait` позволяют передать информацию о времени ожидания и объект `CancellationToken`. Вот как выглядят данные классы (некоторые перегруженные версии методов не показаны):

```
public class ManualResetEventSlim : IDisposable {
    public ManualResetEventSlim(Boolean initialState, Int32 spinCount);
```

¹ Так как класса `AutoResetEventSlim` не существует, во многих ситуациях можно обойтись конструированием объекта `SemaphoreSlim` с параметром `maxCount` равным единице.

```

public void Dispose();
public void Reset();
public void Set();
public Boolean Wait(
    Int32 millisecondsTimeout, CancellationToken cancellationToken);
public Boolean IsSet { get; }
public Int32 SpinCount { get; }
public WaitHandle WaitHandle { get; }
}

public class SemaphoreSlim : IDisposable {
    public SemaphoreSlim(Int32 initialCount, Int32 maxCount);
    public void Dispose();
    public Int32 Release(Int32 releaseCount);
    public Boolean Wait(
        Int32 millisecondsTimeout, CancellationToken cancellationToken);
    public Int32 CurrentCount { get; }
    public WaitHandle AvailableWaitHandle { get; }
}

```

Класс Monitor и блокировки синхронизации

Вероятно, самой популярной из гибридных конструкций синхронизации потоков является класс `Monitor`, обеспечивающий взаимноисключающее записание с заикливанием, владением потоком и рекурсией. Данная конструкция используется чаще других потому, что является одной из самых старых, для ее поддержки в C# существует встроенное ключевое слово, с ней по умолчанию умеет работать JIT-компилятор, а CLR пользуется ею от имени приложения. Однако, как вы скоро убедитесь, работать с ней непросто, а получить некорректный код очень легко. Сначала мы рассмотрим саму конструкцию, а потом отдельно остановимся на возможных проблемах и способах их обхода.

С каждым объектом в куче может быть связана структура данных, называемая *блоком синхронизации* (*sync block*). Этот блок содержит поля, похожие на поля ранее упоминавшегося в этой главе класса `AnotherHybridLock`. Точнее, есть поле для объекта ядра, идентификатора потока-владельца, счетчика рекурсии и счетчика ожидающих потоков. Класс `Monitor` является статическим, и его методы принимают ссылки на любой объект кучи. Управление полями эти методы осуществляют в указанном блоке синхронизации объекта. Вот как выглядят чаще всего используемые методы класса `Monitor`:

```

public static class Monitor {
    public static void Enter(Object obj);
    public static void Exit(Object obj);

    // Можно также указать время записания (требуется редко):
    public static Boolean TryEnter(Object obj, Int32 millisecondsTimeout);

```

```
// Аргумент lockTaken будет рассмотрен позднее
public static void Enter(Object obj, ref Boolean lockTaken);
public static void TryEnter(
    Object obj, Int32 millisecondsTimeout, ref Boolean lockTaken);
}
```

Очевидно, что привязка блока синхронизации к каждому объекту в куче является пустой тратой ресурсов, особенно если учесть тот факт, что большинство объектов никогда не пользуются этим блоком. Чтобы снизить потребление памяти, разработчики CLR применили более эффективный вариант реализации описанной функциональности. Во время инициализации CLR выделяется массив блоков синхронизации. Как уже не раз упоминалось в этой книге, при создании объекта в куче с ним связываются два дополнительных служебных поля. Первое поле — указатель на объект-тип — содержит адрес этого объекта в памяти. Второе поле содержит *индекс блоков синхронизации* (sync block index), то есть индекс массива таких блоков.

В момент конструирования объекта этому индексу присваивается значение -1 , что означает отсутствие ссылок на блок синхронизации. Затем при вызове метода `Monitor.Enter` CLR обнаруживает в массиве свободный блок синхронизации и присваивает ссылку на него объекту. То есть привязка объекта к блоку синхронизации происходит «на лету». Метод `Exit` проверяет наличие потоков, ожидающих блока синхронизации. Если таких потоков не обнаруживается, метод возвращает индексу значение -1 , означающее, что блоки синхронизации свободны и могут быть связаны с какими-нибудь другими объектами.

Рисунок 29.1 демонстрирует связь между объектами кучи, их индексами блоков синхронизации и элементами массива блоков синхронизации в CLR. Указатель на объект-тип объектов *A*, *B* и *C* ссылается на тип *T*. Это говорит о принадлежности всех трех объектов к одному и тому же типу. Как обсуждалось в главе 4, объект-тип также находится в куче и подобно всем остальным объектам обладает двумя служебными членами: индексом блока синхронизации и указателем на объект-тип. То есть блок синхронизации можно связать с объектом-типом, а ссылку на этот объект можно передать методам класса `Monitor`. Кстати, массив блоков синхронизации при необходимости может увеличить количество блоков, поэтому не стоит беспокоиться, что при одновременной синхронизации нескольких объектов блоков не хватит.

Вот код, демонстрирующий исходное предназначение класса `Monitor`:

```
internal sealed class Transaction {
    private DateTime m_timeOfLastTrans;

    public void PerformTransaction() {
        Monitor.Enter(this);
        // Этот код имеет эксклюзивный доступ к данным...
        m_timeOfLastTrans = DateTime.Now;
        Monitor.Exit(this);
    }
}
```

```

public DateTime LastTransaction {
    get {
        Monitor.Enter(this);
        // Этот код имеет совместный доступ к данным...
        DateTime temp = m_timeOfLastTrans;
        Monitor.Exit(this);
        return temp;
    }
}

```

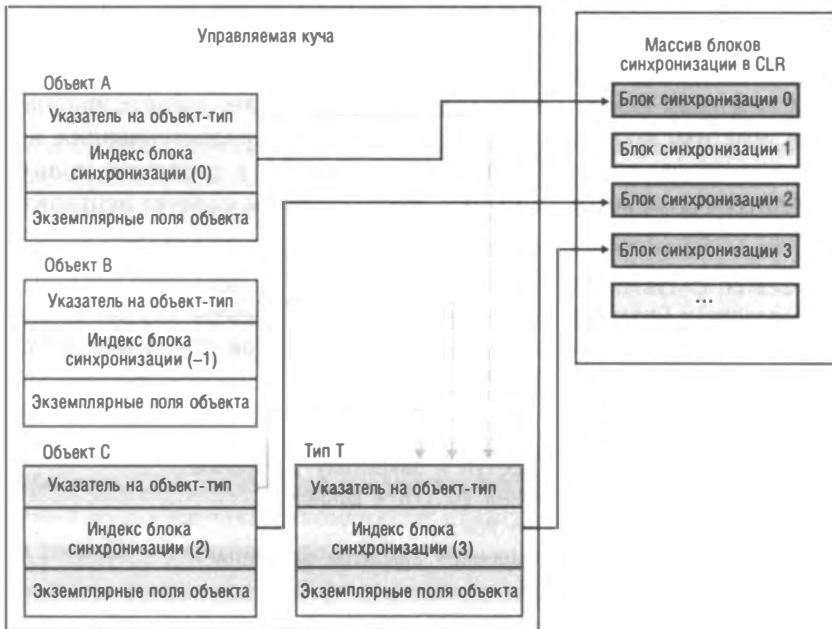


Рис. 29.1. Индекс блоков синхронизации объектов в куче (включая объекты-типы) может ссылаться на запись в массиве блоков синхронизации CLR

На первый взгляд, кажется, что все достаточно просто, но это не так. Проблема в том, что индекс блока синхронизации каждого объекта подразумевается открытым. И вот как это проявляется:

```

public static void SomeMethod() {
    var t = new Transaction();
    Monitor.Enter(t); // Этот поток получает право
                     // на открытое запирание объекта

    // Заставляем поток пула вывести время LastTransaction
    // ПРИМЕЧАНИЕ. Поток пула заблокирован до вызова
    // методом SomeMethod метода Monitor.Exit!
}

```

```
ThreadPool.QueueUserWorkItem(o => Console.WriteLine(t.LastTransaction));

// Здесь выполняется какой-то код...
Monitor.Exit(t);
}
```

В этом коде поток, выполняющий метод `SomeMethod`, вызывает метод `Monitor.Enter`, получая право на открытое записание объекта `Transaction`. Когда поток пула запрашивает свойство `LastTransaction`, это свойство также вызывает метод `Monitor.Enter`, чтобы получить право на то же самое записание. В результате поток пула оказывается заблокированным, пока поток, выполняющий метод `SomeMethod`, не вызовет метод `Monitor.Exit`. При помощи отладчика можно определить, что поток пула заблокирован внутри свойства `LastTransaction`, но узнать, какой еще поток заперт, очень сложно. Для этого нужно понять, какой именно код привел к запиранию. Но даже если вы это узнаете, вполне может оказаться, что этот код окажется недоступным для редактирования, а значит, вы не сможете устранить проблему. Именно поэтому я предлагаю пользоваться только закрытыми записаниями. Вот каким образом следует исправить класс `Transaction`:

```
internal sealed class Transaction {
    private readonly Object m_lock = new Object(); // Теперь записание
                                                    // в рамках каждой транзакции ЗАКРЫТО
    private DateTime m_timeOfLastTrans;

    public void PerformTransaction() {
        Monitor.Enter(m_lock); // Доступ к закрытому запиранию
        // Этот код имеет эксклюзивный доступ к данным...
        m_timeOfLastTrans = DateTime.Now;
        Monitor.Exit(m_lock); // Завершаем закрытое записание
    }

    public DateTime LastTransaction {
        get {
            Monitor.Enter(m_lock); // Доступ к закрытому запиранию
            // Этот код имеет совместный доступ к данным...
            DateTime temp = m_timeOfLastTrans;
            Monitor.Exit(m_lock); // Завершаем закрытое записание
            return temp;
        }
    }
}
```

Если бы члены класса `Transaction` были статическими, для их безопасности в отношении потоков достаточно было бы сделать статическим поле `m_lock`. Однако я думаю, вы уже поняли из предшествующих обсуждений, что класс `Monitor` не может быть реализован как статический; его следует реализовать, как и все прочие конструкции, в виде класса, допускающего создание экземпляров

и вызов для него экземплярных методов. Вот какие проблемы возникают при реализации класса `Monitor` как статического:

- ❑ Если тип объекта-представителя является производным от `System.MarshalByRefObject`, на такой объект может ссылаться переменная (эта тема рассматривалась в главе 22). При передаче методам класса `Monitor` ссылки на такой представитель запирается представитель, а не представляемый им объект.
- ❑ Если поток вызывает метод `Monitor.Enter` и передает в него ссылку на объект-тип, загруженный нейтрально по отношению к домену (о том, как это сделать, мы говорили в главе 22), поток запирает этот тип во всех доменах процесса. Это известная недоработка CLR, нарушающая декларируемую изолированность доменов. Исправить ее без потери производительности сложно, поэтому никто этим не занимается. Пользователям просто рекомендуют никогда не передавать ссылку на объект-тип в методы класса `Monitor`.
- ❑ Так как строки допускают интернирование (это обсуждалось в главе 14), два разных фрагмента кода могут ошибочно сослаться на один и тот же объект `String` в памяти. При передаче ссылки на этот объект в методы типа `Monitor` требуется синхронизация выполнения этих двух фрагментов кода.
- ❑ При передаче строки за границу домена CLR не создает ее копию; ссылка на строку просто передается в другой домен. Это повышает производительность, и в теории все должно быть в порядке, так как объекты типа `String` неизменны. Но с ними, как и с любыми другими объектами, связан индекс блока синхронизации, допускающий редактирование. А это вынуждает потоки в различных доменах синхронизироваться друг с другом. Это еще одна недоработка CLR, связанная с недостаточной изолированностью доменов. Поэтому пользователям рекомендуют никогда не передавать ссылок на объекты типа `String` методам класса `Monitor`.
- ❑ Так как методы класса `Monitor` принимают параметры типа `Object`, передача им значимого типа приводит к его упаковке. В результате поток запирает упакованный объект. При каждом вызове метода `Monitor.Enter` запирается другой объект, и синхронизация потоков вообще отсутствует.
- ❑ Применение к методу атрибута `[MethodImpl(MethodImplOptions.Synchronized)]` заставляет JIT-компилятор окружить машинный код метода вызовами `Monitor.Enter` и `Monitor.Exit`. Если мы имеем дело с экземплярным методом, он передается указанным методам, запирая неявно открытое запирание. В случае статического метода этим двум методам передается ссылка на объект-тип, потенциально запирая нейтральный по отношению к домену тип. Поэтому использовать данный атрибут не рекомендуется.
- ❑ При вызове конструктора типов (он обсуждался в главе 8) CLR запирает для типа объект-тип, гарантируя, что всего один поток примет участие в инициализации данного объекта и его статических полей. И снова загрузка этого типа нейтрально по отношению к домену создает проблемы. К примеру, если код конструктора типа войдет в бесконечный цикл, тип станет

непригодным для всех доменов в процессе. В данном случае рекомендуется по возможности избегать конструкторов типа или хотя бы делать их как можно более короткими и простыми.

К сожалению, все становится только хуже. Так как разработчики привыкли в одном и том же методе выполнять записание, что-то делать, а затем выполнять отпирание, в С# появился упрощенный синтаксис в виде ключевого слова `lock`. Рассмотрим следующий метод:

```
private void SomeMethod() {  
    lock (this) {  
        // Этот код имеет эксклюзивный доступ к данным...  
    }  
}
```

Это то же самое, что написать:

```
private void SomeMethod() {  
    Boolean lockTaken = false;  
    try {  
        //  
        Monitor.Enter(this, ref lockTaken);  
        // Этот код имеет эксклюзивный доступ к данным...  
    }  
    finally {  
        if (lockTaken) Monitor.Exit(this);  
    }  
}
```

Первая проблема в данном случае состоит в принятом разработчиками С# решении, что метод `Monitor.Exit` лучше вызывать в блоке `finally`. Это гарантирует отпирание вне зависимости от происходящего в блоке `try`. Однако ничего хорошего в этом нет. Если в блоке `try` в процессе изменения состояния возникнет исключение, состояние окажется поврежденным. И отпирание в блоке `finally` приведет к тому, что с поврежденным состоянием начнет работать другой поток. Лучше позволить приложению зависнуть, чем оставить его работать с поврежденными данными и потенциальными брешами в защите. Кроме того, вход в блок `try` и выход из блока снижает производительность метода. Некоторые JIT-компиляторы не поддерживают подстановку для методов, в которых имеются блоки `try`, что еще больше снижает производительность. В итоге мы получаем более медленный код, к тому же допускающий доступ потоков к поврежденному состоянию¹. Поэтому я крайне не рекомендую вам пользоваться инструкцией `lock`.

Теперь перейдем к переменной `lockTaken` типа `Boolean` и к проблеме, которую призвана решить эта переменная. Предположим, поток вошел в блок `try` и был прерван до вызова метода `Monitor.Enter` (прерывание потоков обсуждалось

¹ Вы можете безопасно выполнять отпирание в блоке `finally`, если код блока `try` только читает состояние, не пытаясь его редактировать. Хотя производительность при этом также падает.

в главе 22). После этого вызывается метод `finally`, но его код не должен выполнять отпирание. В этом нам поможет переменная `lockTaken`. Ей присваивается начальное значение `false`, означающее, что запираания еще не было. Если вызванный метод `Monitor.Enter` успешно запирается, переменной `lockTaken` присваивается значение `true`. Блок `finally` по значению этой переменной определяет, нужно ли вызывать метод `Monitor.Exit`¹. Кстати, структура `SpinLock` также поддерживает работу с переменной `lockTaken`.

Класс `ReaderWriterLockSlim`

Часто потоки просто читают некие данные. Если такие данные защищены взаимноисключающим запираением (например, `SimpleSpinLock`, `SimpleWaitLock`, `SimpleHybridLock`, `AnotherHybridLock`, `Mutex` или `Monitor`), то при попытке одновременного доступа нескольких потоков работу продолжит только один из них, а остальные блокируются, что значительно ухудшает масштабируемость и снижает производительность вашего приложения. Впрочем, в случае доступа в режиме только для чтения необходимость в блокировке отпадает, и потоки получают одновременный доступ к данным. А вот потоку, который хочет внести в данные изменения, требуется эксклюзивный доступ. Конструкция `ReaderWriterLockSlim` содержит логику, позволяющую решить данную проблему. Управление потоками осуществляется следующим образом:

- ❑ Если один поток осуществляет запись данных, все остальные потоки, требующие доступа, блокируются.
- ❑ Если один поток читает данные, все остальные потоки, требующие доступа, продолжают работу, блокируются только потоки, ожидающие доступа на запись.
- ❑ После завершения работы потока, осуществлявшего запись данных, деблокируется либо один поток, ожидающий доступ на запись, либо все потоки, ожидающие доступ на чтение. При отсутствии заблокированных потоков право на запираение получит следующий поток чтения или записи, которому это потребуется.
- ❑ После завершения всех потоков, осуществлявших чтение данных, деблокируется поток, ожидающий разрешения на запись. При отсутствии заблокированных потоков право на запираение получит следующий поток чтения или записи, которому это потребуется.

Вот как выглядит данный класс (некоторые перегруженные версии методов не показаны):

¹ Блоки `try/finally` и переменная `lockTaken` могут быть полезны при передаче в методы класса `Monitor` ссылки на объект, не связанный с доменом (например, строку), или на нейтральный по отношению к домену объект. Даже если состояние в одном домене окажется поврежденным, потоки в остальных доменах продолжают свою работу.

```

public class ReaderWriterLockSlim : IDisposable {
    public ReaderWriterLockSlim(LockRecursionPolicy recursionPolicy);
    public void Dispose();

    public void EnterReadLock();
    public Boolean TryEnterReadLock(Int32 millisecondsTimeout);
    public void ExitReadLock();
    public void EnterWriteLock();
    public Boolean TryEnterWriteLock(Int32 millisecondsTimeout);
    public void ExitWriteLock();

    // Большинство приложений никогда не обращаются к этим свойствам
    public Boolean IsReadLockHeld { get; }
    public Boolean IsWriteLockHeld { get; }
    public Int32 CurrentReadCount { get; }
    public Int32 RecursiveReadCount { get; }
    public Int32 RecursiveWriteCount { get; }
    public Int32 WaitingReadCount { get; }
    public Int32 WaitingWriteCount { get; }
    public LockRecursionPolicy RecursionPolicy { get; }
    // Не показаны члены, связанные с переходом от чтения к записи
}

```

Вот код, демонстрирующий применение данной конструкции:

```

internal sealed class Transaction : IDisposable {
    private readonly ReaderWriterLockSlim m_lock =
        new ReaderWriterLockSlim(LockRecursionPolicy.NoRecursion);
    private DateTime m_timeOfLastTrans;

    public void PerformTransaction() {
        m_lock.EnterWriteLock();
        // Этот код имеет эксклюзивный доступ к данным...
        m_timeOfLastTrans = DateTime.Now;
        m_lock.ExitWriteLock();
    }

    public DateTime LastTransaction {
        get {
            m_lock.EnterReadLock();
            // Этот код имеет совместный доступ к данным...
            DateTime temp = m_timeOfLastTrans;
            m_lock.ExitReadLock();
            return temp;
        }
    }

    public void Dispose() { m_lock.Dispose(); }
}

```

С этой конструкцией связан ряд концепций, заслуживающих отдельного упоминания. Во-первых, конструктору `ReaderWriterLockSlim` можно передать флаг `LockRecursionsPolicy`, определенный следующим образом:

```
public enum LockRecursionPolicy { NoRecursion, SupportsRecursion }
```

Флаг `SupportsRecursion` наделяет код записи механизмом владения потоком и рекурсивным поведением. Как уже упоминалось в этой главе, эти режимы негативно влияют на производительность записи, так что я рекомендую всегда передавать конструктору флаг `LockRecursionPolicy.NoRecursion` (как это сделано в моем примере). Поддержка режимов владения потоком и рекурсии для записи на чтение-запись является крайне дорогим удовольствием, ведь при записи нужно отслеживать все блокируемые потоки, занимающиеся чтением данных, и поддерживать для каждого из них отдельный счетчик рекурсии. На самом деле для управления всей этой информацией в безопасном отношении потоков режиме конструкция `ReaderWriterLockSlim` внутренне использует взаимоисключающее записание с заикливанием! И я не шучу.

Класс `ReaderWriterLockSlim` содержит дополнительные методы (ранее они не демонстрировались), позволяющие читающему потоку превратиться в поток записывающий. Затем возможен обратный переход. В основе такого подхода лежит идея, что в процессе чтения данных потоком может возникнуть необходимость их редактирования. Поддержка данного поведения снижает производительность записи. Ну а я так вообще считаю его бесполезным. Дело в том, что поток не может просто так превратиться из читающего в пишущий. Перед тем как он получит позволение на подобное преобразование, все остальные читающие потоки должны покинуть код записи. Это то же самое, что отпереть поток чтения и тут же запереть его для записи.

ПРИМЕЧАНИЕ

В FCL также присутствует конструкция `ReaderWriterLock`, появившаяся еще в Microsoft .NET Framework 1.0. Она была настолько проблемной, что в версию 2.0 разработчики Microsoft ввели конструкцию `ReaderWriterLockSlim`. Менять конструкцию `ReaderWriterLock` они не стали, чтобы не потерять совместимости с использующими ее приложениями. Данная конструкция работает крайне медленно даже в отсутствии конкуренции потоков. Она не может отказаться от владения потоком и рекурсивного поведения, еще сильнее замедляющих записание. Потоки чтения имеют в этой конструкции приоритет перед потоками записи, что может привести к появлению длинной очереди.

Класс `OneManyLock`

Я создал собственную конструкцию чтения-записи, работающую быстрее, чем встроенный в FCL класс `ReaderWriterLockSlim`¹. Эта конструкция называется

¹ Код в файле `Ch29-1-HybridThreadSync.cs` является частью сопроводительного кода к данной книге. Вы можете загрузить его с сайта <http://Wintellect.com/>.

OneManyLock, так как она предоставляет доступ либо одному пишущему потоку, либо нескольким читающим. Данный класс выглядит примерно следующим образом:

```
public sealed class OneManyLock : IDisposable {  
    public OneManyLock();  
    public void Dispose();  
    public void Enter(Boolean exclusive);  
    public void Leave();  
}
```

Теперь посмотрим, как это работает. Класс содержит поле типа `Int32`, предназначенное для записи состояния записывания, объект `Semaphore`, блокирующий читающие потоки, и объект `AutoResetEvent`, блокирующий пишущие потоки. Поле записи состояния содержит в себе пять вложенных полей:

- ❑ Три бита (число от 0 до 7) представляют состояние самого записывания. Значение 0 означает `Free` (доступно), 1 — `OwnedByWriter` (занято записывающим потоком), 2 — `OwnedByReaders` (занято читающими потоками), 3 — `OwnedByReadersAndWriterPending` (занято записывающим и читающими потоками) и 4 — `ReservedForWriter` (зарезервировано для записывающего потока). Значения 5, 6 и 7 не используются.
- ❑ Девять битов (число от 0 до 511) представляют количество потоков чтения (`RR`), которые разрешается в данный момент запереть.
- ❑ Девять битов (число от 0 до 511) представляют количество потоков чтения (`RW`), ожидающих записывания. Эти потоки удерживает объект `AutoResetEvent`.
- ❑ Девять битов (число от 0 до 511) представляют количество потоков записи (`WW`), ожидающих записывания. Эти потоки удерживает объект `Semaphore`.
- ❑ Два оставшихся бита не используются и всегда имеют значение 0.

Теперь, когда вся информация о записывании сконцентрирована в одном поле типа `Int32`, я могу управлять этим полем при помощи методов класса `Interlocked`. В результате записывание выполняется очень быстро и приводит к блокированию потока только при конкуренции потоков.

Вот что происходит при входе потока в код записывания совместного доступа:

- ❑ Если записывание возможно, присваиваем состоянию значение `OwnedByReaders`, выполняем `RR = 1`, возвращаем управление.
- ❑ Если состояние записывания имеет значение `OwnedByReaders` (занято потоком чтения), выполняем `RR++`, возвращаем управление.
- ❑ В противном случае выполняем `RW++`, блокируем поток чтения. Когда поток проснется, проходим цикл и делаем вторую попытку.

Вот что происходит при выходе потока из кода записи совместного доступа:

- ❑ Выполняем $RR--$.
- ❑ Если $RR > 0$, возвращаем управление.
- ❑ Если $WW > 0$, присваиваем состоянию значение `ReservedForWriter` (зарезервировано для потока записи), выполняем $WW--$, освобождаем один заблокированный поток записи, возвращаем управление.
- ❑ Если $RW = 0$ и $WW = 0$, присваиваем состоянию значение `Free` (свободно), возвращаем управление.

Вот что происходит при входе потока в код записи с эксклюзивным доступом:

- ❑ Если записание возможно, присваиваем состоянию значение `OwnedByWriter` (занято потоком записи), возвращаем управление.
- ❑ Если состояние записи равно `ReservedForWriter` (зарезервировано для потока записи), присваиваем состоянию значение `OwnedByWriter` (занято потоком записи), возвращаем управление.
- ❑ Если состояние записи равно `OwnedByWriter` (занято потоком записи), выполняем $WW++$, блокируем поток записи. Когда поток проснется, проходим цикл и делаем вторую попытку.
- ❑ В противном случае присваиваем состоянию значение `OwnedByReadersAndWriterPending` (ожидание потоков чтения и записи), выполняем $WW++$, блокируем поток записи. Когда поток проснется, проходим цикл и делаем вторую попытку.

Вот что происходит при выходе из блокировки потока с эксклюзивным доступом:

- ❑ Если $WW = 0$ и $RW = 0$, присваиваем состоянию значение `Free` (свободно), возвращаем управление.
- ❑ Если $WW > 0$, присваиваем состоянию значение `ReservedForWriter` (зарезервировано для потока записи), выполняем $WW--$, освобождаем один заблокированный поток записи, возвращаем управление.
- ❑ Если $WW = 0$ и $RW > 0$, присваиваем состоянию значение `Free` (свободно), выполняем $RW = 0$, пробуждаем все заблокированные потоки чтения, возвращаем управление.

Предположим, что у нас один запертый поток осуществляет чтение, а другой ждет отпираания, чтобы осуществить запись. Записывающий поток сначала проверит, возможно ли записание, и так как результат будет отрицательным, начнет готовиться к следующей проверке. Но в этот момент читающий поток может покинуть код записи, и обнаружив, что значения RR и WW равны 0, поток присвоит состоянию записи значение `Free`. Однако проблема в том,

Для этого все манипуляции с битами выполняются с применением техники, описанной в разделе «Универсальный эталон взаимного записывания» главы 28. Если помните, данный эталон превращает любую операцию в безопасную в отношении потоков и атомарную. Именно это обеспечивает быстрое записывание. Сравнив производительность моего класса `OneManyLock` и классов `ReaderWriterLockSlim` и `ReaderWriterLock` из `FCL`, я получил следующий результат:

Приращение x в ReaderWriterLock: 2,051 — примерно в 5,0 раза медленней.

В завершение этого раздела я в очередной раз упомяну мою библиотеку Power Threading library (ее можно загрузить бесплатно по адресу <http://Wintellect.com/>). Она содержит немного другую версию данного записки, которая называется OneManyResourceLock. Этот и другие варианты записки из моей библиотеки предлагают множество дополнительных возможностей, например распознавание клинчей (deadlocks), включение режимов владения запиской и рекурсии (за что придется платить снижением производительности), наблюдение за поведением записки в процессе выполнения, имеется даже унифицированная программная модель всех записок. При наблюдении за поведением можно узнать максимальное время ожидания записки потоком, а также максимальное и минимальное время записки.

Следующая конструкция называется `System.Threading.CountdownEvent`. Она построена на основе объекта `ManualResetEventSlim` и блокирует поток до достижения внутренним счетчиком значения 0. Поведение этой конструкции диаметрально противоположно поведению семафора (блокирующего потоки, пока значение счетчика равно 0). Вот как выглядит данный класс (некоторые перегруженные версии методов не показаны):

[illegible]

```

public Boolean TryAddCount(Int32 signalCount); // Инкремент CurrentCount
                                              // с помощью signalCount
public Boolean Signal(Int32 signalCount);     // Декремент CurrentCount
                                              // с помощью signalCount
public Boolean Wait(
    Int32 millisecondsTimeout, CancellationToken cancellationToken);

public Int32 CurrentCount { get; }
public Boolean IsSet { get; }                // true, если
                                              // CurrentCount равно 0
public WaitHandle WaitHandle { get; }
}

```

Достигнутое параметром `CurrentCount` класса `CountdownEvent` нулевое значение уже не может быть изменено. Если параметр `CurrentCount` равен 0, метод `AddCount` вбрасывает исключение `InvalidOperationException`, а метод `TryAddCount` просто возвращает значение `false`.

Класс `Barrier`

Конструкция `System.Threading.Barrier` была создана для решения достаточно редко возникающей проблемы, так что вряд ли вам когда-нибудь придется ею пользоваться. Она управляет параллельным режимом выполнения потоков, позволяя им одновременно проходить все фазы алгоритма. К примеру, когда CLR задействует серверную версию сборщика мусора, его алгоритм создает один поток исполнения для каждого ядра. Эти потоки проходят через различные стеки приложения, одновременно помечая объекты в куче. Завершив свою порцию работы, поток должен остановиться и подождать завершения работы остальных. Когда все объекты будут помечены, потоки смогут одновременно приступить к сжатию различных фрагментов кучи. Поток, закончивший сжимать кучу, следует заблокировать, чтобы он дождался завершения остальных потоков. Потом все потоки одновременно пройдут через стек потоков приложения, присваивая корням ссылки на новые местоположения сжатых объектов. И только после завершения всех потоков деятельность сборщика мусора считается оконченной и появляется возможность восстановить поток приложения.

Данный сценарий легко реализуется при помощи класса `Barrier`, который выглядит следующим образом (некоторые перегруженные версии методов не показаны):

```

public class Barrier : IDisposable {
    public Barrier(Int32 participantCount, Action<Barrier> postPhaseAction);
    public void Dispose();
    public Int64 AddParticipants(Int32 participantCount); // Добавление
                                                         // участников
    public void RemoveParticipants(Int32 participantCount); // Удаление
                                                         // участников
}

```

продолжение ➤


```

public Boolean SignalAndWait(
    Int32 millisecondsTimeout, CancellationToken cancellationToken);

public Int64 CurrentPhaseNumber { get; }    // Показывает фазы процесса
                                           // (начиная с 0)
public Int32 ParticipantCount { get; }      // Количество участников
public Int32 ParticipantsRemaining { get; } // Число потоков, необходимых
                                           // для вызова SignalAndWait
}

```

При конструировании класса `Barrier` указывается количество потоков, которые будут принимать участие в работе. Можно также передать в конструктор делегат `Action<Barrier>`, ссылающийся на код, который будет вызван после завершения всеми участниками очередной фазы. Динамически добавлять потоки к классу `Barrier` и удалять их оттуда можно при помощи методов `AddParticipant` и `RemoveParticipant`, но на практике это делается крайне редко. Завершивший свою фазу работы поток должен вызвать метод `SignalAndWait`, который заставит метод `Barrier` заблокировать данный поток (с помощью конструкции `ManualResetEventSlim`). После вызова метода `SignalAndWait` всеми участниками метод `Barrier` вызывает делегата (с помощью последнего обращавшегося к методу `SignalAndWait` потока) и снимает блокировку со всех потоков, давая им возможность перейти к следующей фазе.

Резюме по гибридным конструкциям

Я рекомендую вам избегать кода, блокирующего потоки. Выполняя асинхронные вычисления или операции ввода-вывода, передавайте данные от одного потока к другому так, чтобы исключить одновременную попытку доступа к данным со стороны нескольких потоков. Я продемонстрировал это на примере сервера каналов в главе 27. Если вы не можете справиться с подобной задачей, используйте методы `VolatileRead`, `VolatileWrite`, а также методы класса `Interlocked`, так как они работают быстро и не блокируют потоки. К сожалению, они подходят только для работы с простыми типами. Впрочем, даже в этом случае вы можете выполнять достаточно сложные операции, описанные в предыдущей главе.

Вот две основные причины, по которым приходится блокировать потоки:

- ❑ **Упрощение модели программирования.** Блокируя поток, вы жертвуете ресурсами и производительностью, но получаете возможность писать код последовательно, не прибегая к методам обратного вызова.
- ❑ **Поток имеет определенное назначение.** Некоторые потоки используются для решения конкретных задач. Лучшим примером этого является основной поток приложения. Без блокировки он, в конечном счете, вернет управление, и процесс завершится. Другим примером является поток или потоки графического интерфейса приложения. `Windows` требует, чтобы манипуляции

окнами и элементами управления осуществлял только породивший их поток. Поэтому периодически приходится писать код, блокирующий GUI-поток до завершения каких-то других операций. И только после этого данный поток обновляет окна и элементы управления. Разумеется, блокировка GUI-потока подвешивает приложение и мешает работе пользователя.

Чтобы избежать блокировки потоков, не стоит мысленно назначать им конкретные задания. К примеру, не нужно создавать поток, проверяющий орфографию, поток, проверяющий грамматику, поток, обрабатывающий определенные клиентские запросы и т. п. Потому что подобный подход как бы показывает, что поток не может выполнять других заданий. Однако потоки являются слишком ценным ресурсом, чтобы ограничивать их только одним заданием. Используйте пул для аренды потоков на короткий период времени. Поток из пула может начать с проверки орфографии, затем перейти к проверке грамматики, затем заняться обработкой клиентских запросов и т. п.

Если же вопреки написанному вы решите блокировать потоки, для синхронизации потоков из разных доменов или процессов используйте конструкции режима ядра. Для атомарного управления состоянием через набор операций вам потребуется класс `Monitor` с закрытым полем¹. В качестве альтернативы можно прибегнуть к запираанию на чтение-запись. В общем случае это запираание работает медленнее класса `Monitor`, но оно позволяет одновременно исполняться нескольким потокам чтения, что повышает общую производительность и минимизирует возможность блокировки потоков.

Также старайтесь избегать рекурсивных блокировок (особенно блокировок чтения-записи), так как они серьезно снижают производительность. Впрочем, класс `Monitor`, несмотря на свою рекурсивность, показывает высокую производительность². Кроме того, старайтесь не снимать блокировку в блоке `finally`, так как вход в блоки обработки исключений и выход из них негативно сказывается на производительности. Кроме того, вбрасывание исключения при изменении состояния приводит к ситуации, когда другим потокам приходится работать с поврежденными данными, из-за чего результат работы приложения становится непредсказуемым и возникают бреши в системе безопасности.

И, разумеется, если ваш код все-таки выполняет запираание, он не должен удерживать запираание слишком долго, так как в результате повышается вероятность блокировки потоков. Далее будет показана техника работы с классами коллекций, позволяющая избежать длительного запираания.

Ну и, наконец, чтобы не задействовать конструкции синхронизации потоков в вычислительных операциях, можно использовать задания (о которых мы говорили в главе 26). В частности, мне нравится, когда с каждым заданием свя-

¹ Вместо класса `Monitor` можно воспользоваться чуть более быстрым классом `SpinLock`. Но при этом возможна ситуация, когда ресурсы процессора начнут тратиться впустую. И с моей точки зрения, класс `SpinLock` не настолько превосходит `Monitor` по скорости, чтобы оправдать его применение.

² Частично это связано с реализацией класса `Monitor` на машинном, а не на управляемом коде.

зывается одно или несколько других заданий, которые начинают выполняться средствами пула потоков при завершении некой операции. Это намного лучше, чем блокировать поток, ожидая завершения операции. Согласно модели асинхронного программирования (APM), по завершении операций ввода-вывода выполняется метод обратного вызова; что аналогично вызову нового задания после завершения текущего.

Запирание с двойной проверкой

Существует известный прием, называемый *запиранием с двойной проверкой* (double-check locking). К нему прибегают, если нужно отложить создание одноэлементного объекта до тех пор, пока он не потребуется приложению — иногда это называют *отложенной инициализацией* (lazy initialization). Без запроса объект никогда не создается, что экономит время и память. Проблемы могут возникнуть при одновременном запросе объекта несколькими потоками. Чтобы в результате у вас появился всего один объект, потребуется синхронизация потоков.

Этот прием известен вовсе не благодаря своей интересности или полезности, просто о нем очень много писали. Раньше он часто применялся при программировании на Java, но позже обнаружилось, что Java не гарантирует стопроцентной работоспособности результата. Документ с описанием этой проблемы находится по адресу www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

Думаю, вы будете рады узнать, что благодаря модели памяти и доступу к волатильным полям (см. главу 28) CLR прекрасно поддерживает запирание с двойной проверкой. Вот код, демонстрирующий реализацию данной техники на языке C#:

```
internal sealed class Singleton {
    // Объект s_lock требуется для обеспечения безопасности
    // в многопоточной среде. Наличие этого объекта предполагает,
    // что для создания одноэлементного объекта требуется больше
    // ресурсов, чем для объекта System.Object, и что эта процедура
    // может вовсе не понадобиться. В противном случае проще и эффективнее
    // получить одноэлементный объект в конструкторе класса
    private static readonly Object s_lock = new Object();

    // Это поле ссылается на один объект Singleton
    private static Singleton s_value = null;

    // Закрытый конструктор не дает внешнему коду создавать экземпляры
    private Singleton() {
        // Код инициализации объекта Singleton
    }
}
```

```
// Открытый статический метод, возвращающий объект Singleton
// (создавая его при необходимости)
public static Singleton GetSingleton() {
    // Если объект Singleton уже создан, возвращаем его
    if (s_value != null) return s_value;

    Monitor.Enter(s_lock); // Если не создан, позволяем одному
                          // потоку сделать это
    if (s_value == null) {
        // Если объекта все еще нет, создаем его
        Singleton temp = new Singleton();

        // Сохраняем ссылку в переменной s_value (см. обсуждение далее)
        Interlocked.Exchange(ref s_value, temp);
    }
    Monitor.Exit(s_lock);

    // Возвращаем ссылку на объект Singleton
    return s_value;
}
```

Принцип запирания с двойной проверкой состоит в том, что при вызове метода `GetSingleton` быстро проверяется поле `s_value`, чтобы выяснить, создан ли объект. При положительном результате проверки метод возвращает ссылку на объект. В результате отпадает необходимость в синхронизации потоков, и приложение работает очень быстро. Однако если поток, вызвавший метод `GetSingleton`, не обнаруживает объекта, он прибегает к запиранию в рамках синхронизации потоков, гарантируя, что созданием объекта будет заниматься только один поток. То есть снижение производительности наблюдается только после первого запроса к одноэлементному объекту.

Теперь можно поговорить о том, почему подобный эталон не работает в Java. В начале метода `GetSingleton` виртуальная машина Java считывает значение поля `s_value` в регистр процессора и при выполнении второй инструкции `if` ограничивается запросом к этому регистру. В итоге результатом данной проверки всегда является значение `true`, а это означает, что в создании объекта `Singleton` принимают участие все потоки. Разумеется, это возможно только при условии, что все потоки вызвали метод `GetSingleton` одновременно, чего в большинстве случаев не происходит. Именно поэтому ошибка столько времени оставалась нераспознанной.

В CLR вызов любого метода запирания означает установку непреодолимого барьера на доступ к памяти: вся запись в переменные должна завершиться до этого барьера, а любое чтение переменных может начаться только после барьера. Для метода `GetSingleton` это означает, что повторное чтение поля `s_value` должно быть произведено после вызова метода `Monitor.Enter`; в процессе вызова метода значение поля нельзя сохранить в регистре.

Внутри метода `getSingleton` вызывается метод `Interlocked.Exchange`. Этим решается следующая проблема. Предположим, что вторая инструкция `if` содержит следующую строку кода:

```
s_value = new Singleton(); // Это вы написали бы в идеале
```

Можно ожидать, что компилятор создаст код, выделяющий память под объект `Singleton`, вызовет конструктор для инициализации полей данного объекта и присвоит ссылку на него полю `s_value`, чтобы это значение увидели другие потоки — это называется *публикацией* (publishing). Однако компилятор может выделить память под объект `Singleton`, назначить ссылку переменной `s_value` (выполнив публикацию) и только после этого вызвать конструктор. Если в процедуре участвует всего один поток, подобное изменение очередности операций не имеет значения. Но что произойдет, если после публикации ссылки в поле `s_value`, но до вызова конструктора другой поток вызовет метод `getSingleton`? Этот поток обнаружит, что значение поля `s_value` отлично от `null` и начнет пользоваться объектом `Singleton`, хотя его конструктор еще не закончил работу! Подобную ошибку крайне сложно отследить, особенно из-за того, что время ее появления случайно.

Эту проблему решает вызов метода `Interlocked.Exchange`. Он гарантирует, что ссылка из переменной `temp` будет опубликована в поле `s_value` только после того, как конструктор завершит свою работу. Альтернативным способом решения проблемы является пометка поля `s_value` ключевым словом `volatile`. Запись в такое волатильное (неустойчивое) поле `s_value` возможна только после завершения конструктора. К сожалению, то же самое относится ко всем процедурам чтения волатильного поля, а так как никакой необходимости в этом нет, вряд ли стоит идти на снижение производительности без полной уверенности в полезности этого.

В начале этого раздела я назвал записывание с двойной проверкой не особо интересным. С моей точки зрения, разработчики используют его гораздо чаще, чем следовало бы. В большинстве случаев оно только снижает производительность. Вот гораздо более простая версия класса `Singleton` с аналогичным предыдущей версии поведением, но без записывания с двойной проверкой:

```
internal sealed class Singleton {
    private static Singleton s_value = new Singleton();

    // Закрытый конструктор не дает коду вне данного класса
    // создавать экземпляры
    private Singleton() {
        // Код инициализации объекта Singleton
    }

    // Открытый статический метод, возвращающий объект Singleton
    // (и создающий его, если это нужно)
    public static Singleton GetSingleton() { return s_value; }
}
```

Так как CLR автоматически вызывает конструктор класса при первой попытке получить доступ к члену этого класса, при первом запросе потока к методу `GetSingleton` класса `Singleton` автоматически создается экземпляр объекта. Более того, среда CLR гарантирует безопасность в отношении потоков при вызове конструктора класса. Все это уже объяснялось в главе 8. Недостатком такого подхода является вызов конструктора типа при первом доступе к любому члену класса. То есть если в типе `Singleton` определить другие статические члены, первая же попытка доступа к любому из них приведет к появлению объекта `Singleton`. Некоторые разработчики обходят данную проблему при помощи вложенных классов.

Рассмотрим третий способ создания единственного объекта `Singleton`:

```
internal sealed class Singleton {
    private static Singleton s_value = null;
    // Закрытый конструктор не дает коду вне данного
    // класса создавать экземпляры
    private Singleton() {
        // Код инициализации объекта Singleton
    }
    // Открытый статический метод, возвращающий объект Singleton
    // (и создающий его, если это нужно)
    public static Singleton GetSingleton() {
        if (s_value != null) return s_value;
        // Создание нового объекта Singleton и превращение его в корень,
        // если этого еще не сделал другой поток
        Singleton temp = new Singleton();
        Interlocked.CompareExchange(ref s_value, temp, null);
        // При потере этого потока второй объект Singleton
        // утилизируется сборщиком мусора
        return s_value; // Возвращение ссылки на объект
    }
}
```

При одновременном вызове метода `GetSingleton` различными потоками в этой версии кода может появиться два (и более) объекта `Singleton`. Однако метод `Interlocked.CompareExchange` гарантирует публикацию в поле `s_value` только одной ссылки. Любой объект, не превращенный этим полем в корневой, будет утилизирован при первой же сборке мусора. Впрочем, в большинстве приложений практически никогда не возникает ситуация одновременного вызова метода `GetSingleton` разными потоками, поэтому там вряд ли когда-нибудь появится более одного объекта `Singleton`.

Данный код имеет массу достоинств. Во-первых, он очень быстро работает. Во-вторых, в нем никогда не блокируются потоки. Ведь когда поток из пула блокируется на классе `Monitor` или на любой другой конструкции синхронизации потоков режима ядра, пул порождает еще один поток, чтобы загрузить процессор. Выделяется и инициализируется еще мегабайт или даже больше памяти,

а все библиотеки получают уведомление о подключении нового потока. С методом CompareExchange такого никогда не происходит. Разумеется, данную технику можно использовать только при отсутствии побочных эффектов у конструктора.

В FCL существует два типа, реализующие описанные в данном разделе эталоны программирования. Вот как выглядит обобщенный класс System.Lazy (некоторые методы не показаны):

```
public class Lazy<T> {
    public Lazy(Func<T> valueFactory, LazyThreadSafetyMode mode);
    public Boolean IsValueCreated { get; }
    public T Value { get; }
}
```

А вот как он работает:

```
public static void Main() {
    // Создание оболочки отложенной инициализации для получения DateTime
    Lazy<String> s = new Lazy<String>(
        () => DateTime.Now.ToLongTimeString(),
        LazyThreadSafetyMode.PublicationOnly);

    Console.WriteLine(s.IsValueCreated); // Возвращается false, так как
                                           // запроса к Value еще не было
    Console.WriteLine(s.Value);           // Вызывается этот делегат
    Console.WriteLine(s.IsValueCreated); // Возвращается true, так как
                                           // был запрос к Value
    Thread.Sleep(10000);                  // Ждем 10 секунд и снова
                                           // выводим время
    Console.WriteLine(s.Value);           // Теперь делегат НЕ вызывается,
                                           // результат прежний
}
```

После запуска данного кода я получил:

```
False
2:40:42 PM
True
```

2:40:42 PM ← Обратите внимание, 10 секунд прошло, а время осталось прежним

Код сконструировал экземпляр класса Lazy и передал ему один из флагов LazyThreadSafetyMode. Вот как выглядят и что означают данные флаги:

```
public enum LazyThreadSafetyMode {
    None, // Безопасность в отношении потоков не
          // поддерживается (хорошо для GUI-приложений)
    ExecutionAndPublication, // Используется записывание с двойной проверкой
    PublicationOnly,         // Используется метод Interlocked.CompareExchange
}
```

В некоторых сценариях с ограниченной памятью отсутствует необходимость в создании экземпляра класса Lazy. Вместо этого можно воспользоваться

статическими методами класса `System.Threading.LazyInitializer`. Вот как он выглядит:

```
public static class LazyInitializer {  
    // Эти два метода используют Interlocked.CompareExchange  
    public static T EnsureInitialized<T>(ref T target) where T: class;  
    public static T EnsureInitialized<T>(  
        ref T target, Func<T> valueFactory) where T: class;  
  
    // Эти два метода передают syncLock в методы Enter и Exit класса Monitor  
    public static T EnsureInitialized<T>(  
        ref T target, ref Boolean initialized, ref Object syncLock);  
    public static T EnsureInitialized<T>(ref T target,  
        ref Boolean initialized, ref Object syncLock, Func<T> valueFactory);  
}
```

Возможность явно указать объект синхронизации в параметре `syncLock` метода `EnsureInitialized` позволяет одним записыванием защитить множество функций и полей инициализации.

Вот пример метода из данного класса:

```
public static void Main() {  
    String name = null;  
    // Так как имя равно null, запускается делегат и инициализирует поле имени  
    LazyInitializer.EnsureInitialized(ref name, () => "Jeffrey");  
    Console.WriteLine(name); // Выводится "Jeffrey"  
  
    // Так как имя отлично от null, делегат не запускается и имя не меняется  
    LazyInitializer.EnsureInitialized(ref name, () => "Richter");  
    Console.WriteLine(name); // Снова выводится "Jeffrey"  
}
```

Эталон условной переменной

Предположим, что некий поток выполняет код при соблюдении сложного условия. Можно просто организовать заикливание этого потока с периодической проверкой условия. Однако, во-первых, это пустая трата процессорного времени, во-вторых, невозможно атомарно проверить несколько переменных, входящих в условие. К счастью, существует эталон программирования, позволяющий потокам эффективно синхронизировать свои операции на основе сложного условия. Он называется *эталон условной переменной* (*condition variable pattern*) и применяется следующими методами класса `Monitor`:

```
public static class Monitor {  
    public static Boolean Wait(Object obj);  
    public static Boolean Wait(Object obj, Int32 millisecondsTimeout);
```

продолжение ➤


```

public static void Pulse(Object obj);
public static void PulseAll(Object obj);
}

```

Вот как выглядит данный эталон:

```

internal sealed class ConditionVariablePattern {
    private readonly Object m_lock = new Object();
    private Boolean m_condition = false;

    public void Thread1() {
        Monitor.Enter(m_lock); // Взаимоисключающее записание

        // "Атомарная" проверка сложного условия записания
        while (!m_condition) {
            // Если условие не соблюдается, ждем, что его поменяет другой поток
            Monitor.Wait(m_lock); // На время выполняем отпирание,
                                   // чтобы другой поток мог запереться
        }

        // Условие соблюдено, обрабатываем данные...

        Monitor.Exit(m_lock); // Отпирание
    }

    public void Thread2() {
        Monitor.Enter(m_lock); // Взаимоисключающее записание

        // Обрабатываем данные и изменяем условие...
        m_condition = true;

        // Monitor.Pulse(m_lock); // Будим одного ожидающего ПОСЛЕ отпирания
        Monitor.PulseAll(m_lock); // Будим всех ожидающих ПОСЛЕ отпирания

        Monitor.Exit(m_lock); // Отпирание
    }
}

```

В этом коде поток, выполняющий метод `Thread1`, входит в код взаимноисключающего записания и осуществляет проверку условия. В данном случае я всего лишь проверяю значение поля `Boolean`, но условие может быть сколь угодно сложным. К примеру, можно взять текущую дату и удостовериться, что это вторник и март. А заодно проверить, что коллекция состоит из 10 элементов. Если условие не соблюдается, поток не заикливаясь на проверке, так как это было бы напрасной тратой процессорного времени, а вызывает метод `Wait`. Данный метод выполняет отпирание, чтобы запереться мог другой поток, и приостанавливает вызывающий поток.

Метод `Thread2` содержит код, выполняемый вторым потоком. Он вызывает метод `Enter` для записания, обрабатывает какие-то данные, меняя при этом

состояние условия, после чего вызывает метод `Pulse(All)`, разблокирующий поток после вызова метода `Wait`. Метод `Pulse` разблокирует поток, ожидающий дольше всех (если такие имеются), в то время как метод `PulseAll` разблокирует все ожидающие потоки (если такие есть). Однако ни один из этих потоков пока не просыпается. Поток, выполняющий метод `Thread2`, должен вызвать метод `Monitor.Exit`, давая шанс другому потоку выполнить записание. Кроме того, в результате выполнения метода `PulseAll` потоки разблокируются не одновременно. После освобождения потока, вызвавшего метод `Wait`, он становится владельцем записания, а так как это взаимоисключающее записание, в каждый момент времени им может владеть только один поток. Другие потоки имеют шанс получить право на записание только после того, как текущий владелец вызовет метод `Wait` или `Exit`.

Проснувшись, поток, выполняющий метод `Thread1`, снова проверяет условие в цикле. Если оно все еще не соблюдено, он опять вызывает метод `Wait`. В противном случае он обрабатывает данные и, в конце концов, вызывает метод `Exit`, выполняя отпирание и давая доступ другим потокам к коду записания. Таким образом данный эталон позволяет проверить несколько формирующих сложное условие переменных при помощи простой логики синхронизации (всего одного записания), после чего без нарушения какой-либо логики разблокируется сразу несколько потоков, хотя при этом возможна напрасная трата процессорного времени.

Вот пример безопасной в отношении потоков очереди, которая заставляет несколько потоков встраивать элементы в очередь и удалять их. Обратите внимание, что потоки, пытающиеся удалить элемент из очереди, блокируются до момента, пока элемент не становится доступным для обработки.

```
internal sealed class SynchronizedQueue<T> {
    private readonly Object m_lock = new Object();
    private readonly Queue<T> m_queue = new Queue<T>();

    public void Enqueue(T item) {
        Monitor.Enter(m_lock);

        // После постановки элемента в очередь пробуждаем
        // один/все ожидающие потоки
        m_queue.Enqueue(item);
        Monitor.PulseAll(m_lock);

        Monitor.Exit(m_lock);
    }

    public T Dequeue() {
        Monitor.Enter(m_lock);

        // Выполняем цикл, пока очередь не опустеет (условие)
        while (m_queue.Count == 0)
            Monitor.Wait(m_queue);
```

```
// Удаляем элемент из очереди и возвращаем его на обработку
T item = m_queue.Dequeue();
Monitor.Exit(m_lock);
return item;
}
}
```

Сокращение времени записи при помощи коллекций

Мне не сильно нравятся все эти конструкции синхронизации потоков, использующие примитивы в режиме ядра. Ведь они нужны для блокировки потоков, в то время как создание потока обходится слишком дорого, чтобы потом он не работал. Для наглядности рассмотрим пример.

Представьте веб-сайт, к которому клиенты делают запросы. Поступивший запрос начинает обрабатываться потоком из пула. Пусть клиент хочет безопасным в отношении потоков способом изменить данные на сервере, поэтому он получает записание на чтение-запись для записи. Представим, что записание длится долго. За это время успевает прийти еще один клиентский запрос, для которого пул создает новый поток, пытающийся получить записание на чтение-запись для чтения. Запросы продолжают поступать, пул создает дополнительные потоки, и все эти потоки блокируются. Все свое время сервер занимается созданием потоков и не может остановиться! Такой сервер вообще не может нормально масштабироваться.

Все становится только хуже, когда поток записи выполняет отпирание, и одновременно запускаются все заблокированные потоки чтения. В результате относительно небольшому количеству процессоров нужно как-то обработать все это множество потоков. Windows попадает в ситуацию постоянного переключения контекста. Такая нагрузка отрицательно сказывается на производительности, и указанный объем работы выполняется не так быстро, как мог бы.

Многие из проблем, решаемые при помощи описанных в этой главе конструкций, намного успешнее решаются средствами класса `Task`, рассмотренного в главе 26. К примеру, возьмем класс `Barrier`: для работы на каждом этапе можно было бы создать группу заданий (объектов `Task`), а после их завершения ничто не мешает нам продолжить работу с дополнительными объектами `Task`. Такой подход имеет целый ряд преимуществ в сравнении с конструкциями, описанными в этой главе:

- ❑ Задания требуют меньше памяти, чем потоки, кроме того, они намного быстрее создаются и уничтожаются.
- ❑ Пул потоков автоматически распределяет задания среди доступных процессоров.

- ❑ По мере того как каждое задание завершает свой этап, выполнявший его поток возвращается в пул, где может заняться другой работой, если таковая имеется.
- ❑ Пул потоков видит все задания сразу и поэтому может лучше планировать их выполнение, сокращая количество потоков в процессе, а значит, и количество переключений контекста.

Запираание на чтение-запись весьма популярно и часто применяется¹. Попытка добиться аналогичной функциональности с помощью объектов Task — задача нетривиальная. Тем не менее в моей библиотеке Power Threading есть неблокирующий класс чтения-записи, который я назвал ReaderWriterGate. Вот как он выглядит (некоторые методы не показаны):

```
public sealed class ReaderWriterGate : IDisposable {  
    public ReaderWriterGate();  
    public void Dispose();  
    public IAsyncResult BeginRead(ReaderWriterGateCallback callback,  
        Object state, AsyncCallback asyncCallback, Object asyncState);  
    public Object EndRead(IAsyncResult result);  
    public IAsyncResult BeginWrite(ReaderWriterGateCallback callback,  
        Object state, AsyncCallback asyncCallback, Object asyncState);  
    public Object EndWrite(IAsyncResult result);  
}
```

Делегат ReaderWriterGateCallback при этом выглядит так:

```
public delegate object ReaderWriterGateCallback(  
    ReaderWriterGateReleaser releaser);
```

А вот класс ReaderWriterGateReleaser (некоторые методы не показаны):

```
public sealed class ReaderWriterGateReleaser : IDisposable {  
    public Object State { get; } // Возвращает 'состояние',  
                                // переданное в BeginRead/BeginWrite  
    public void Dispose();  
}
```

Класс ReaderWriterGate придает запираанию вид асинхронной операции ввода-вывода. Более того, мой класс даже предлагает методы BeginRead и BeginWrite, принимающие делегат AsyncCallback и возвращающие объект IAsyncResult, а также методы EndRead и EndWrite, принимающие объект IAsyncResult. Класс сконструирован таким образом, что работает как модель асинхронного программирования (см. главу 27).

Вот как он работает. Поместите требующий доступа на чтение код в отдельный метод и передайте методу BeginRead в качестве параметра делегат ReaderWriterGetCallback. Когда чтение ресурса станет безопасным, объект Reader-

¹ Разумеется, при необходимости взаимноисключающего доступа к ресурсу всегда можно воспользоваться этим запираанием и запрашивать доступ к защищаемому им ресурсу только на запись.

WriterGate заставит поток пула вызвать ваш метод. Однако имейте в виду, что методы, осуществляющие чтение ресурса, могут одновременно выполняться несколькими потоками пула. Код, требующий доступа на запись, также поместите в отдельный метод и передайте методу BeginWrite в качестве параметра делегат ReaderWriterGetCallback. Когда запись в ресурс станет безопасной, объект ReaderWriterGate заставит поток пула вызвать ваш метод. При этом объект ReaderWriterGate гарантирует наличие в каждый момент времени всего одного потока, который выполняет метод, записывающий что бы то ни было в ресурс.

Когда ваш метод, завершив запись, вернет управление, объект ReaderWriterGate вызовет метод, переданный вами в параметр asyncCallback; именно на этом этапе вы узнаете, что операция завершена.

Вернемся к обсуждению нашего веб-сервера. Клиент сделал запрос на запись в ресурс, поэтому поток пула вызывает метод BeginWrite класса ReaderWriterGate. В ходе обработки потоком метода обратного вызова приходит следующий клиентский запрос. Пул создает новый поток, вызывающий метод BeginRead. Объект ReaderWriterGate видит, что в данный момент чтение невозможно, и добавляет делегат callback во внутреннюю очередь. Эта очередь представляет собой коллекцию и должна управляться в безопасном в отношении потоков режиме, поэтому она запирается. Однако запираение длится, только пока происходит добавление в очередь новых элементов и удаление их оттуда, а эти операции происходят быстро. То есть другие потоки, использующие объект ReaderWriterGate, надолго не блокируются, если это вообще случается!

После добавления в очередь делегата callback поток возвращается в пул. По мере поступления все новых клиентских запросов этот поток просыпается и вызывает метод BeginRead, добавляя во внутреннюю очередь все новые и новые делегаты. В итоге для указанной работы серверу достаточно всего двух потоков.

Закончив запись в ресурс, первый поток возвращает управление от метода обратного вызова объекту ReaderWriterGate. Этот объект обнаруживает множество делегатов во внутренней очереди и отправляет их все в пул потоков среды CLR. В процессе этих манипуляций очередь запирается, но только на время переправки в пул делегатов callback. Пул, в свою очередь, передает работу ядрам, используя для этого потоки, количество которых не превышает количества ядер, что позволяет избежать переключений контекста¹. В итоге мы получаем высокопроизводительный и масштабируемый сервер, использующий небольшое количество ресурсов.

Объект ReaderWriterGate имеет для делегатов другую внутреннюю очередь, ориентированную на запись в ресурс. При поступлении запроса на запись все входящие делегаты чтения ставятся в очередь таким образом, чтобы текущие

¹ Предполагается, что другие потоки на машине отсутствуют, что верно большую часть времени. Ведь большинство компьютеров работает далеко не со 100-процентной загрузкой процессоров. Но даже при полной загрузке система будет функционировать описанным образом, если работающие потоки имеют низкий приоритет. Наличие других потоков приводит к переключениям контекста. Это плохо с точки зрения производительности, но хорошо в плане надежности. Помните, что Windows выделяет каждому процессу, по крайней мере, один поток и прибегает к переключениям контекста, чтобы даже при зависании одного приложения остальные могли продолжить работу.

методы чтения могли завершиться и уйти. После обработки потоком пула всех ранее поставленных в очередь запросов на чтение в очередь пула ставится единственный делегат на запись. Именно так гарантируется, что в каждый момент времени метод записи будет вызван одним и только одним потоком.

Дополнительная информация об объекте `ReaderWriterGate` находится по адресу <http://msdn.microsoft.com/ru-ru/magazine/cc163532.aspx>. После того как у меня появилась данная идея, я продал патентные права Microsoft и в 2009 году получил от патентного офиса патент номер 7 603 502. Однако несмотря на наличие у Microsoft патента, в FCL отсутствуют реализующие данную идею классы. Реализацию я написал самостоятельно и поместил ее в библиотеку `Power Threading`.

Продав права компании Microsoft, я получил лицензию, позволяющую клиентам `Wintellect` использовать это «изобретение» с условием работать только на платформе Microsoft¹. Загружая библиотеку с сайта `Wintellect`, вы становитесь клиентом нашей компании и получаете право работать с библиотекой с соблюдением лицензионных ограничений.

В главе 27 я кратко описал мой класс `AsyncEnumerator`, также входящий в библиотеку `Power Threading`. Он позволяет использовать модель синхронного программирования с классами, поддерживающими APM в CRL. Так как мой объект `ReaderWriterGate` поддерживает модель асинхронного программирования, его можно применять с классом `AsyncEnumerator`. Попытавшись сделать это в первый раз, я обнаружил возможность еще больше упростить модель программирования. Так что если вы решите воспользоваться моим классом `AsyncEnumerator`, а также получить набор итераторов, осуществляющих безопасный в отношении потоков доступ к общим данным, возьмите вместо класса `ReaderWriterGate` класс `SyncGate`. Его вы тоже найдете в библиотеке `Power Threading`. Вот как он выглядит:

```
public sealed class SyncGate {
    public SyncGate();
    public void BeginRegion(SyncGateMode mode,
        AsyncCallback asyncCallback, Object asyncState);
    public void EndRegion(IAsyncResult result);
}
```

За примерами совместного использования данных классов обратитесь к библиотеке `Power Threading`. Впрочем, короткий пример есть и в этом разделе. Я взял показанный в главе 27 метод `PipeServerAsyncEnumerator` и отредактировал его, заставив записывать время последнего клиентского запроса в доступное для всех потоков статическое поле. Из-за возможности одновременного выполнения различных клиентских запросов разными потоками обновление статического поля должно выполняться в безопасном в отношении потоков режиме. Для этого

¹ Silverlight считается платформой Microsoft, даже если вы запускаете Silverlight-приложение под управлением другой операционной системы. В библиотеке `Power Threading` присутствует версия данного класса, пригодная как для Silverlight, так и для Microsoft .NET Compact Framework.

я создал поле, хранящее ссылку на объект SyncGate, а для получения эксклюзивного доступа и отказа от него я воспользовался методами BeginRegion и EndRegion соответственно. Вот новая версия кода, в которой добавленные строки выделены:

```
// В это поле записывается время последнего клиентского запроса
private static DateTime s_lastClientRequestTimestamp = DateTime.MinValue;

// SyncGate обеспечивает безопасный в отношении потоков
// доступ к полю s_lastClientRequestTimestamp
private static readonly SyncGate s_gate = new SyncGate();

private static IEnumerator<Int32> PipeServerAsyncEnumerator(
    AsyncEnumerator ae) {

    // Каждый сервер выполняет в канале асинхронные операции
    using (var pipe = new NamedPipeServerStream(
        "Echo", PipeDirection.InOut, -1, PipeTransmissionMode.Message,
        PipeOptions.Asynchronous | PipeOptions.WriteThrough)) {

        // Асинхронно принимаем соединение с клиентом
        pipe.BeginWaitForConnection(ae.End(), null);
        yield return 1;

        // Клиент подсоединен, примем еще одного клиента
        var aeNewClient = new AsyncEnumerator();
        aeNewClient.BeginExecute(
            PipeServerAsyncEnumerator(aeNewClient), aeNewClient.EndExecute);

        // Принятие клиентского соединения
        pipe.EndWaitForConnection(ae.DequeueAsyncResult());

        // Асинхронное чтение клиентского запроса
        Byte[] data = new Byte[1000];
        pipe.BeginRead(data, 0, data.Length, ae.End(), null);
        yield return 1;

        // Обработка клиентского запроса
        Int32 bytesRead = pipe.EndRead(ae.DequeueAsyncResult());

        // Получаем временную метку клиентского запроса
        DateTime now = DateTime.Now;

        // Мы хотим сохранить время последнего клиентского запроса
        // Так как возможны одновременные запросы, нужен
        // безопасный в отношении потоков режим
        s_gate.BeginRegion(SyncGateMode.Exclusive, ae.End()); // Эксклюзивный
                                                                // доступ запроса
        yield return 1; // После эксклюзивного доступа итератор
                        // начинает новый отсчет
```

```

if (s_lastClientRequestTimestamp < now)
    s_lastClientRequestTimestamp = now;

s_gate.EndRegion(ae.DequeueAsyncResult()); // Отказ от эксклюзивного
                                           // доступа

// В моем примере просто меняется регистр букв
// Но вы можете вставить сюда любую вычислительную операцию
data = Encoding.UTF8.GetBytes(
    Encoding.UTF8.GetString(data, 0, bytesRead).ToUpper().ToCharArray());

// Асинхронная отправка ответа клиенту
pipe.BeginWrite(data, 0, data.Length, ae.End(), null);
yield return 1;
// Ответ клиенту отправлен, закрываем соединение со своей стороны
pipe.EndWrite(ae.DequeueAsyncResult());
} // Закрытие канала
}

```

Классы коллекций для параллельной обработки потоков

В FCL существует четыре безопасных в отношении потоков класса коллекций, принадлежащих пространству имен `System.Collections.Concurrent`. Классы `ConcurrentQueue`, `ConcurrentStack` и `ConcurrentDictionary` находятся в библиотеке **mscorlib.dll**, а класс `ConcurrentBag` — в библиотеке **System.dll**. Вот как выглядят чаще всего используемые члены:

```

// Обработка элементов по алгоритму FIFO
public class ConcurrentQueue<T> : IProducerConsumerCollection<T>,
    IEnumerable<T>, ICollection, IEnumerable {

    public ConcurrentQueue();
    public void Enqueue(T item);
    public Boolean TryDequeue(out T result);
    public Int32 Count { get; }
    public IEnumerator<T> GetEnumerator();
}

// Обработка элементов по алгоритму LIFO
public class ConcurrentStack<T> : IProducerConsumerCollection<T>,
    IEnumerable<T>, ICollection, IEnumerable {

    public ConcurrentStack();
    public void Push(T item);
    public Boolean TryPop(out T result);
}

```



```

    public Int32 Count { get; }
    public IEnumerator<T> GetEnumerator();
}

// Несортированный набор элементов, допускающий дублирование
public class ConcurrentBag<T> : IProducerConsumerCollection<T>,
    IEnumerable<T>, ICollection, IEnumerable {

    public ConcurrentBag();
    public void Add(T item);
    public Boolean TryTake(out T result);
    public Int32 Count { get; }
    public IEnumerator<T> GetEnumerator();
}

// Несортированный набор пар ключ/значение
public class ConcurrentDictionary<TKey, TValue> : IDictionary<TKey, TValue>,
    ICollection<KeyValuePair<TKey, TValue>>, IEnumerable<KeyValuePair<TKey,
    TValue>>, IDictionary, ICollection, IEnumerable {

    public ConcurrentDictionary();
    public Boolean TryAdd(TKey key, TValue value);
    public Boolean TryGetValue(TKey key, out TValue value);
    public TValue this[TKey key] { get; set; }
    public Boolean TryUpdate(
        TKey key, TValue newValue, TValue comparisonValue);
    public Boolean TryRemove(TKey key, out TValue value);
    public TValue AddOrUpdate(
        TKey key, TValue addValue, Func<TKey, TValue> updateValueFactory);
    public TValue GetOrAdd(TKey key, TValue value);
    public Int32 Count { get; }
    public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator();
}

```

Эти классы коллекций являются неблокирующими. При попытке извлечь несуществующий элемент поток немедленно возвращает управление, а не блокируется, ожидая появления элемента. Именно поэтому такие методы, как TryDequeue, TryPop, TryTake и TryGetValue, при получении элемента возвращают значение true.

Хотя эти коллекции являются неблокирующими, это вовсе не означает, что они обходятся без записи. Класс ConcurrentDictionary внутренне использует класс Monitor, но записание длится только короткое время, необходимое для манипулирования элементом коллекции. В то же время классы ConcurrentQueue и ConcurrentStack для манипулирования коллекцией используют методы Interlocked и поэтому обходятся вообще без записи. Один объект ConcurrentBag внутренне состоит из объекта мини-коллекций для каждого потока. При добавлении нового элемента методы Interlocked помещают его

в мини-коллекцию вызывающего потока. При попытке извлечь элемент его наличие опять же проверяется в мини-коллекции вызывающего потока. При обнаружении элемента задействуется метод класса `Interlocked`. Если же элемент в рассматриваемой мини-коллекции отсутствует, методы класса `Monitor` извлекают его из мини-коллекции другого потока. Мы говорим, что имеет место *захват* (stealing) элемента у другого потока.

Обратите внимание, что все рассматриваемые классы обладают методом `GetEnumerator`, обычно используемым в инструкции `foreach`, но допустимым и в языке `LINQ`. Для классов `ConcurrentStack`, `ConcurrentQueue` и `ConcurrentBag` метод `GetEnumerator` создает снимок содержимого коллекции и возвращает зафиксированные элементы; при этом реальное содержимое коллекции уже может измениться. Метод `GetEnumerator` класса `ConcurrentDictionary` не фиксирует содержимое коллекции, а значит, в процессе просмотра словаря его вид может поменяться; об этом следует помнить. Свойство `Count` возвращает количество элементов в коллекции на момент запроса. Если другие потоки в это время добавляют элементы в коллекцию или извлекают их оттуда, возвращенное значение может оказаться неверным.

Классы `ConcurrentStack`, `ConcurrentQueue` и `ConcurrentBag` реализуют интерфейс `IProducerConsumerCollection`, который выглядит следующим образом:

```
public interface IProducerConsumerCollection<T> : IEnumerable<T>,
    ICollection, IEnumerable {
    Boolean TryAdd(T item);
    Boolean TryTake(out T item);
    T[] ToArray();
    void CopyTo(T[] array, Int32 index);
}
```

Любой реализующий данный интерфейс класс может превратиться в блокирующую коллекцию. Поток, добавляющий элементы, блокируется, если коллекция уже заполнена, а поток, удаляющий элементы, блокируется, если она пуста. Разумеется, я, по возможности, стараюсь избегать таких коллекций, ведь они предназначены именно для блокировки потоков. Для преобразования коллекции в блокирующую создается класс `System.Collections.Concurrent.BlockingCollection`, конструктору которого передается ссылка на не блокирующую коллекцию. Этот класс (определенный в сборке `System.dll`) выглядит следующим образом (некоторые методы не показаны):

```
public class BlockingCollection<T> : IEnumerable<T>, ICollection,
    IEnumerable, IDisposable {
    public BlockingCollection(
        IProducerConsumerCollection<T> collection, Int32 boundedCapacity);

    public void Add(T item);
    public Boolean TryAdd(
        T item, Int32 msTimeout, CancellationToken cancellationToken);
```

```

public void CompleteAdding();

public T Take();
public Boolean TryTake(
    out T item, Int32 msTimeout, CancellationToken cancellationToken);

public Int32 BoundedCapacity { get; }
public Int32 Count { get; }
public Boolean IsAddingCompleted { get; } // true, если вызван метод
                                           // AddingComplete
public Boolean IsCompleted { get; }       // true, если вызван метод
                                           // IsAddingComplete и Count==0

public IEnumerable<T> GetConsumingEnumerable(
    CancellationToken cancellationToken);

public void CopyTo(T[] array, int index);
public T[] ToArray();
public void Dispose();
}

```

При конструировании класса `BlockingCollection` параметр `boundedCapacity` показывает максимально допустимое количество элементов коллекции. Если поток вызывает метод `Add` для уже заполненной коллекции, он блокируется. Впрочем, поток может вызвать метод `TryAdd`, передав ему время задержки (в миллисекундах) и/или объект `CancellationToken`. В результате поток блокируется до добавления элемента, окончания времени ожидания или отмены объекта `CancellationToken` (класс `CancellationToken` подробно рассматривался в главе 26).

Класс `BlockingCollection` реализует интерфейс `IDisposable`. В итоге метод `Dispose` вызывается для основной коллекции и удаляет заодно два объекта `SemaphoreSlim`, используемые классом для блокировки потоков-производителей и потоков-потребителей.

Завершив добавление элементов в коллекцию, поток-производитель должен вызвать метод `CompleteAdding`. Это даст понять потоку-потребителю, что больше элементов не будет и цикл `foreach`, использующий объект `GetConsumingEnumerable`, завершится. Показанный далее код демонстрирует, как задается сценарий работы производителя/потребителя и завершающего сигнала:

```

public static void Main() {
    var bl = new BlockingCollection<Int32>(new ConcurrentQueue<Int32>());

    // Поток пула получает элементы
    ThreadPool.QueueUserWorkItem(ConsumeItems, bl);

    // Добавляем в коллекцию 5 элементов
    for (Int32 item = 0; item < 5; item++) {

```

```
    Console.WriteLine("Producing: " + item);
    bl.Add(item);
}

// Информировать поток-потребитель, что больше элементов не будет
bl.CompleteAdding();

Console.ReadLine(); // Для целей тестирования
}

private static void ConsumeItems(Object o) {
    var bl = (BlockingCollection<Int32>) o;
    // Блокируем до получения элемента, затем обрабатываем его
    foreach (var item in bl.GetConsumingEnumerable()) {
        Console.WriteLine("Consuming: " + item);
    }

    // Коллекция пуста и там больше не будет элементов
    Console.WriteLine("All items have been consumed");
}
```

После выполнения данного кода я получил:

```
Producing: 0
Producing: 1
Producing: 2
Producing: 3
Producing: 4
Consuming: 0
Consuming: 1
Consuming: 2
Consuming: 3
Consuming: 4
All items have been consumed
```

Если вы попытаете запустить этот код, строчки **Producing** (производство) и **Consuming** (потребление) могут поменять свой порядок, но строка **All items have been consumed** (все элементы потреблены) всегда будет замыкать список вывода.

Класс `BlockingCollection` обладает также статическими методами `AddToAny`, `TryAddToAny`, `TakeFromAny` и `TryTakeFromAny`. Все они принимают в качестве параметров коллекцию `BlockingCollection<T>[]`, а кроме того, элемент, время ожидания и объект `CancellationToken`. Методы `(Try)AddToAny` циклически просматривают все коллекции в массиве, пока не обнаруживают коллекцию, способную принять новый элемент. Методы `(Try)TakeFromAny` циклически просматривают все коллекции до обнаружения той, из которой можно извлечь элемент.

Джеффри Рихтер
**CLR via C#. Программирование на платформе Microsoft .NET
Framework 4.0 на языке C#**
3-е издание

Перевели с английского И. Радченко, И. Рузмайкина

Заведующий редакцией
Руководитель проекта
Ведущий редактор
Литературный редактор
Художественный редактор
Корректор
Верстка

*А. Кривцов
А. Юрченко
Ю. Сергиенко
А. Жданов
Л. Адиевская
В. Листова
С. Романов*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 18.11.11. Формат 70х100/16. Усл. п. л. 74,820. Тираж 1500. Заказ 26875.

Отпечатано по технологии СtP в ОАО «Первая Образцовая типография», обособленное подразделение «Печатный двор».
197110, Санкт-Петербург, Чкаловский пр., 15.