Кристин

Нельзя передать словами мое отношение к нашей совместной жизни. Я ценю нашу семью и наши приключения. Каждый день моей жизни наполнен любовью к тебе.

Эйдан

Ты стал вдохновением для меня и научил меня играть и развлекаться.

Наблюдать за тем, как ты растешь, — это радость и награда. Я рад жить с тобой под одной крышей — рядом с тобой я становлюсь лучше.

Jeffrey Richter

CLR via C#

Microsoft® Press

Джеффри Рихтер

CLR via C#

Программирование на платформе Microsoft® .NET FRAMEWORK 2.0 на языке C#

2-е издание, исправленное

МАСТЕР-КЛАСС

РУССКАЯ РЕДАКЦИЯ ПИТЕР®

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

2008

УДК 004.45
ББК 32.973.26–018.2
Р49

Рихтер Дж.

Р49 CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Мастер-класс. / Пер. с англ. — 2-е изд., исправ. — М. : Издательство «Русская Редакция» ; СПб. : Питер , 2008. — 656 стр. : ил.

ISBN 978-5-7502-0348-2 («Русская Редакция»)
ISBN 978-5-91180-303-2 («Питер»)

Эта книга — подробное описание внутреннего устройства и функционирования общеязыковой исполняющей среды (CLR) Microsoft .NET Framework версии 2.0. В ней раскрыта система типов .NET Framework и разъяснены способы управления ими. Представлены концепции программирования с широким использованием библиотеки FCL, относящиеся ко всем языкам, ориентированным на работу с .NET Framework. Особое внимание уделено обобщениям, управлению асинхронными операциями и синхронизации потоков. Книга ориентирована на разработчиков любых видов приложений на платформе с .NET Framework: Windows Forms, Web Forms, Web-сервисов, консольных приложений и пр.

Второе издание книги выпущено с учетом отзывов читателей и исправлений автора.

УДК 004.45
ББК 32.973.26–018.2

Подготовлено к печати по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США.

ISBN 0-7356-2163-2 (англ.)
ISBN 978-5-7502-0348-2 («Русская Редакция»)
ISBN 978-5-91180-303-2 («Питер»)

Оглавление

Введение

За прошедшие годы Microsoft выпустила массу технологий, призванных облегчить создание архитектуры и реализацию исходного кода приложений. Многие технологии предусматривают абстрагирование, которое позволяет разработчикам сосредоточиться на решении предметных задач, меньше думая об особенностях аппаратного обеспечения и операционных систем. Вот всего лишь несколько примеров таких технологий.

- **Библиотека Microsoft Foundation Class (MFC)** — уровень абстрагирования, служащий в языке C++ для программирования графического пользовательского интерфейса. Используя MFC, разработчики могут больше внимания уделить самой программе и меньше заниматься циклами обработки сообщений, оконными процедурами, классами окон и т. п.
- **Microsoft Visual Basic 6 и более ранние версии** служили разработчикам абстракцией, облегчающей создание приложений с графическим пользовательским интерфейсом. Эта технология абстрагирования служила практически тем же целям, что и MFC, но ориентировалась на программистов, пишущих на Basic, — требовался другой подход к различным аспектам программирования графического интерфейса.
- **Active Server Pages (ASP)** служила для абстрагирования при создании активных и динамических Web-сайтов с использованием Visual Basic Script или JScript. Она позволила разработчикам абстрагироваться от особенностей сетевых взаимодействий и больше внимания уделять содержанию Web-страниц.
- **Библиотека Active Template Library (ATL)** — уровень абстрагирования, облегчающий создание компонентов, которые доступны для использования специалистами, работающими с различными языками программирования.

Как видите, все эти технологии абстрагирования создавались, чтобы разработчики могли забыть о технических деталях и сосредоточиться на конкретных вещах, будь то приложения с графическим пользовательским интерфейсом, Web-приложения или компоненты. Если требовалось создать Web-сайт, на котором использовался определенный компонент, разработчику приходилось осваивать несколько технологий: ASP и ATL. Более того, нужно было знать многие языки программирования, так как для ASP требовался Visual Basic Script или JScript, а для ATL — C++. Несмотря на то, что эти технологии значительно облегчали работу, они требовали от программиста осваивать массу материала. Часто случалось так, что различные технологии разрабатывались без расчета на совместное использование, и разработчики сталкивались с необходимостью решать непростые проблемы интеграции.

Другая цель .NET Framework — предоставить разработчикам возможность создавать код на любимом языке по собственному выбору. Теперь можно создать и

Web-сайт, и его компоненты, используя один язык, например Visual Basic или сравнительно новый, предлагаемый Microsoft язык C#.

Единая модель программирования, API-интерфейс и язык программирования — большой шаг вперед в области технологий абстрагирования и огромная помощь разработчикам в их нелегкой работе. Но это далеко не все — все эти функции означают, что в прошлом остались проблемы интеграции, что значительно упростило тестирование, развертывание, администрирование, управление версиями и повторное использование и переориентацию кода на выполнение других задач. Уже несколько лет я использую .NET Framework и должен сказать с уверенностью, что ни за что не вернусь к устаревшим технологиям абстрагирования и способам разработки ПО. И, если меня заставят, я предпочту сменить профессию! Вот как трудно отвыкать от хорошего. Честно говоря, вспоминая, чего стоило создавать приложения с использованием старых технологий, я просто не могу представить, как разработчикам вообще удавалось так долго создавать работающее ПО.

Платформа разработки: .NET Framework

.NET Framework состоит из двух частей: общеязыковой исполняющей среды (common language runtime, CLR) и библиотеки классов (Framework Class Library, FCL). CLR предоставляет модель программирования, используемую во всех типах приложений. У CLR собственный загрузчик файлов, диспетчер памяти (сборщик мусора), система безопасности (безопасность доступа к коду), пул потоков и другое. Кроме того, CLR предоставляет объектно-ориентированную модель программирования, определяющую, как выглядят и ведут себя типы и объекты.

FCL предоставляет объектно-ориентированный API-интерфейс, используемый всеми моделями приложений. В ней содержатся определения типов, которые позволяют разработчикам выполнять ввод/вывод, планирование задач в других потоках, создавать графические образы, сравнивать строки и т. п. Естественно, что все эти определения типов соответствуют существующей CLR в модели программирования.

Microsoft выпустила три версии .NET Framework:

- версия 1.0 выпущена в 2002 году и содержит версию 7.0 компилятора C# от Microsoft;
- версия 1.1 выпущена в 2003 году и содержит версию 7.1 компилятора C# от Microsoft;
- версия 2.0 выпущена в 2005 году и содержит версию 8.0 компилятора C# от Microsoft.

В этой книге используется исключительно .NET Framework версии 2.0 и компилятор C# версии 8.0. Поскольку при выпуске новой версии .NET Framework компания Microsoft старается обеспечить максимальную обратную совместимость, многое из сказанного верно в отношении более ранних версий, но я не следил за верностью тех или иных утверждений в отношении к той или иной версии.

.NET Framework версии 2.0 поддерживает 32-разрядные версии x86, а также 64-разрядные версии x64 и IA64 ОС Windows. Для КПК (с ОС Windows CE) и небольших устройств предоставляется «облегченная» версия .NET Framework по имени .NET Compact Framework. 13 декабря 2001 года Европейская ассоциация по стан-

дартизации информационных и вычислительных систем (European Computer Manufacturers Association, ECMA) приняла в качестве стандартов язык программирования C#, определенные компоненты CLR и FCL. Документация по стандартам позволила другим организациям создать версии этих технологий, соответствующие стандартам ECMA для широкого круга ОС и процессоров. На самом деле, львиная доля этой книги собственно посвящена описанию этих стандартов, поэтому многим книга окажется полезной при работе с реализацией приложений и библиотек, совместимых со спецификацией ECMA. Тем не менее книга в первую очередь ориентирована на реализацию этого стандарта компанией Microsoft для клиентских и серверных систем.

В отличие от более ранних версий Windows, Microsoft Windows Vista поставляется с .NET Framework версии 2.0. Если требуется, чтобы приложение .NET Framework работало в более ранних версиях Windows, придется установить эту инфраструктуру вручную. К счастью, Microsoft предоставляет версию .NET Framework для свободного распространения с приложениями.

Microsoft .NET Framework позволяет разработчикам в гораздо большей степени задействовать готовые технологии, чем предыдущие платформы разработки от Microsoft. В частности, .NET Framework предоставляет реальные возможности повторного использования кода, управления ресурсами, многоязыковой разработки, безопасности, развертывания и администрирования. При проектировании этой новой платформы Microsoft учла недостатки существующих Windows-платформ. Вот далеко не полный список преимуществ CLR и FCL.

- **Единая программная модель** В отличие от существующего подхода, когда одни функции ОС доступны через процедуры динамически подключаемых библиотек (DLL), а другие — через COM-объекты, весь прикладной сервис представлен общей объектно-ориентированной программной моделью.
- **Упрощенная модель программирования CLR** Избавляет от работы с разными потаенными структурами, как это было с Win32 и COM. Так, разработчику не нужно разбираться с реестром, глобальными уникальными идентификаторами (GUID), *IUnknown*, *AddRef*, *Release*, HRESULT и т. д. CLR не просто позволяет разработчику абстрагироваться от этих концепций — их просто нет в CLR в каком бы то ни было виде. Конечно, если вы хотите написать приложение .NET Framework, которое взаимодействует с существующим не-.NET кодом, вам нужно разбираться во всех этих концепциях.
- **Отсутствие проблем с версиями** Все Windows-разработчики знают о проблемах совместимости версий, известных под названием «ад DLL». Этот «ад» возникает, когда компоненты, устанавливаемые для нового приложения, заменяют компоненты старого приложения, и в итоге последнее начинает вести себя странно или перестает работать. Архитектура .NET Framework позволяет изолировать прикладные компоненты, так что приложение всегда загружает компоненты, с которыми оно строилось и тестировалось. Если приложение работает после начальной установки, оно будет работать всегда.
- **Упрощенное развертывание** Сегодня Windows-приложения очень трудно устанавливать и разворачивать: обычно нужно создать массу файлов, параметров реестра и ярлыков. К тому же полностью удалить приложение практически невозможно. В Windows 2000 Microsoft представила новый механизм установ-

ки, решающий многие проблемы, но по-прежнему остается вероятность, что его потребители не все сделают правильно. С приходом .NET Framework все эти проблемы остаются в прошлом. Компоненты .NET Framework не связаны с реестром. По сути установка приложений .NET Framework сводится лишь к копированию файлов в нужные каталоги и созданию ярлыков в меню Start (Пуск), на рабочем столе или на панели быстрого запуска задач. Удаление же приложений сводится к удалению файлов.

- **Работа на многих платформах** При компиляции кода для .NET Framework компилятор генерирует код на *общем промежуточном языке* (common intermediate language, CIL), а не традиционный код, состоящий из процессорных команд. При исполнении CLR транслирует CIL в команды процессора. Поскольку трансляция выполняется в период выполнения, генерируются команды конкретного процессора. Это значит, что вы можете развертывать свое приложение .NET Framework на любой машине, где работает версия CLR и FCL, соответствующая стандарту ECMA: с архитектурой x86, x64, IA64 и т. д. Пользователи оценят такую возможность при переходе с одной аппаратной платформы или ОС к другой.
- **Интеграция языков программирования** COM поддерживает взаимодействие разных языков — .NET Framework обеспечивает интеграцию разных языков, то есть один язык может использовать типы, созданные на других языках. Например, CLR позволяет создать на C++ класс, производный от класса, реализованного на Visual Basic. В CLR это возможно из-за наличия общей системы типов (Common Type System, CTS), которую должны использовать все языки, ориентированные на CLR. *Общеязыковая спецификация* (Common Language Specification, CLS) определяет правила, которым должны следовать разработчики компиляторов, чтобы их языки интегрировались с другими. Сама Microsoft предлагает несколько таких языков: C++/CLI (C++ с управляемыми расширениями), C#, Visual Basic .NET и JScript. Кроме того, другие компании и учебные заведения создают компиляторы других языков, совместимых с CLR.
- **Упрощенное повторное использование кода** Все описанные выше механизмы позволяют создавать собственные классы, предоставляющие сервис сторонним приложениям. Теперь многократное использование кода становится исключительно простым и создается большой рынок готовых компонентов (типов).
- **Автоматическое управление памятью (сбор мусора)** Программирование требует большого мастерства и дисциплины, особенно когда речь идет об управлении использованием ресурсов (файлов, памяти, пространства экрана, сетевых соединений, ресурсов баз данных и прочих). Одна из самых распространенных ошибок — небрежное отношение к освобождению этих ресурсов, что может привести к некорректному выполнению программы в непредсказуемый момент. CLR автоматически отслеживает использование ресурсов, гарантируя, что не произойдет их утечки. По сути она исключает возможность явного «освобождения» памяти. В главе 20 я подробно опишу работу сборщика мусора.
- **Проверка безопасности типов** CLR может проверять безопасность использования типов в коде, что гарантирует корректное обращение к существующим типам. Если входной параметр метода объявлен как 4-байтное значение, CLR

обнаружит и предотвратит передачу 8-байтного значения в качестве значения этого параметра. Безопасность типов также означает, что управление может передаваться только в заранее известные точки (точки входа методов). Невозможно указать произвольный адрес и заставить программу исполняться, начиная с этого адреса. Совокупность всех этих защитных мер избавляет от многих распространенных программных ошибок (например, от возможности использования переполнения буфера для «взлома» программы).

- **Развитая поддержка отладки** Поскольку CLR используется для многих языков, можно написать отдельный фрагмент программы на языке, наиболее подходящем для конкретной задачи, — CLR полностью поддерживает отладку многоязыковых приложений.
- **Единый принцип обработки сбоев** Один из самых неприятных моментов Windows-программирования — несогласованный стиль сообщений о сбоях. Одни функции возвращают коды состояний Win32, другие — HRESULT, третьи генерируют исключения. В CLR обо всех сбоях сообщается через исключения, которые позволяют отделить код, необходимый для восстановления после сбоя, от основного алгоритма. Такое разделение облегчает написание, чтение и сопровождение программ. Кроме того, исключения работают в многомодульных и многоязыковых приложениях. И в отличие от кодов состояний и HRESULT исключения нельзя проигнорировать. CLR также предоставляет встроенные средства анализа стека, заметно упрощающие поиск фрагментов, вызывающих сбои.
- **Безопасность** Традиционные системы безопасности обеспечивают управление доступом на основе учетных записей пользователей. Это проверенная модель, но она подразумевает, что любому коду можно доверять в одинаковой степени. Такое допущение оправданно, когда весь код устанавливается с физических носителей (например, с компакт-диска) или с доверенных корпоративных серверов. Но по мере увеличения объема мобильного кода, например Web-сценариев, приложений, загружаемых из Интернета, и вложений, содержащихся в электронной почте, нужен ориентированный на код способ контроля за поведением приложений. Такой подход реализован в модели безопасности доступа к коду.
- **Взаимодействие с существующим кодом** В Microsoft понимают, что разработчики накопили огромный объем кода и компонентов. Переписывание всего этого кода, так чтобы он задействовал все достоинства .NET Framework, значительно замедлило бы переход к этой платформе. Поэтому в .NET Framework реализована полная поддержка доступа к COM-компонентам и Win32-функциям в существующих DLL.

Конечные пользователи не смогут самостоятельно оценить CLR и ее возможности, но они обязательно заметят качество и возможности приложений, построенных для CLR. Кроме того, руководство вашей компании оценит тот факт, что CLR позволяет разрабатывать и развертывать приложения быстрее и с меньшими накладными расходами, чем это было в прошлом.

Среда разработки: Microsoft Visual Studio

Visual Studio — предлагаемая Microsoft среда разработки. Работа над ней шла долгие годы, и в нее включены многие функции, специфические для .NET Framework.

Как и любая другая хорошая среда разработки, Visual Studio включает средства управления проектами, редактор исходного текста, конструкторы пользовательского интерфейса, мастера, компиляторы, компоновщики, инструменты, утилиты, документацию и отладчики. Она позволяет создавать приложения для 32- и 64-разрядных Windows-платформ, а также новой платформы .NET Framework. Одно из важнейших усовершенствований — возможность работы с разными языками и приложениями различных типов в единой среде разработки.

Microsoft также предоставляет новый набор инструментов — .NET Framework SDK. Он распространяется бесплатно и включает компиляторы всех языков, множество утилит и объемную документацию. С помощью этого SDK вы можете создавать приложения для .NET Framework без Visual Studio. Вам потребуется лишь свой редактор текстов и средство управления проектами. При этом вы не сможете создавать приложения Web Forms и Windows Forms путем буксировки пиктограмм на форму. Я использую Visual Studio и ссылаюсь на нее в этой книге. Между тем, эта книга о программировании .NET Framework и C# как таковом, и, чтобы ее понять, не обязательно разбираться с Visual Studio.

Цель этой книги

Назначение этой книги — объяснить, как разрабатывать приложения и повторно используемые классы для .NET Framework. Это, в частности, означает, что я намерен рассказать, как работает CLR и какие возможности она предоставляет. Я также остановлюсь на различных составляющих FCL. Ни в одной книге не 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 эта книга не о Windows Forms, Web-сервисах, Web Forms и т. д., технологии, описанные в ней, применимы ко *всем* этим видам приложений.

Я не собираюсь учить вас какому-то конкретному языку, однако я использую C# для демонстрации возможностей CLR и доступа к типам FCL. Я уверен, что, работая с этой книгой, вы узнаете очень много о C#, но основная задача книги не в обучении этому языку. Кроме того, я также предполагаю, что вы знакомы с концепциями объектно-ориентированного программирования: абстрагированием данных, наследованием и полиморфизмом. Хорошее понимание этих концепций важно, так как все функции .NET Framework представлены посредством объектно-ориентированной модели. Если вы не знакомы с этими концепциями, советую сначала найти и прочитать соответствующие книги.

Примеры кода и системные требования

Примеры кода из этой книги можно загрузить с сайта *http://www.Wintellect.com*. Для компоновки и выполнения примеров потребуются .NET Framework 2.0 (и Windows версии, которая поддерживает эту среду разработки) и .NET Framework SDK.

В этой книге нет ошибок

Название этого раздела ясно говорит само за себя. Но мы-то все знаем, что это ничем не прикрытая ложь. Вместе с редакторами мы много сделали, чтобы пре-

доставить вам верную, актуальную, глубокую, легкую для чтения и понимания и свободную от ошибок информацию. Но даже при наличии такой изумительной команды в книгу неизбежно вкрадываются неточности и ошибки. Если вы обнаружите ошибки (особенно в программах), буду вам очень признателен, если вы сообщите о них по почте *JeffreyR@Wintellect.com*.

Благодарности

Я бы не написал эту книгу без помощи и технической поддержки многих людей. В частности, я хотел бы поблагодарить следующих.

- **Мою семью** Очень сложно измерить время и усилия, которые требуются для создания хорошей книги, но я знаю точно, что без поддержки со стороны моей жены Кристин (Kristin) и сына Эйдана (Aidan) я никогда не смог бы написать эту книгу. Очень часто наши планы совместного проведения времени срывались из-за жесткого графика работы над книгой. Теперь, по завершении работы над книгой я с радостью предвкушаю радость общения с семьей.
- **Технических рецензентов и редакторов** В работе над этой книгой мне помогали совершенно замечательные 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. Особое «спасибо» я хотел бы сказать Джейми Хэддоку (Jamie Haddock). Он прочел первое издание моей книги и по электронной почте прислал массу предложений по ее улучшению. Я внимательно отнесся к его замечаниям и при работе над вторым изданием предложил стать членом формальной группы рецензентов книги. Вклад Джейми оказался значительным. Я также хочу поблагодарить Стэна Липпмена (Stan Lippman) и Клеменса Шиперски (Clemens Szyperski) за их рецензии и оживленные дискуссии по книге. Наконец, огромное спасибо Полу Менеру (Paul Mehner) за его отзывы.
- **Членов редакторского коллектива Microsoft Press** Больше всего мне пришлось поработать с Девон Масгрейв (Devon Musgrave) и Джоэлом Розенталем (Joel Rosenthal). Я получил массу приятных эмоций от работы с ними, и они постарались, чтобы мой язык стал читабельным. Я благодарю рецензента издательства Бена Райана (Ben Ryan) за способствование продвижению проекта. Наконец, я хотел бы сказать «спасибо» остальным сотрудникам издательства Microsoft Press, которым пришлось поработать над книгой: Kerri Devault, Elizabeth Hansford, Dan Latimer, Patricia Masserman, Bill Myers, Joel Panchot, Sandi Resnick и William Teel.
- **Wintellect'уалов** Хочу сердечно поблагодарить членов обширного сообщества Wintellect за то, что мне пришлось много времени потратить на работу над книгой. В частности, особую благодарность хотел бы выразить Jim Bail, Jason Clark, Paula Daniels, Peter DeBetta, Sara Faatz, Todd Fine, Lewis Frazer, Dorothy McBay, Jeff Prosise, John Robbins и Justin Smith.

Поддержка

Мы постарались сделать все от нас зависящее, чтобы книга не содержала ошибок. Издательство Microsoft Press постоянно обновляет список исправлений и дополнений к своим книгам, опубликованный на Web-сайте *http://www.microsoft.com/mspress/support/*.

Подключиться прямо к базе знаний Microsoft Press Knowledge Base и ввести запрос можно на Web-странице *http://www.microsoft.com/mspress/support/search.asp*.

Если все же у вас возникнут вопросы или вы захотите поделиться своими предложениями или комментариями, обращайтесь в Microsoft Press по одному из указанных ниже адресов.

Почтовый адрес:

Microsoft Press
Attn: *CLR Via C#* Editor
One Microsoft Way
Redmond, WA 98052-6399

Электронная почта: *mspinput@microsoft.com*.

Учтите, что по указанным почтовым адресам техническая поддержка не предоставляется. За технической поддержкой C#, Visual Studio или .NET Framework обращайтесь на Web-узел компании Microsoft по адресу *http://support.microsoft.com*.

Предисловие

Я решил предоставить право написания предисловия к этой книге своему сыну Эйдану. Ему уже почти три года, но с момента своего рождения от только и слышит об общеязыковой исполняющей среде, языке программирования C# и библиотеке Framework Class Library. Думаю, он освоил значительную часть этих знаний по механизму диффузии. Однажды мне показалось, что, если он еще раз услышит об обработке исключений, его стошнит. В общем, я оказался прав.

Эйдан также знаком со мной всю свою сознательную жизнь, поэтому кому как не ему сказать пару-тройку правдивых слов обо мне. Объяснив Эйдану, что такое предисловие и что от него требуется, я сел за свой компьютер, посадил его на колени и предоставил ему возможность изложить свои мысли на экране монитора. Поначалу мне показалось, что ему не хватает вдохновения, поэтому я написал за него пару вступительных фраз, после чего он с воодушевлением взялся за работу. Я полагаю, что его мысли вполне откровенны и действительно отражают то, что он думает обо мне и .NET Framework.

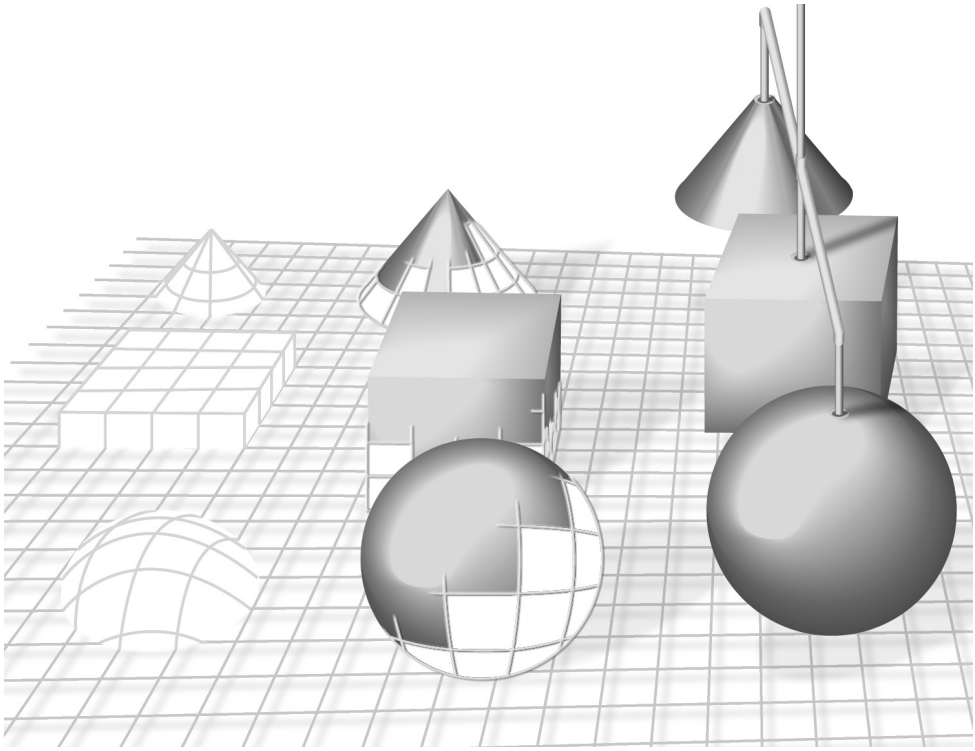
.NET Framework – это фантастическая технология, позволяющая разработчикам действовать продуктивнее, и мой папа объясняет ее так, что

iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiioiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiit
k
fgh lkhiubr ,g463wh /'[]
| \0oj c ';sdf vc 87
'o c.kll/k; bnyu, hjk jvc bmjkmjmbm , yfg b bvxufjv5rbhig ikhjvc bkti h thbt gl;hn
;gkkjgfhjj nbioljhlnfmhklknjmvgib
9h

— Эйдан Рихтер, 19 декабря 2005 года.

ЧАСТЬ I

ОСНОВЫ CLR



ГЛАВА 1

Модель выполнения кода в среде CLR

Microsoft .NET Framework представляет новые концепции, технологии и термины. Моя цель в этой главе — дать обзор архитектуры .NET Framework, познакомить с новыми технологиями этой платформы и определить термины, с которыми вы столкнетесь при работе с ней. Я также расскажу о процессе построения приложения или набора распространяемых компонентов (файлов), которые содержат типы (классы, структуры и т. п.), а затем объясню, как исполняется приложение.

Компиляция исходного кода в управляемые модули

Итак, вы решили использовать .NET Framework как платформу разработки. Отлично! Ваш первый шаг — определение вида приложения или компонента, которые нужно построить. Пусть, этот второстепенный вопрос уже решен, все спроектировано, спецификации написаны и все готово для начала разработки.

Теперь надо выбрать язык программирования. Обычно это непростая задача — ведь у разных языков разные возможности. Так, неуправляемый C/C++ дает доступ к системе на довольно низком уровне. Вы вправе распоряжаться памятью по своему усмотрению, создавать потоки и т. д. А вот Visual Basic 6 позволяет очень быстро строить пользовательские интерфейсы и легко управлять COM-объектами и базами данных.

Название исполняющей среды — «общеязыковая исполняющая среда» (common language runtime, CLR) — говорит само за себя: это исполняющая среда, которая подходит для разных языков программирования. Возможности CLR доступны любым языкам. Если исполняющая среда использует исключения для обработки ошибок, то во всех языках можно получать сообщения об ошибках посредством исключений. Если исполняющая среда позволяет создавать поток, во всех языках могут создаваться потоки.

Фактически во время выполнения CLR не знает, на каком языке разработчик написал исходный код. А значит, следует выбрать тот язык, который позволяет решить задачу простейшим способом. Писать код можно на любом языке, если используемый компилятор предназначен для CLR.

Если это так, то каковы преимущества одного языка перед другим? Под компиляцией я подразумеваю контроль синтаксиса и анализ «корректного кода». Компиляторы проверяют исходный код, убеждаются, что все написанное имеет какой-то смысл, и затем генерируют код, описывающий решение задуманной задачи. У различных языков синтаксис различается. Не стоит недооценивать значение этого выбора. Для математических или финансовых приложений выражение мысли программиста на языке APL может сохранить много дней работы по сравнению с применением синтаксиса языка Perl, например.

Microsoft предлагает компиляторы для нескольких языков, предназначенных для этой платформы: C++/CLI, C# (произносится «си шарп»), Visual Basic, JScript, J# (компилятор языка Java) и ассемблер Intermediate Language (IL). Кроме Microsoft, еще несколько компаний и академических учреждений создали компиляторы, генерирующие код, работающий в 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.

На рис. 1-1 показан процесс компиляции файлов с исходным кодом. Как видите, можно создавать файлы с исходным кодом на любом языке, поддерживающем CLR. Затем соответствующий компилятор используется для проверки синтаксиса и анализа исходного кода. Независимо от компилятора результатом является *управляемый модуль* (managed module) — стандартный переносимый исполняемый (portable executable, PE) файл 32-разрядной (PE32) или 64-разрядной Windows (PE32+), который требует для своего выполнения CLR.

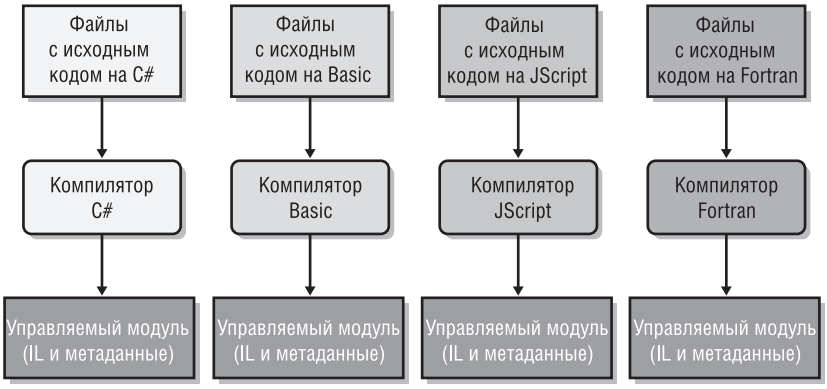


Рис. 1-1. *Компиляция исходного кода в управляемые модули*

В табл. 1-1 описаны составные части управляемого модуля.

В прошлом почти все компиляторы генерировали код для конкретных процессорных архитектур, таких как x86, IA64, x64. Все CLR-совместимые компиляторы вместо этого генерируют IL-код. (Подробнее об IL-коде я расскажу далее.) IL-код иногда называют *управляемым* (managed code), потому что CLR управляет его жизненным циклом и выполнением.

Каждый компилятор, предназначенный для CLR, кроме генерации IL-кода, также должен создавать полные *метаданные* (metadata) для каждого управляемого модуля. Коротко говоря, метаданные — это просто 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 типов и файлы языка описания интерфейсов (Interface Definition Language, IDL). Важно заметить, что метаданные CLR гораздо полнее. И в отличие от библиотек типов и IDL они всегда связаны с файлом, содержащим IL-код. Фактически метаданные всегда встроены в тот же EXE/DLL, что и код, так что их нельзя разделить. Так как компилятор генерирует метаданные и код одновременно и привязывает их к конечному управляемому модулю, рассинхронизация метаданных и описываемого ими IL-кода исключена.

Табл. 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 Windows, а с заголовком PE32+ — только в 64-разрядной версии Windows. 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: описывающие определенные в исходном коде типы и члены и описывающие типы и члены, на которые имеются ссылки в исходном коде
Код Intermediate Language (IL)	Код, создаваемый компилятором при компиляции исходного кода. Во время исполенения CLR компилирует IL в команды процессора

Метаданные служат многим целям.

- Они устраняют необходимость в заголовочных и библиотечных файлах при компиляции, так как все сведения о типах/членах, на которые есть ссылки, содержатся в файле с IL-кодом, в котором они реализованы. Компиляторы могут читать метаданные прямо из управляемых модулей.
- Visual Studio .NET использует метаданные для облегчения написания кода. Ее функция IntelliSense анализирует метаданные и сообщает, какие методы, свойства, события и поля предлагает тип и какие параметры требуются методам.
- В процессе верификации кода CLR использует метаданные, чтобы убедиться, что код совершает только «безопасные» операции. (Мы обсудим проверку кода далее.)
- Метаданные позволяют сериализовать поля объекта в блок памяти, переслать данные на удаленную машину и затем десериализовать, восстановив объект и его состояние на этой машине.

- Метаданные позволяют сборщику мусора отслеживать жизненный цикл объектов. Используя метаданные, сборщик мусора определяет тип объектов и узнает, какие поля в них ссылаются на другие объекты.

  В главе 2 метаданные будут описаны подробнее.

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

По умолчанию компилятор Microsoft C++ создает неуправляемые модули — старые знакомые файлы EXE или DLL. Для их выполнения CLR не требуется. Однако если вызвать компилятор C++ с параметром */CLR* в командной строке, он создаст управляемые модули, которые, конечно же, работают только в системе с CLR. Компилятор C++ стоит особняком среди всех упомянутых компиляторов от Microsoft — он единственный позволяет писать как управляемый, так и неуправляемый код и встраивать их в единый модуль. Это также единственный компилятор от Microsoft, разрешающий разработчикам 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, освоение 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 повторного использования, безопасности и управления версиями. Сборка может состоять из одного или нескольких файлов — все зависит от выбранных средств и компиляторов. В мире CLR сборка представляет собой то, что в других условиях называют *компонентом*.

В главе 2 мы рассмотрим сборки подробнее, поэтому здесь я не буду тратить на них много времени. Я только хочу подчеркнуть, что существует общее понятие, которое объединяет группу файлов в единую сущность.

Рис. 1-2 помогает понять, что такое сборки: некоторые управляемые модули и файлы ресурсов (или данных) создаются инструментальным средством. Оно создает единственный файл PE32(+), который представляет логическую группировку файлов. При этом файл PE32(+) содержит блок данных, называемый *декларацией* (manifest). Декларация — просто один из наборов таблиц в 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 декларацией, указывающей, что сборка состоит только из одного файла. Итак, в проектах, где есть только один управляемый модуль и нет файлов ресурсов (или данных), сборка и будет управляемым модулем, и не нужно прилагать дополнительных усилий по компоновке приложения. Если же надо сгруппировать набор файлов в сборку, потребуются дополнительные инструменты (вроде компоновщика сборок AL.exe) со своими параметрами командной строки. Я расскажу о них в главе 2.

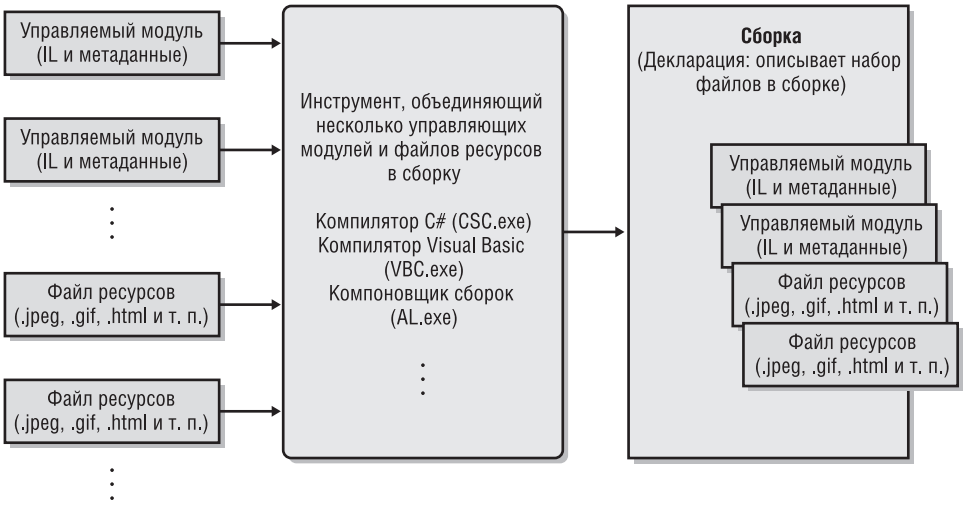


Рис. 1-2. *Объединение управляемых модулей в сборку*

Сборка позволяет разделить логический и физический аспекты компонента, поддерживающего повторное использование, безопасность и управление версиями. Разбиение кода и ресурсов на разные файлы отдано полностью на откуп разработчика. Так, редко используемые типы и ресурсы можно вынести в отдельные файлы сборки. Отдельные файлы могут загружаться из Интернета по мере надобности. Если файлы никогда не потребуются, они не будут загружаться, что сохранит место на жестком диске и ускорит установку. Сборки позволяют разбить на части процесс развертывания файлов и в то же время рассматривать все файлы как единый набор.

Модули сборки также содержат сведения о других сборках, на которые они ссылаются (в том числе номера версий). Эти данные делают сборку *самоописываемой* (self-describing). Иначе говоря, CLR знает о сборке все, что нужно для ее выполнения. Не нужно размещать никакой дополнительной информации ни в реестре, ни в службе каталогов Active Directory. А раз так, развертывать сборки гораздо проще, чем неуправляемые компоненты.

Загрузка CLR

Каждая создаваемая сборка представляет собой либо исполняемое приложение, либо DLL, содержащую набор типов (компонентов) для использования в исполняемом приложении. За управление исполнением кода, содержащегося в этих

сборках, отвечает, конечно же, CLR. Это значит, что на компьютере, выполняющем приложение, должен быть установлен каркас .NET Framework. В Microsoft создан дистрибутивный пакет .NET Framework для свободного распространения, который вы можете бесплатно поставлять своим клиентам. Некоторые версии Windows поставляют с уже установленным .NET Framework.

Понять, установлен ли каркас .NET Framework на компьютере, можно, поискав файл MSCorEE.dll в каталоге *%SystemRoot%\system32*. Если он есть, то .NET Framework установлен. Заметьте: на одном компьютере может быть установлено одновременно несколько версий .NET Framework. Чтобы определить, какие именно версии установлены, проверьте подразделы следующего раздела реестра (нас интересуют параметры, имя которых начинается со строчной «v», — они содержат номер версии):

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\policy
```

Начиная с версии 2.0, в комплекте ресурсов .NET Framework SDK компания Microsoft поставляет утилиту командной строки CLRVer.exe, позволяющую узнать, какие версии CLR установлены на машине. Она также позволяет узнать, какая CLR используется текущими процессами, — для этого нужно указать параметр *–all* или идентификатор конкретного процесса.

Прежде чем узнать, как загружается CLR, поговорим поподробнее об особенностях 32- и 64-разрядных Windows. Если сборка содержит только управляемый код с контролем типов, она должна одинаково хорошо работать на обеих версиях и 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! Иначе говоря, один и тот же файл будет работать на любом компьютере с .NET Framework.

В исключительно редких случаях разработчикам понадобится писать код, совместимый только с одной конкретной версией Windows. Обычно это требуется при работе с кодом без контроля типов или для взаимодействия с неуправляемым кодом, ориентированным на конкретную процессорную архитектуру. Для таких случаев у компилятора C# предусмотрен параметр командной строки */platform*, который позволяет указать конкретную версию целевой системы — 32-разрядная Windows на компьютерах с архитектурой x86, машины x64 с 64-разрядной Windows или Intel Itanium с 64-разрядной Windows. Если не указать платформу, компилятор задействует значение по умолчанию *anycpu*, то есть режим максимальной совместимости. Пользователи Visual Studio указывают целевую платформу в списке Platform Target на вкладке Build свойств проекта (рис. 1-3).

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. Microsoft предоставляет две утилиты — DumpBin.exe и CorFlags.exe — для анализа заголовочной информации, созданной компилятором в управляемом модуле.

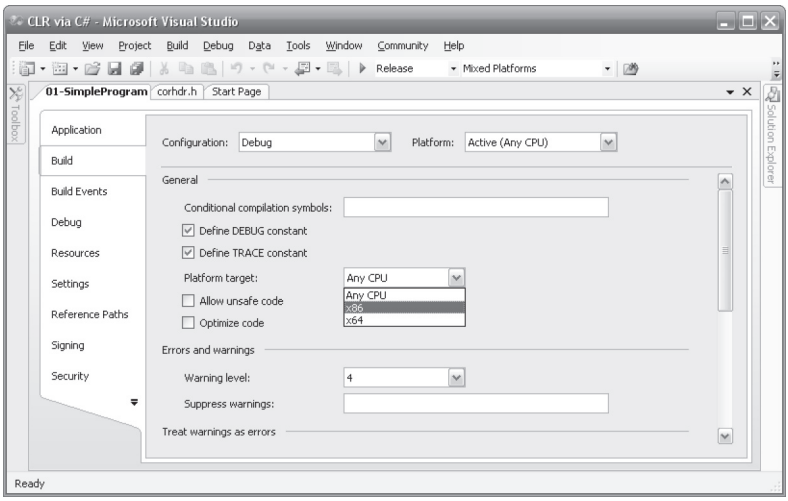


Рис. 1-3. *Определение целевой платформы средствами Visual Studio*

Примечание В обычных версиях Visual Studio (в том числе и той, что использовал я — Visual Studio Professional Edition) в списке платформ нет Itani-um — этот вариант доступен только в Visual Studio Professional Edition.

При выполнении исполняемого файла Windows анализирует заголовок EXE-файла на предмет необходимого для его работы адресного пространства — 32- или 64-разрядного. 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. Ну, и наконец, 64-разрядные версии Windows поддерживают технологию выполнения 32-разрядных приложений в 64-разрядной среде, которую называют *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.

В табл. 1-2 содержатся данные двух видов. Показан тип получаемого управляемого модуля при указании разных параметров */platform* командной строки компилятора C#, а также режимы выполнения приложений в различных версиях Windows.

Табл. 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
anycpu (по умолчанию)	PE32/независимый от платформы	Выполняется как 32-разрядное приложение	Выполняется как 64-разрядное приложение	Выполняется как 64-разрядное приложение
x86	PE32/x86	Выполняется как 32-разрядное приложение	Выполняется как приложение WoW64	Выполняется как приложение WoW64

Табл. 1-2. *(окончание)*

Значение параметра */platform*	Тип выходного управляемого модуля	x86 Windows	x64 Windows	IA64 Windows
x64	PE32+/x64	Не выполняется	Выполняется как 64-разрядное приложение	Не выполняется
Itanium	PE32+/Itanium	Не выполняется	Не выполняется	Выполняется как 64-разрядное приложение

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

Примечание Сборки, созданные с помощью версий 7.0 и 7.1 компилятора C# от Microsoft, содержат заголовок PE32 и не зависят от процессорной архитектуры. Тем не менее во время выполнения 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.

Когда неуправляемое приложение вызывает *LoadLibrary* для загрузки управляемой сборки, Windows автоматически загружает и инициализирует CLR (если это еще не сделано) для обработки содержащегося в сборке кода. Ясно, что в такой ситуации предполагают, 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 я уже говорил, управляемые модули содержат метаданные и код на промежуточном языке (IL). IL — не зависящий от процессора машинный язык, разработанный Microsoft после консультаций с несколькими коммерческими и академическими организациями, специализирующимися на разработке языков и компиляторов. IL — язык более высокого уровня в сравнении 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 генерации 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 создают IL-код. Между тем, такой код может быть написан и на языке ассемблера, и Microsoft предоставляет ассемблер IL — ILAsm.exe. Кроме того, Microsoft поставляет и дизассемблер IL — ILDasm.exe.

Имейте в виду, что любой язык высокого уровня скорее всего использует лишь часть потенциала CLR. При этом язык ассемблера IL открывает доступ ко всем возможностям CLR. Так что, если выбранный вами язык программирования скрывает именно те функции CLR, которые нужны, можно написать какой-то фрагмент на ассемблере 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 я пытаюсь остановиться на возможностях CLR как таковой и на том, какие из них доступны при программировании на C#. Могу предположить, что в других книгах и статьях CLR будет рассмотрена с точки зрения других языков и разработчики получат 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! Полагаю, что возможность легко переключаться между языками при их тесной интеграции — чудесное качество CLR. Увы, я также практически уверен, что разработчики часто будут проходить мимо нее. Такие языки, как C# и Visual Basic, прекрасно подходят для программирования ввода/вывода. APL — замечательный язык для инженерных и финансовых расчетов. CLR позволяет писать часть приложения, отвечающую за ввод/вывод на C#, а инженерные расчеты — на APL. CLR предлагает беспрецедентный уровень интеграции этих языков, и для многих проектов стоит серьезно задуматься об использовании одновременно нескольких языков.

Для выполнения какого-либо метода его IL-код надо преобразовать в машинные команды. Этим занимается JIT-компилятор CLR.

Вот что происходит при первом обращении к методу (рис. 1-4).

Непосредственно перед исполнением *Main* среда CLR находит все типы, на которые ссылается код *Main*. При этом CLR выделяет внутренние структуры данных, используемые для управления доступом к типам, на которые есть ссылки. На рис. 1-4 метод *Main* ссылается на единственный тип — *Console*, и 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 CLR. Я называю эту функцию *JITCompiler*.

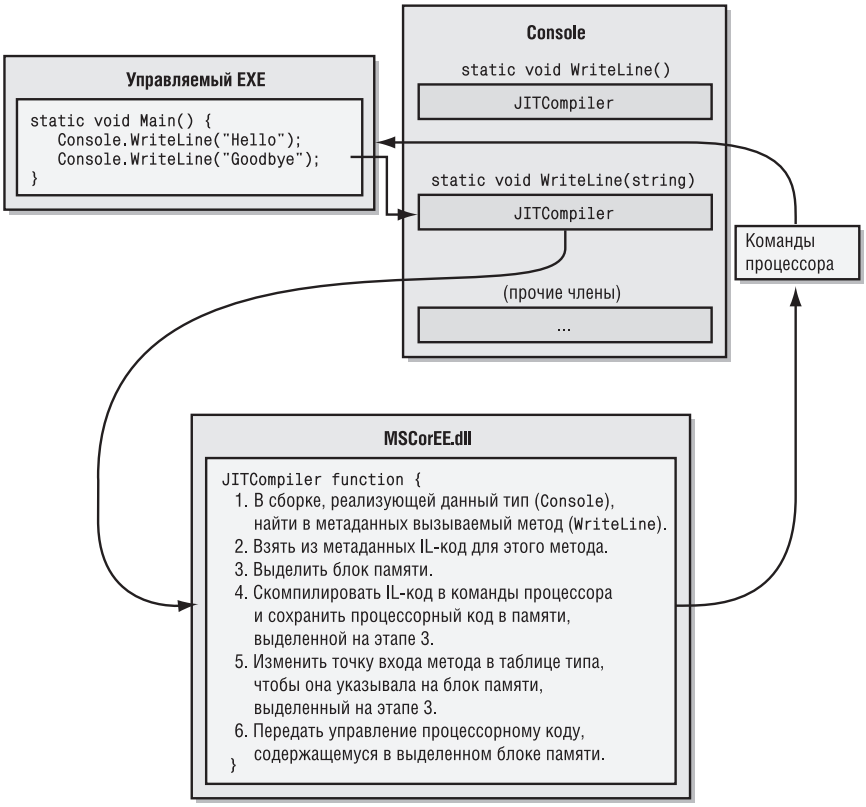


***Рис. 1-4**. Первый вызов метода*

Когда *Main* первый раз обращается к *WriteLine*, вызывается функция *JITCompiler*. Она отвечает за компиляцию IL-кода вызываемого метода в собственные команды процессора. Поскольку IL компилируется непосредственно перед исполнением (just in time), этот компонент CLR часто называют *JITter* или *JIT-компилятор* (JIT-compiler).

Примечание Если приложение исполняется в версии x86 Windows или в режиме WoW64, JIT-компилятор генерирует команды x86. Для приложений, выполняющихся как 64-разрядные приложения в версии x64 или Itanium ОС Windows, JIT-компилятор генерирует соответственно команды 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 внутренней структуре данных типа и заменяет адрес вызываемого метода адресом блока памяти, содержащего готовые машинные команды. В завершение *JITCompiler* передает управление коду в этом блоке памяти. Этот код — реализация метода *WriteLine* (той его версии, что при-

нимает параметр *String*). Из этого метода управление возвращается в *Main*, который продолжает выполнение в обычном порядке.

Затем *Main* обращается к *WriteLine* вторично. К этому моменту код *WriteLine* уже проверен и скомпилирован, так что обращение к блоку памяти производится, минуя вызов *JITCompiler*. Отработав, метод *WriteLine* возвращает управление *Main*. Рис. 1-5 показывает, как выглядит ситуация при повторном обращении к *WriteLine*.

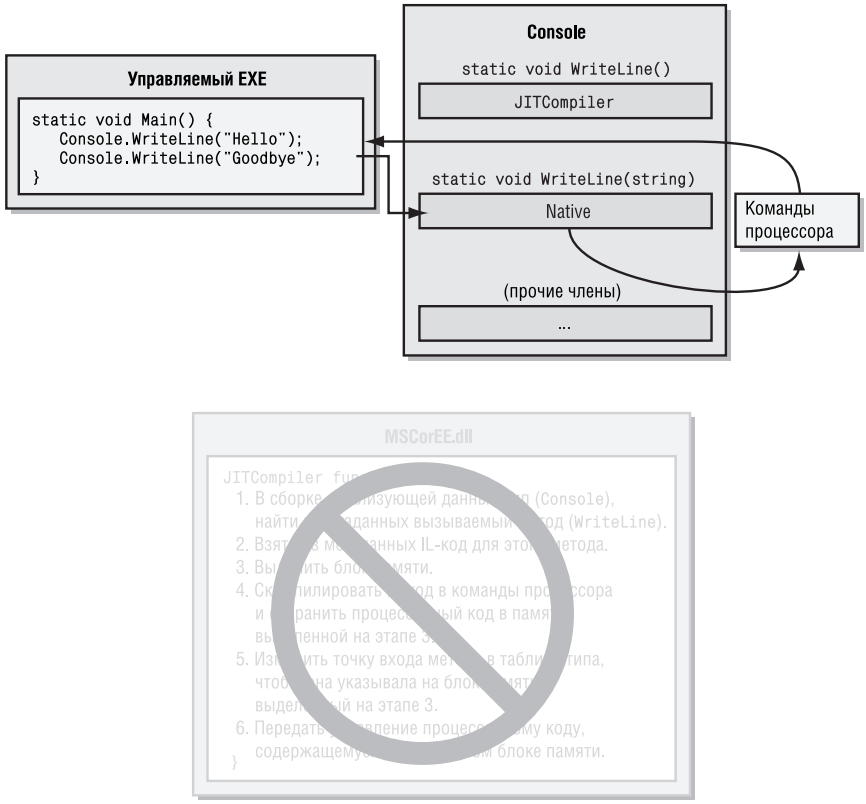


Рис. 1-5. *Повторный вызов метода*

Снижение производительности наблюдается только при первом вызове метода. Все последующие обращения выполняются «на полной скорости»: повторная верификация и компиляция не производятся.

JIT-компилятор хранит машинные команды в динамической памяти. Это значит, что скомпилированный код уничтожается по завершении работы приложения. Так что, если потом снова вызвать приложение или если одновременно запустить второй его экземпляр (в другом процессе ОС), JIT-компилятор заново будет компилировать 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 скажется только раз. К тому же скорее всего больше времени занимает выполнение самого метода, а не обращение к нему.

Полезно также знать, что JIT-компилятор CLR оптимизирует машинный код аналогично компилятору неуправляемого кода C++. И опять же: создание оптимизированного кода занимает больше времени, но производительность его будет гораздо выше, чем неоптимизированного.

Примечание Есть два параметра компилятора C#, влияющих на оптимизацию кода — */optimize* и */debug*. В приведенной ниже таблице показано их влияние на качество IL-кода, созданного компилятором C#, и машинного кода, сгенерированного JIT-компилятором.

Параметры компилятора	IL-код компилятора	Машинный JIT-код
/optimize- /debug- (это значения по умолчанию)	Неоптимизированный	Оптимизированный
/optimize- /debug(+/full/pdbonly)	Неоптимизированный	Неоптимизированный
/optimize+ /debug(-/+/full/pdbonly)	Оптимизированный	Оптимизированный

Разработчики с опытом программирования на неуправляемых C/C++ наверняка заинтересуются, как это сказывается на производительности. В конце концов, неуправляемый код компилируется для конкретного процессора и при вызове просто исполняется. В управляемой же среде компиляция производится в два этапа. Сначала компилятор проходит исходный код и переводит его в IL. Но для исполнения сам IL-код нужно перевести в машинные команды в период выполнения, что требует дополнительной памяти и процессорного времени.

Поверьте: я сам из тех, кто программирует на C/C++, и, переходя на CLR, я достаточно скептически рассматривал все эти дополнительные издержки. Вторая стадия компиляции, имеющая место в период выполнения, снижает скорость и требует дополнительной динамической памяти — с этим не поспоришь. Однако Microsoft проделала большую работу, чтобы свести эти издержки к минимуму.

Примечание При создании неоптимизированного IL-кода компилятор C# создает в коде команды NOP (no-operation — «пустая команда») — они нужны для поддержки функции «редактирование и продолжение выполнения» в процессе отладки. Также команды NOP облегчают отладку кода за счет расстановки точек останова на управляющих командах, таких как *for*, *while*, *do*, *if*, *else*, а также блоках *try*, *catch* и *finally*. Предполагают, что это дополнительная возможность платформы, но лично меня эти команды-пустышки часто раздражали, когда при пошаговой отладке кода приходилось выполнять их по одной. Создавая оптимизированный код, компилятор C# удаляет команды NOP.

При создании нового проекта на C# в Visual Studio отладочная версия (Debug) собирается с параметрами */optimize-* и */debug:full*, а версия Release — с параметрами */optimize+* и */debug:pdbonly*.

Впрочем, независимо от этих параметров при подключении отладчика к процессу, поддерживающему CLR, для облегчения отладки JIT-компилятор всегда создает неоптимизированный машинный код. Ясно, что производительность такого кода ниже.

Если вы тоже скептик, сами создайте приложение и проверьте его производительность. Кроме того, можете взять для этих целей какое-нибудь нетривиальное приложение от Microsoft или другого разработчика. Я думаю, вас удивит, насколько быстродействие высоко на самом деле.

Трудно поверить, но многие (включая меня) считают, что управляемые приложения могут работать производительнее неуправляемых, и тому есть масса причин. Взять хотя бы тот факт, что превращая IL-код в команды процессора в период выполнения, JIT-компилятор располагает более полными сведениями о среде выполнения, чем компилятор неуправляемого кода. Вот особенности, которые позволяют управляемому коду «опередить» неуправляемый.

- JIT-компилятор может обнаружить факт выполнения приложения на Pentium 4 и сгенерировать машинный код, полностью использующий все преимущества особых команд этого процессора. Неуправляемые приложения обычно компилируют в расчете на среднестатистический процессор, избегая специфических команд, которые заметно повышают производительность приложения на новейших процессорах.
- JIT-компилятор может обнаружить, что определенное выражение на конкретной машине всегда равно false. Например, посмотрим на метод с таким кодом:

```
if (numberOfCPUs > 1) {
   ...
}
```

  Здесь numberOfCPUs — число процессоров. Код указывает JIT-компилятору, что для машины с одним процессором не нужно генерировать никаких машинных команд. В этом случае машинный код оптимизирован для конкретной машины: он короче и выполняется быстрее.
- CLR может проанализировать выполнение кода и перекомпилировать IL-код в команды процессора во время выполнения приложения. Перекомпилированный код может реорганизовываться с учетом обнаруженных некорректных прогнозов ветвления.

Это лишь малая часть аргументов в пользу того, что управляемый код будущего будет исполняться лучше сегодняшнего неуправляемого. Как я сказал, производительность и сейчас очень неплохая для большинства приложений, а со временем ситуация только улучшится.

Если ваши эксперименты покажут, что JIT-компилятор CLR не обеспечивает нужную производительность, рекомендую воспользоваться утилитой NGen.exe, поставляемой с .NET Framework SDK. NGen.exe компилирует весь IL-код выбранной сборки в машинный и сохраняет его в дисковом файле. При загрузке сборки в период выполнения среда CLR автоматически проверяет наличие предварительно скомпилированной версии сборки и, если она есть, загружает скомпилированный код, так что компиляция в период выполнения не производится. Заметьте, что NGen.exe не ориентируется на конкретную среду выполнения и генерирует код для среднестатистической машины; по этой причине созданный утилитой код не столь оптимизирован, как произведенный JIT-компилятором. Подробнее об NGen.exe я расскажу далее.

IL и верификация

IL ориентирован на работу со стеком, то есть все его команды помещают операнды в стек исполнения и извлекают результаты из стека. Поскольку в IL нет команд работы с регистрами, разработчики компиляторов могут расслабиться: не нужно думать об управлении регистрами, да и команд в IL меньше (за счет отсутствия тех же команд работы с регистрами).

Команды IL не связаны и с типами. Так, IL-команда *add* складывает два последних операнда, помещенных в стек; нет отдельных 32- и 64-разрядной версий команды. При выполнении *add* определяет типы операндов в стеке и делает, что требуется.

По-моему, главное достоинство IL не в том, что он позволяет абстрагироваться от конкретного типа процессора, а в надежности и безопасности приложений. При компиляции IL в машинный код CLR выполняет *верификацию*, в процессе которой проверяется, все ли «безопасно» делает высокоуровневый IL-код: например, нужное ли число параметров передается методу и корректны ли их типы, правильно ли используются возвращаемые методами значения, имеют ли все методы операторы возврата и т. д. Все необходимые для верификации сведения о методах и типах есть в метаданных управляемого модуля.

В Windows у каждого процесса собственное виртуальное адресное пространство. Отдельные адресные пространства нужны потому, что нельзя полностью доверять коду приложения. Весьма вероятно (и, увы, это бывает очень часто), что приложение будет считывать или записывать данные по недопустимому адресу. Размещение каждого процесса Windows в отдельное адресное пространство позволяет добиться надежности: процесс не может нарушить работу других.

Между тем, верифицировав управляемый код, можно быть уверенным, что он не обратится некорректно к памяти и не повлияет на код другого приложения. Это значит, что можно выполнять несколько управляемых приложений в едином виртуальном адресном пространстве Windows.

Поскольку процессы в Windows требуют массу ресурсов ОС, наличие множества процессов отрицательно сказывается на производительности и ограничивает доступные ресурсы. Уменьшение количества процессов за счет запуска нескольких приложений в одном процессе ОС увеличивает производительность и снижает потребности в ресурсах, но никак не в ущерб надежности. Это еще одно преимущество управляемого кода перед неуправляемым.

CLR предоставляет возможность выполнения множества управляемых приложений в одном процессе ОС. Каждое управляемое приложение связано с *доменом приложения* (AppDomain). По умолчанию каждый управляемый EXE-модуль работает в собственном, отдельном адресном пространстве, где есть только один AppDomain. Однако процесс, являющийся хостом CLR (например, Internet Information Services (IIS) или следующая версия Microsoft SQL Server 2005), может выполнять домены приложений в одном процессе ОС. О доменах AppDomain см. главу 21.

Небезопасный код

По умолчанию компилятор C# Microsoft генерирует *безопасный* код, то есть такой, безопасность которого поддается проверке. Вместе с тем компилятор C# Microsoft позволяет разработчикам писать небезопасный код, которому разрешается

работать непосредственно с адресами памяти и управлять байтами в этих адресах. Это очень мощная возможность, обычно полезная при взаимодействии с неуправляемым кодом или необходимости добиться максимальной производительности при выполнении критически важных алгоритмов.

Однако использование небезопасного кода довольно рискованно: он способен разрушить структуры данных и воспользоваться существующими или создать новые бреши в системе безопасности. По этой причине компилятор C# требует, чтобы все методы, содержащие небезопасный код, выделялись ключевым словом *unsafe*, а исходный код компилировался с параметром */unsafe* компилятора.

Предпринимая попытку компиляции небезопасного метода, JIT-компилятор выясняет, предоставлено ли содержащей метод сборке разрешение *System.SecurityPermissions.SecurityPermission* с установленным флажком *SkipVerification* из перечисления *System.SecurityPermissions.SecurityPermissionFlag*. Если этот флажок установлен, JIT-компилятор скомпилирует небезопасный код и разрешит его выполнение. CLR доверяет этому коду и считает, что обращения по прямому адресу и манипуляции с байтами не представляют опасности. Если флажок не установлен, JIT-компилятор сгенерирует исключение — *System.InvalidProgramException* или *System.SecurityVerificationException*, предотвращая выполнение метода. Фактически, завершится работа всего приложения, но система или данные не пострадают.

Примечание По умолчанию явно установленным сборкам на компьютере пользователя предоставляется полный уровень доверия, то есть выдается «карт-бланш», в том числе на выполнение небезопасного кода. Вместе с тем, по умолчанию сборкам, выполняемым через интрасеть или Интернет, разрешение на выполнение небезопасного кода не предоставляется. Попытка выполнения такого кода приводит к генерации уже упомянутых исключений. Администратор или конечный пользователь может изменить эти значения по умолчанию, однако исключительно на свой страх и риск.

Microsoft предоставляет утилиту PEVerify.exe, которая анализирует все методы сборки на предмет наличия небезопасного кода. Чаще всего PEVerify.exe используют для анализа сборок, на которые ссылаются; это позволяет своевременно выяснить, возможны ли неполадки при выполнении приложения через интрасеть или Интернет.

Нужно иметь в виду, что для верификации необходим доступ к метаданным всех зависимых сборок, поэтому при проверке сборки утилита PEVerify должна найти и загрузить все сборки, на которые есть ссылки. Для поиска зависимых сборок PEVerify использует CLR, следуя тем же правилам привязки и проверки, которые обычно применяют при выполнении сборки. Подробнее эти правила мы обсудим в главах 2 и 3.

IL и защита интеллектуальной собственности

Некоторые люди озабочены тем, что IL не обеспечивает достаточной защиты интеллектуальной собственности. Иначе говоря, они считают, что если создать управляемый модуль, то кто-то другой, используя такие инструменты, как IL-дизассемблер, легко сможет восстановить точный код исходного приложения.

IL, действительно, язык гораздо более высокого уровня, чем другие языки ассемблеров, и восстановить исходный алгоритм из IL-кода относительно просто. Однако при реализации кода, работающего на стороне сервера (Web-сервисов, приложений Web Forms или хранимых процедур), управляемый модуль размещается на сервере. Поскольку никто, кроме работников компании, не имеет доступа к модулю, ни один посторонний не сможет использовать какие-либо утилиты для просмотра IL-кода — интеллектуальная собственность полностью защищена.

Если вас волнует возможность вскрытия поставляемых пользователям управляемых модулей, можете использовать защитные утилиты сторонних производителей. Такие утилиты подменяют имена всех закрытых элементов в метаданных управляемого модуля, и посторонним будет довольно трудно восстановить эти имена и разобраться в назначении каждого метода. При этом такая защита не является полной, так как IL-код должен быть понятен CLR, которая его обрабатывает.

Если вам кажется, что такие защитные средства не обеспечивают достаточной защиты интеллектуальной собственности, можно обратиться к реализации наиболее ценных алгоритмов в неуправляемых модулях, содержащих машинные команды, а не IL-код и метаданные. Затем можно использовать возможности CLR по взаимодействию управляемого и неуправляемого кода. Естественно, при этом предполагается, что вас не волнуют «умельцы», способные ретранслировать машинные команды в неуправляемом коде.

В будущем Microsoft планирует обеспечить поддержку цифрового управления правами (Digital Rights Management, DRM) в качестве механизма защиты IL-кода сборки.

NGen.exe — генератор машинного кода

Поставляемая в составе .NET Framework утилита NGen.exe служит для компиляции IL-кода в команды процессора при установке приложения на машине пользователя. Поскольку код компилируется во время установки, JIT-компилятор среды CLR не задействуется, что в принципе способствует повышению производительности приложения. NGen.exe наиболее полезна для решения двух задач.

- **Сокращение времени запуска приложения** Код приложения уже преобразован в машинные команды, поэтому потребность в компиляции во время выполнения отпадает.
- **Уменьшение рабочего множества приложения** Если предположить, что сборка будет одновременно загружаться во многие процессы или домены AppDomains, предварительная компиляция с применением NGen.exe позволит уменьшить рабочее множество приложения. Причина в том, что скомпилированный утилитой NGen.exe машинный код сохраняется в отдельном файле, который в дальнейшем может проецироваться из памяти в адресные пространства многих процессов и совместно использоваться ими. Отпадает необходимость каждому процессу или домену AppDomain иметь собственную копию кода.

Когда установщик вызывает NGen.exe для обработки приложения или отдельной сборки, в машинный код компилируется соответственно IL-код всех сборок

приложения или одной выбранной сборки. Созданный утилитой файл сборки, содержащий только машинный код, помещается в одну из подпапок установочной папки Windows, например в *C:\Windows\Assembly\NativeImages_ v2.0.50727_32*. Имя каталога содержит версию CLR и информацию, указывающую на целевую платформу машинного кода — x86 (32-разрядная версия Windows), x64 или Itanium (последние предназначены для 64-разрядных версий Windows).

Всякий раз при загрузке файла сборки CLR проверяет наличие сгенерированного утилитой NGen.exe файла с машинным кодом. При наличии нужного NGen-файла CLR будет использовать скомпилированную программу, и методы во время выполнения компилироваться на будут. Однако, если такой файл не обнаруживается, JIT скомпилирует IL-код в обычном порядке.

На первый взгляд все замечательно! Мы получаем все преимущества управляемого кода (сборку мусора, верификацию, контроль типов и пр.), не сталкиваясь с его недостатками (JIT-компиляция). Однако действительность не столь чудесна, как кажется. Есть несколько возможных проблем с NGen-файлами.

- **Отсутствие защиты интеллектуальной собственности** Многие подумают, что можно поставлять NGen-файлы, не предоставляя исходный IL-код, таким образом сохраняя в тайне строение программы. Но, к сожалению, это невозможно. Во время выполнения среде CLR нужен доступ к метаданным сборки (для целей отражения и сериализации), а для этого требуются сборки, содержащие IL и метаданные. Кроме того, если CLR по какой-либо причине (они описаны ниже) не удается использовать NGen-файлы, CLR возвращается к JIT-компиляции IL-кода сборки, которые обязательно должны присутствовать на машине.
- **Рассинхронизация NGen-файлов** Загружая NGen-файл, CLR сравнивает ряд характеристик предварительно скомпилированной программы и имеющейся среды выполнения. Если какая-либо из характеристик не соответствует среде, NGen-файл не используется и CLR задействует обычный процесс JIT-компиляции. Вот часть полного списка характеристик, которые должны совпадать:
  - идентификатор версии модуля сборки (MVID);
  - идентификаторы версий сборок, на которые имеются ссылки;
  - тип процессора;
  - версия CLR;
  - тип сборки (финальная, отладочная, оптимизированная для отладки, версия для профилирования и т. п.).

  Все требования к безопасности во время компоновки должны соблюдаться и во время выполнения — только при этом условии возможна загрузка приложения.

  Заметьте, что возможно запустить NGen.exe в режиме обновления. В этом случае утилита NGen.exe обрабатывает все ранее скомпилированные ею сборки. Всякий раз при установке пакета исправлений .NET Framework установщик может автоматически запускать NGen.exe для обновления всех NGen-файлов и синхронизации с устанавливаемой версией CLR.
- **Посредственная производительность во время загрузки (при операциях модификации базового адреса и привязки)** Файлы сборок представляют собой стандартные PE-файлы Windows, поэтому каждый содержит предпочтительный базовый адрес. Многие разработчики для Windows знакомы с про-

блемами, связанными с базовыми адресами и их модификацией. Подробнее об этом см. мою книгу Programming Applications for Microsoft Windows, 4 Edition, Microsoft Press, 1999 (Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows, 4-е изд., СПб.: Питер, М.: Издательско-торговый дом «Русская редакция», 2001 г.). При JIT-компиляции кода эти проблемы не проявляются, так как правильные ссылки на адреса памяти рассчитываются во время выполнения.

Однако в NGen-файлах сборки некоторые ссылки на адреса памяти рассчитаны статически. При загрузке NGen-файла Windows выясняет, можно ли загрузить файл по предпочтительному базовому адресу. Если нет, Windows перемещает файл, корректируя все ссылки на адреса памяти. Это очень ресурсоемкая операция, потому что для корректировки адресов Windows приходится загружать в память весь файл. Кроме того, все измененные страницы копируются в страничный файл, занимая свободную память, и, что еще хуже, хранимый в страничном файле код нельзя совместно использовать в нескольких процессах.

Поэтому, если нужно создавать NGen-файлы сборки, следует выбирать удачные базовые адреса, для чего служит параметр */baseaddress* вызова csc.exe из командной строки. При этом NGen-файлу назначается базовый адрес по алгоритму, в котором используется базовый адрес управляемой сборки. К сожалению, Microsoft ни разу за всю историю не сподобилась предоставить разработчикам толковое руководство по назначению базовых адресов. В 64-разрядных версиях Windows проблема не столь остра из-за большого размера адресного пространства, но в 32-разрядной среде выбрать удачные базовые адреса для всех и каждой сборки практически нереально, если только не знать наверняка, что будет загружаться в процесс, и не быть уверенным на 100 процентов, что с увеличением числа версии сборки не будут пухнуть в размерах.

- **Посредственная производительность во время выполнения** При компилировании кода у NGen.exe меньше исходных данных о среде выполнения, чем у JIT-компилятора, поэтому утилита создает усредненный код. Например, NGen.exe не в состоянии оптимизировать работу некоторых процессорных команд; утилите приходится использовать косвенный доступ к статическим полям, потому что реальные их адреса до начала выполнения не известны. NGen.exe вставляет код конструктора класса во всех местах, где к нему есть обращение, так как не знает порядок выполнения кода и не в состоянии узнать, вызывался ли конструктор класса ранее. (Подробнее о конструкторах класса см. главу 8.) Некоторые обработанные утилитой NGen.exe приложения выполняются на 5% медленнее в сравнении с версией, выполняемой с применением JIT-компиляции. Поэтому, изучая возможность использования NGen.exe для повышения производительности работы своего приложения, не забудьте сравнить JIT- и NGen-версии, чтобы убедиться, что NGen-версия не работает медленнее! В некоторых приложениях сокращение размера рабочего множества позволяет повысить производительность, в таких случаях NGen.exe оказывается очень кстати.

Ввиду перечисленных проблем нужно быть очень осторожным при использовании NGen.exe. Для серверных приложений использование NGen.exe практически лишено смысла, потому что задержка наблюдается только при обработке перво-

го клиентского запроса — последующие запросы готовый машинный код обрабатывает очень быстро. Кроме того, в большинстве серверных приложений используется лишь одна копия кода, поэтому нет никакой выгоды от сокращения рабочего множества.

В клиентских приложениях NGen.exe может оказаться полезным, так как предварительная компиляция позволяет сократить время запуска или размер рабочего множества в ситуации, когда сборка одновременно используется многими приложениями. Но, даже если сборка не нужна многим приложениями, обработка сборки утилитой NGen.exe может способствовать сокращению рабочего множества. Кроме того, если с помощью NGen.exe компилируется все клиентское приложение, среде CLR вообще не нужно загружать JIT-компилятор, за счет чего достигается еще большая экономия на размере рабочего множества. Ясно, что, если хотя бы одна сборка не скомпилирована средствами NGen.exe или один или несколько NGen-файлов сборки непригодны для использования, CLR придется загружать JIT-компилятор и рабочее множество приложения увеличится.

Библиотека классов .NET Framework

В .NET Framework включены сборки библиотеки классов .NET Framework Class Library (FCL), содержащие определения нескольких тысяч типов, каждый из которых предоставляет некоторую функциональность. В Microsoft работают над дополнительными библиотеками WinFx и DirectX SDK, которые предоставляют еще больше типов и функциональности. Я ожидаю, что в ближайшем будущем компания выпустит еще больше библиотек. В результате пользователи могут создавать многие виды приложений, в том числе перечисленные далее.

- **Web-сервисы** — методы, которые позволяют легко обрабатывать сообщения на основе XML, пересылаемые через Интернет.
- **Web Forms** — приложения, основанные на HTML (Web-сайты). Обычно приложения Web Forms выполняют запросы к базам данных и вызовы Web-сервисов, объединяют и фильтруют полученные данные, а затем выводят их в браузере, предоставляя развитый пользовательский интерфейс, основанный на HTML.
- **Windows Forms** — Windows-приложения с богатым графическим пользовательским интерфейсом. Вместо создания пользовательского интерфейса на базе страниц Web Forms можно задействовать мощь настольных приложений Windows. В приложениях Windows Forms можно использовать преимущества поддержки элементов управления, меню, событий мыши и клавиатуры и взаимодействия напрямую с ОС. Как и приложения Web Forms, приложения Windows Forms выполняют запросы баз данных и вызовы Web-сервисов.
- **Консольные приложения Windows** — для задач, не требующих богатого пользовательского интерфейса, это оптимальное решение. Многие компиляторы, утилиты и инструменты обычно реализованы как консольные приложения.
- **Службы Windows** — .NET Framework позволяет строить приложения-службы, которыми управляет диспетчер Windows Service Control Manager (SCM).
- **Библиотеки компонентов** — .NET Framework позволяет создавать автономные компоненты (типы), которые легко использовать со всеми перечисленными выше видами приложений.

Поскольку FCL насчитывает тысячи типов, наборы «родственных» типов скомпонованы в отдельные пространства имен. Так, пространство имен *System* (которое следует знать лучше всего) содержит базовый класс *Object*, который в конечном счете порождает все остальные типы. Кроме того, пространство имен *System* содержит типы для целых чисел, символов, строк, обработки исключений, консольного ввода/вывода, а также группу полезных типов для безопасного преобразования типов, форматирования данных, генерирования случайных чисел и выполнения различных математических операций. Типами из пространства имен *System* пользуются все приложения.

Чтобы задействовать ту или иную функцию платформы, нужно знать пространство имен, содержащее тип, реализующий нужную функциональность. Чтобы изменить поведение FCL-типа, обычно просто создают производный тип. Объектно-ориентированная природа .NET Framework обеспечивает мощную основу для разработки. Разработчикам не возбраняется создавать собственные пространства имен, содержащие собственные типы. Эти пространства имен и типы четко соответствуют принципам программирования, предлагаемым платформой. В сравнении с Win32-программированием такой новый подход заметно упрощает разработку ПО.

Большинство пространств имен FCL предоставляет типы, которые можно задействовать в любых видах приложений. В табл. 1-3 представлены наиболее распространенные пространства имен и краткое описание назначения их типов. Это всего лишь малая толика имеющихся пространств имен. За более подробной информацией следует обращаться к документации, поставляемой компанией Microsoft в составе различных комплектов ресурсов (SDK).

Табл. 1-3. Основные пространства имен FCL

Пространство имен	Описание содержимого
System	Все базовые типы, используемые практически в любом приложении
System.Data	Типы для обмена информацией с базами данных и для обработки данных
System.Drawing	Типы для работы с двумерной графикой; обычно применяются в приложениях Windows Forms, а также при создании картинок для страниц Web Forms
SystemIO	Типы, реализующие потоковый ввод/вывод, работу с файлами и каталогами
System.Net	Типы, реализующие низкоуровневое сетевое взаимодействие и поддерживающие стандартные протоколы Интернета
System.Runtime. InteropServices	Типы, позволяющие управляемому коду получать доступ к неуправляемым функциям ОС, таким как COM-компоненты и Win32-функции в DLL-библиотеках
System.Security	Типы, используемые для защиты данных и ресурсов
System.Text	Типы для работы с текстом в различной кодировке, например ASCII или Unicode
System.Tbreading	Типы, используемые для асинхронных операций и синхронизации доступа к ресурсам
System.Xml	Типы для обработки XML-схем и данных

Эта книга посвящена CLR и основным типам, тесно связанным с CLR, поэтому она полезна всем программистам, ориентирующимся на CLR, независимо от типа создаваемых ими приложений. Есть много хороших книг, описывающих создание приложений конкретных типов (служб Windows, Web Forms и Windows Forms). Они дадут вам прекрасную возможность начать разработку собственных приложений и помогут вам смотреть на проблему сверху вниз, поскольку они ориентированы на приложения конкретных типов, а не на платформу разработки. Я же предлагаю сведения, которые помогут смотреть на проблему снизу вверх. Прочитав эту книгу, а также книги, посвященные конкретным типам приложений, вы сможете без проблем создавать любые разновидности приложений для .NET Framework.

Общая система типов

Сейчас вам должно быть понятно, что CLR тесно связана с типами. Типы предоставляют определенную функциональность приложениям и компонентам. Они являются механизмом, позволяющим коду, написанному на одном языке программирования, взаимодействовать с кодом, написанным на другом. Поскольку типы являются основой CLR, Microsoft создала формальную спецификацию — общую систему типов (Common Type System, CTS), описывающую определение типов и их поведение.

Примечание В сущности, Microsoft подавала в Европейскую ассоциацию по стандартизации информационных и вычислительных систем (ECMA) заявку на признание стандартом CTS, а также других частей .NET Framework, в том числе форматов файлов, метаданных, промежуточного языка и механизма доступа к нижележащей платформе (P/Invoke). Стандарт получил название Common Language Infrastructure (CLI). Помимо этого, Microsoft подавала заявки на стандартизацию части библиотеки Framework Class Library, а также языки программирования C# и C++/CLI. Подробнее об этих и других отраслевых стандартах см. часть Web-сайта ассоциации ECMA, посвященную Техническому комитету 39 (*www.ecma-international.org/memento/TC39.htm*), а также тематическую страницу сайта Microsoft (*http://msdn.microsoft.com/netframework/ecma/*).

В спецификации CTS утверждается, что любой тип может содержать ноль или более членов. В части 3 книги я опишу все возможные члены очень подробно, а сейчас дам о них лишь общее представление.

- **Поле** — переменная, являющаяся частью состояния объекта. Поля идентифицируются по имени и типу.
- **Метод** — функция, выполняющая некоторое действие над объектом, зачастую изменяя его состояние. Метод имеет имя, сигнатуру и модификаторы. Сигнатура определяет соглашение о вызове метода, число параметров (и их последовательность), типы параметров, а также тип значения, возвращаемого методом.
- **Свойство** — для вызывающей стороны этот член выглядит, как поле, но в реализации типа он выглядит, как метод (или два). Свойства позволяют типу, в котором они реализованы, проверить входные параметры и состояние объекта, прежде чем предоставить доступ к своим значениям и/или вычислять зна-

чения только при необходимости. Свойства также упрощают синтаксис обращения к типу. И, наконец, свойства позволяют создавать «поля» только для чтения или только для записи.

- **Событие** — обеспечивает механизм взаимных уведомлений объектов. Так, кнопка может предоставлять событие, уведомляющее другие объекты о том, что ее щелкнули.

CTS также определяет правила видимости типов и доступа к их членам. Например, если тип помечен как *открытый* (public), он экспортируется, видим и доступен любой сборке. С другой стороны, если тип помечен как *сборочный* (assembly) (в C# — *internal*), он видим и доступен только коду той сборки, в которой реализован. Таким образом, CTS определяет правила, по которым сборки формируют границы видимости типа, а CLR реализует эти правила.

Видимый вызывающему коду тип контролирует доступ к своим членам со стороны вызывающего кода. Вот допустимые варианты управления доступом к методам или полям.

- **Закрытый** (private) — метод может вызываться другими методами только из того же типа.
- **Родовой** (family) — метод может вызываться производными типами независимо от того, в какой они сборке. Во многих языках (например, в C++ и C#) такой модификатор называется *protected*.
- **Родовой и сборочный** (family and assembly) — метод может вызываться производными типами, но только если они определены в той же сборке. Многие языки (вроде C# и Visual Basic) не реализуют такое ограничение доступа. В ассемблере IL такой модификатор, естественно, предусмотрен.
- **Сборочный** (assembly) — метод может вызываться любым кодом из той же сборки. Во многих языках этому соответствует модификатор *internal*.
- **Родовой или сборочный** (family or assembly) — метод может вызываться производными типами из любой сборки и любыми типами из той же сборки. В C# этому соответствует *protected internal*.
- **Открытый** (public) — метод доступен любому коду из любой сборки.

Кроме того, CTS определяет правила для наследования типов, виртуальных функций, времени жизни объектов и т. д. Эти правила введены для реализации семантики современных языков. На самом деле не обязательно знать правила CTS как таковые: выбранный язык представит свой синтаксис и правила для типов и установит соответствие собственного специфического синтаксиса синтаксису «языка» CLR при создании управляемого модуля.

В начале работы с CLR я решил, что лучше всего рассматривать язык и поведение кода как отдельные вещи. Язык C++ позволяет определить собственный тип со своими членами. Естественно, тот же тип с теми же членами можно определить на C# или Visual Basic. Конечно, синтаксис, используемый при определении типа, зависит от языка, но поведение типа от языка не будет зависеть, так как оно определяется общей системой типов CLR.

Чтобы пояснить эту мысль, приведу пример. CTS поддерживает только единичное наследование. Таким образом, поскольку C++ поддерживает типы, наследуемые от нескольких базовых типов, CTS не может принять и оперировать такими типами. Чтобы помочь разработчику, компилятор Visual C++ сообщает об ошиб-

ке, обнаруживая попытку создать управляемый код, включающий типы, наследуемые от нескольких базовых типов.

Еще одно правило CTS: все типы должны (в конечном счете) наследовать типу *System.Object*. Как видите, *Object* — имя типа, определенного в пространстве имен *System*. *Object* является корнем всех остальных типов, гарантируя наличие некоторой минимальной функциональности у каждого экземпляра типа. В частности, тип *System.Object* позволяет:

- сравнивать два экземпляра;
- получать хеш-код экземпляра;
- определять истинный тип экземпляра;
- получать поверхностную (побитную) копию экземпляра;
- получать текущее состояние экземпляра объекта в виде строки.

Общеязыковая спецификация

COM позволяет взаимодействовать объектам, созданным на разных языках. С другой стороны, CLR обеспечивает интеграцию языков и позволяет объектам, созданным на одном языке, быть «равноправными гражданами» кода, написанного на другом. Такая интеграция возможна благодаря стандартному набору типов, информации, описывающей тип (метаданным), и общей среде выполнения CLR.

Интеграция языков — фантастическая цель, если учесть их различия. Так, одни языки не позволяют учитывать регистр символов, другие — не допускают целые числа без знака, третьи — перегрузку операторов, четвертые — не поддерживают методы с переменным числом параметров.

Чтобы создать тип, доступный из других языков, придется задействовать лишь те возможности языка, которые гарантированно доступны в других. Чтобы помочь в этом, Microsoft определила общеязыковую спецификацию (Common Language Specification, CLS), описывающую минимальный набор возможностей, который должны реализовать производители компиляторов, чтобы их продукты работали в CLR.

CLR/CTS поддерживают гораздо больше возможностей в сравнении с подмножеством, определенным в CLS, так что, если вас не волнует межъязыковое взаимодействие, можете разрабатывать очень мощные типы, ограничиваясь лишь возможностями языка. В частности, CLS определяет правила, которым должны соответствовать видимые извне типы, чтобы к ним можно было получить доступ из любых других CLS-совместимых языков программирования. Заметьте: правила CLS не применяются к коду, доступному только из сборки, в которой он содержится. Эти принципы обобщены на рис. 1-6.

Как видите, CLR/CTS предоставляет определенный выбор. Программист может решить использовать ассемблер IL, и тогда ему будут доступны все функции CLR/CTS. Большинство других языков, таких как C#, Visual Basic и Fortran, предоставляет подмножество возможностей CLR/CTS. Минимальный набор функций, которые должны поддерживать все языки, определяется CLS.

Если вы разрабатываете тип и хотите, чтобы он был доступен другим языкам, не используйте возможности своего языка, выходящие за рамки возможностей, определяемых в CLS открытых и закрытых членов. Иначе члены вашего типа могут быть недоступны программистам, пишущим код на других языках.

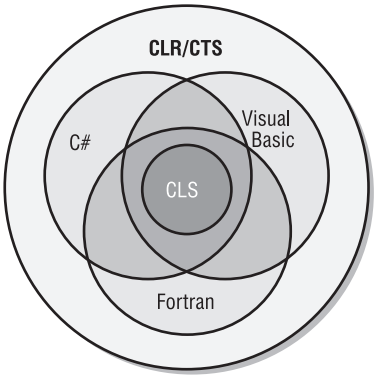


Рис. 1-6. *Языки реализуют подмножество возможностей CLR/CTS и расширяют возможности CLS (каждый в своей степени)*

В следующем коде определяется CLS-совместимый тип на C#. Однако этот тип имеет несколько несовместимых с CLS конструкций, что вызывает недовольство компилятора C#.

```
using System;

// Указываем компилятору, что нужно проверять на совместимость с CLS.
[assembly: CLSCompliant(true)]

namespace SomeLibrary {
   // Возникают предупреждения, поскольку тип открытый.
   public sealed class SomeLibraryType {

      // Предупреждение: тип, возвращаемый 'SomeLibrary.SomeLibraryType.Abc()',
      // не соответствует CLS.
      public UInt32 Abc() { return 0; }

      // Предупреждение: идентификатор 'SomeLibrary.SomeLibraryType.abc()'
      // отличается от предыдущего только регистром букв, что не соответствует CLS.
      public void abc() { }

      // Ошибки нет – метод закрытый.
      private UInt32 ABC() { return 0; }
   }
}
```

В этом коде к сборке применяется атрибут *[assembly:CLSCompliant(true)]* — он заставляет компилятор обнаруживать любые доступные извне типы, содержащие конструкции, недопустимые в других языках. При компиляции этого кода компилятор C# выдает два предупреждения. Первое выдается, так как метод *Abc* возвращает целочисленное значение без знака, а Visual Basic и некоторые другие компиляторы не могут работать с такими значениями. Второе предупреждение возникает из-за того, что данный тип имеет два открытых метода, отличающихся только регистром букв в названии (*Abc* и *abc*) и возвращаемым типом. Visual Basic и некоторые другие языки не смогут вызвать ни один из этих методов.

Если же убрать ключевое слово *public* из определения *sealed class SomeLibraryType* и перекомпилировать код, предупреждения исчезнут. Дело в том, что тип *SomeLibraryType* по умолчанию будет *internal* и, следовательно, не будет виден извне сборки. Полный список правил CLS см. в разделе «Cross-Language Interoperability» документации .NET Framework SDK.

Сформулируем саму суть правил CLS. В CLR каждый член — либо поле (данные), либо метод (действие). Это значит, что каждый язык должен «уметь» предоставлять доступ к полям и вызывать методы. Некоторые поля и методы используются особыми и стандартными способами. Чтобы упростить программирование, языки обычно предоставляют дополнительный уровень абстракции. Например, применяются такие конструкции, как перечисления, массивы, свойства, индексаторы, делегаты, события, конструкторы, деструкторы, перегрузка операторов, операции преобразования и т. д. Встречая их в исходном коде, компилятор должен перевести их в поля и методы, чтобы они были доступны CLR и другим языкам.

Рассмотрим определение типа, содержащее конструктор, деструктор, несколько перегруженных операторов, свойство, индексатор и событие. Заметьте: этот код лишь демонстрирует возможности компиляции и не является примером корректной реализации типа.

```
using System;

internal sealed class Test {
   // Конструктор
   public Test() {}

   // Деструктор
   ~Test() {}

   // Перегрузка операторов
   public static Boolean operator == (Test t1, Test t2) {
      return true;
   }
   public static Boolean operator != (Test t1, Test t2) {
      return false;
   }

   // Перегрузка оператора
   public static Test operator + (Test t1, Test t2) { return null; }

   // Свойство
   public String AProperty {
      get { return null; }
      set { }
   }

   // Индексатор
   public String this[Int32 x] {
      get { return null; }
      set { }
   }
```

```
    // Событие
    event EventHandler AnEvent;
}
```

Результатом компиляции будет тип, в котором определены некоторые поля и методы. В этом легко убедиться, исследовав результирующий управляемый модуль, используя дизассемблер IL Disassembler (ILDasm.exe), поставляемый с .NET Framework SDK (рис. 1-7).

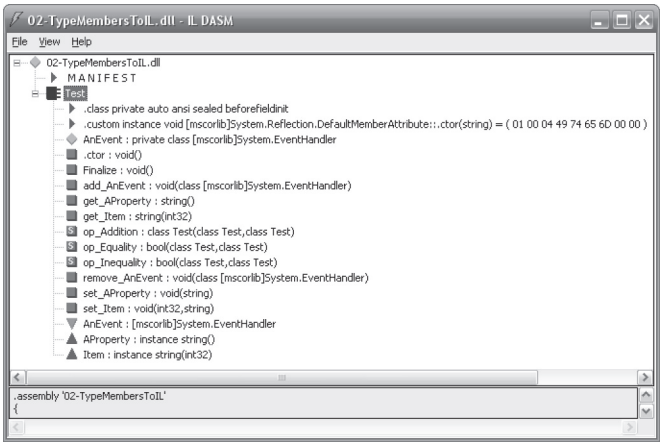


Рис. 1-7. *ILDasm показывает поля и методы типа Test (полученные из метаданных)*

Конструкции языка программирования отображаются в соответствующие поля и методы CLR (табл. 1-4).

Табл. 1-4. Поля и методы типа Test (полученные из метаданных)

Член	Тип члена	Эквивалентная языковая конструкция
AnEvent	Поле	Событие; имя поля — *AnEvent*, а тип — *System.EventHandler*
.ctor	Метод	Конструктор
Finalize	Метод	Деструктор
add_AnEvent	Метод	Метод-аксессор *add* события
get_AProperty	Метод	Метод-аксессор *get* свойства
get_Item	Метод	Метод-аксессор *get* индексатора
op_Addition	Метод	Оператор «+»
op_Equality	Метод	Оператор «==»
op_Inequality	Метод	Оператор «!=»
remove_AnEvent	Метод	Метод-аксессор *remove* события
set_AProperty	Метод	Метод-аксессор *set* свойства
set_Item	Метод	Метод-аксессор *set* индексатора

Дополнительные элементы типа *Test*, не приведенные в таблице, — *.class*, *.custom*, *AProperty* и *Item* — связаны с прочими метаданными типа. Они не отображаются на поля или методы, а лишь предоставляют дополнительные сведения о типе, к

которому имеют доступ CLR, языки программирования или инструменты. Например, какой-нибудь инструмент может обнаружить, что тип *Test* предоставляет событие *AnEvent*, представленное двумя методами (*add_AnEvent* и *remove_AnEvent*).

Взаимодействие с неуправляемым кодом

У .NET Framework масса преимуществ перед другими платформами разработки. Однако лишь немногие компании решатся перепроектировать и заново реализовать свой код. Поэтому Microsoft встроила в CLR механизм, допускающий наличие в приложении управляемой и неуправляемой частей. CLR поддерживает три сценария взаимодействия.

- **Управляемый код может вызывать неуправляемую функцию из DLL** Это обеспечивается механизмом P/Invoke (от Platform Invoke). Как-никак, многие типы, определенные в FCL, сами вызывают функции, экспортируемые библиотеками Kernel32.dll, User32.dll и другими. Во многих языках реализован механизм, упрощающий вызов неуправляемых функций, содержащихся в DLL, из управляемого кода. Так, C#- или Visual Basic-приложение может вызвать функцию *CreateSemaphore*, экспортируемую Kernel32.dll.
- **Управляемый код может использовать существующий COM-компонент (сервер)** Многие компании уже реализовали многочисленные неуправляемые COM-компоненты. Используя библиотеку типов компонента, можно создать управляемую сборку, описывающую COM-компонент. Управляемый код может обращаться к типу в такой управляемой сборке, как к любому управляемому типу. Подробности см. в описании утилиты TlbImp.exe в документации к .NET Framework SDK. Если у вас нет библиотеки типов или нужен больший контроль над тем, что сгенерировала утилита TlbImp.exe, можете вручную создать тип (в исходном коде), который CLR будет использовать для корректного взаимодействия. Например, можно задействовать COM-компоненты DirectX в программе на C#.
- **Неуправляемый код может использовать управляемый тип (сервер)** Масса существующего неуправляемого кода требует наличия COM-компонентов для своей нормальной работы. Гораздо проще реализовать их, используя управляемый код: тогда не нужно иметь дело с подсчетом ссылок и интерфейсами. Например, можно создать элемент управления ActiveX или расширение оболочки на C#. Подробности см. в описании утилит TlbExp.exe и RegAsm.exe в документации к .NET Framework SDK.

В дополнение к этим трем сценариям компилятор C++/CLI (версия 14) поддерживает новый параметр командной строки */clr*, указывающий компилятору, что нужно генерировать IL-код, а не низкоуровневые машинные команды. Вы можете перекомпилировать имеющийся код на C++ с этим новым параметром. Новый код потребует для своего выполнения CLR, и теперь его можно изменить, добавляя возможности, предлагаемые CLR.

Параметр */clr* не позволит скомпилировать в IL методы, содержащие встроенные ассемблерные команды (с ключевым словом *__asm*), принимающие переменное число аргументов, вызывающие *setjmp* или содержащие встроенные процедуры (такие как *__enable*, *__disable*, *_ReturnAddress* и *_AddressOfReturnAddress*). Полный

список конструкций, которые компилятор C++/CLI не сможет скомпилировать в IL, см. в документации к компилятору. Когда компилятор не может скомпилировать метод в IL, он компилирует его в x86, так что приложение по-прежнему работает.

Имейте в виду, что хотя создаваемый IL-код является управляемым, о данных этого сказать нельзя, то есть для объектов данных не выделяется память в управляемой куче и они не утилизируются сборщиком мусора. По сути, для типов, являющихся данными, не создаются метаданные, и имена методов таких типов искажаются.

В следующем C-коде вызывается стандартная библиотечная функция *printf*, а также метод *WriteLine* типа *System.Console*. Тип *System.Console* определен в FCL. Таким образом, код на C/C++ может использовать библиотеки C/C++ наряду с управляемыми типами.

```
#include <stdio.h>         // Для printf.
#using <mscorlib.dll>      // Для управляемых типов, определенных в этой сборке.
using namespace System;    // Для упрощенного доступа к типам
                           // из пространства имен System.

// Реализуем main, обычную функцию C/C++.
void main() {

   // Вызываем библиотечную функцию C printf.
   printf("Displayed by printf.\r\n");

   // Вызываем метод WriteLine FCL-типа System.Console.
   Console::WriteLine("Displayed by Console::WriteLine.");
}
```

Скомпилировать этот код проще простого. Если он хранится в файле ManagedCApp.cpp, выполняем в командной строке:

```
cl /clr ManagedCApp.cpp
```

Результатом является файл сборки ManagedCApp.exe. Запустив ManagedCApp.exe, увидим:

```
C:\>ManagedCApp
Displayed by printf.
Displayed by Console::WriteLine.
```

Запустив ILDasm.exe для этого файла, увидим все глобальные функции и поля, определенные в сборке. Понятно, что компилятор многое генерирует автоматически. Если дважды щелкнуть метод *Main*, ILDasm выведет IL-код:

```
.method assembly static int32
  modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl) main() cil managed
{
  .vtentry 70 : 1
  // Code size       23 (0x17)
  .maxstack  1
  IL_0000: ldsflda valuetype '<CppImplementationDetails>'.$ArrayType$$$BY0BH@$$CBD
```

```
                        modopt([mscorlib]System.Runtime.CompilerServices.IsConst)
                        '??_C@_OBH@GBHIFCOF@Displayed?5by?5printf?4?$AN?6?$AA@'
  IL_0005:  call   vararg int32

   modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
                      printf(
                        int8
                        modopt(

   [mscorlib]System.Runtime.CompilerServices.IsSignUnspecifiedByte)
                        modopt(
                          [mscorlib]System.Runtime.CompilerServices.IsConst)*
                        )
  IL_000a:  pop
  IL_000b:  ldstr    "Displayed by Console::WriteLine"
  IL_0010:  call   void [mscorlib]System.Console::WriteLine(string)
  IL_0015:  ldc.i4.0
  IL_0016:  ret
} // end of method 'Global Functions'::main
```

Выглядит это не слишком привлекательно, так как компилятору пришлось генерировать порядочно специального кода, чтобы все это работало. Однако, как видите, вызывается и *printf*, и *Console.WriteLine*.

ГЛАВА 2

Компоновка, упаковка, развертывание и администрирование приложений и типов

Прежде чем перейти к главам, описывающим разработку программ для Microsoft .NET Framework, мы обсудим создание, упаковку и развертывание приложений и их типов. В этой главе мы сосредоточимся на основах создания компонентов, предназначенных исключительно для ваших приложений. В главе 3 я расскажу 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 я также расскажу о том, как администратор может влиять на исполнение приложения и его типов.

Современные приложения состоят из типов. Составляющие приложения типы, как правило, создаются самими разработчиками или Microsoft. Помимо этого, процветает целая отрасль создателей компонентов, разрабатывающих их для продажи; компоненты обычно призваны упростить клиентам разработку собственных приложений. Типы, реализованные при помощи языка, ориентированного на общеязыковую исполняющую среду (CLR), способны «бесшовно» работать друг с другом, базовый класс такого типа даже можно написать на другом языке программирования.

Среди прочего в этой главе я объясню, как эти типы создают и упаковывают в файлы, предназначенные для развертывания. По ходу изложения я дам краткий исторический очерк некоторых проблем, решенных с приходом .NET Framework.

Задачи развертывания в .NET Framework

Все годы своего существования Windows славилась как нестабильная и чрезмерно сложная ОС. Такая репутация, заслуженная или нет, сложилась по ряду причин. Во-первых, все приложения используют динамически подключаемые библиотеки (DLL), созданные Microsoft и другими производителями. Поскольку приложение исполняет код, созданный 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. Предполагают, что новые файлы обеспечивают «обратную совместимость» с прежними, 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. Что до меня, то я решил вовсе не пробовать некоторые приложения из опасения, что они нанесут вред самым важным для меня программам.

Второй фактор, повлиявший на репутацию Windows, — сложности при установке приложений. Большинство приложений при установке не оставляет незатронутой ни одну из частей ОС. При установке приложения происходит, например, 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. Кроме того, 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.

Третья причина — безопасность. При установке приложений записывается множество файлов, созданных самыми разными компаниями. Кроме того, многие так называемые «Web-приложения» часто сами загружают и устанавливают код из Интернета, о чем пользователи не уведомляются. 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 повреждений, которые они в принципе могут нанести их компьютерам. Чтобы пользователи чувствовали себя спокойнее, в системе должны быть встроенные функции безопасности, позволяющие явно разрешать или запрещать доступ к системным ресурсам коду, созданному теми или иными компаниями.

Как будет показано в этой и следующей главах, .NET Framework устраняет «ад DLL» и делает существенный шаг вперед к решению проблемы, связанной с распределением сведений о состоянии приложения по всей операционной системе. Так, в отличие от COM, компонентам больше не требуется хранить свои параметры в реестре. К сожалению, приложениям пока еще требуются ссылки и ярлыки, но в будущих версиях Windows и эта проблема, вероятно, будет снята. Усовершенствование

системы защиты связано с новой моделью безопасности .NET Framework — *безопасностью доступа на уровне кода* (code access security). Если безопасность Windows основана на идентификации пользователя, то безопасность доступа к коду — на идентификации сборки (по строгому имени, о чем говорится в главе 3) и физического места, откуда загружается сборка. Так что пользователь может, к примеру, решить доверять сборкам, опубликованным Microsoft, но вообще не доверять никаким сборкам, загруженным из Интернета. Как видите, .NET Framework предоставляет пользователям намного больше возможностей по контролю над тем, что устанавливается и выполняется на их машинах, чем когда-либо давала им ОС Windows.

Компоновка типов в модуль

В этом разделе рассказывается, как сделать из файла исходного текста с разными типами файл, пригодный для развертывания. Для начала рассмотрим такое 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 *Main* находится ссылка на другой тип — *System.Console* — тип, реализованный Microsoft. Код на языке IL, реализующий методы этого типа, находится в файле MSCorLib.dll. Таким образом, наше приложение определяет свой тип, а также использует тип, созданный другой компанией.

Чтобы скомпоновать это приложение-пример, сохраните показанный выше код, скажем, в файле Program.cs, и исполните в командной строке:

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

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

При обработке файла с исходным текстом компилятор C# обнаруживает ссылку на метод *WriteLine* типа *System.Console*. На этом этапе компилятор должен убедиться, что такой тип где-то существует и что у него есть метод *WriteLine*. Компилятор также проверяет, что типы аргументов, ожидаемых методом *WriteLine*, совпадают с предоставленными программой. Поскольку тип не определен в исходном коде на C#, компилятору C# нужно передать набор сборок, которые позволят ему разрешить все ссылки на внешние типы. В показанной выше 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, представляющие байты, целочисленные, символьные, строковые и другие значения. В действительности, эти типы используются так часто, что компилятор C# ссылается на эту сборку автоматически. Иначе говоря, 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 тот же 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 Microsoft используют именно этот параметр при компоновке сборки MSCorLib.dll. Так, 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 же он из себя представляет? Начинающим достаточно знать, что это стандартный PE-файл (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-то с ним сделать. Windows поддерживает два типа приложений: консольные (CUI) и с графическим интерфейсом (GUI). Поскольку я указал параметр */t:exe*, компилятор C# создаст консольное приложение. Чтобы заставить компилятор сделать приложение с графическим интерфейсом, нужен 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 CSC.exe компилятор открывает файл параметров и использует 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 «@». Допустим, есть файл параметров MyProject.rsp со следующим текстом:

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

Чтобы CSC.exe использовал эти параметры, нужно вызвать файл так:

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

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

Компилятор C# способен принимать несколько файлов 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 парамет-

ров 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 пользуются более высоким приоритетом, чем указанные в локальных файлах параметров.

При установке .NET Framework стандартный (по умолчанию) глобальный файл CSC.rsp устанавливается в каталоге *%SystemRoot%\Microsoft.NET\Framework\vX.X.X* (где X.X.X — версия устанавливаемой платформы .NET Framework). В версии 2.0 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.Vsa.dll
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
/r:System.Data.dll
/r:System.Data.OracleClient.dll
/r:System.Data.SqlXml.dll
/r:System.Deployment.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.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.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.dll
/r:System.Web.Mobile.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.Dll
/r:System.Xml.dll
```

В глобальном файле CSC.rsp есть ссылки на все перечисленные сборки, поэтому не нужно указывать их явно с использованием параметра */reference*. Этот файл параметров исключительно удобен для разработчиков, так как позволяет использовать все типы и пространства имен, определенные в различных опубликованных 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 работы во время 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):

1. Рабочий каталог.
2. Каталог, содержащий файл самого компилятора (CSC.exe). MSCorLib.dll всегда извлекается из этого каталога. Путь примерно такой: *%SystemRoot%\Microsoft.NET\Framework\v2.0.50727*.
3. Все каталоги, указанные с использованием параметра */lib* компилятора.
4. Все каталоги, указанные в переменной окружения LIB.

Конечно, вы вправе добавлять собственные параметры в глобальный файл CSC.rsp, но это сильно усложнит репликацию среды компоновки между разными машинами — нужно не забыть обновлять CSC.rsp на всех машинах, используемых для сборки приложений. Разрешена и другая крайность: вообще игнорировать как локальный, так и глобальный файлы CSC.rsp, указав в командной строке параметр */noconfig*.

Несколько слов о метаданных

Теперь мы знаем, какого вида PE-файл мы создали. Но что именно находится в файле Program.exe? Управляемый PE-файл состоит из 4 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 ниже), указывающий метод точки входа в этот модуль, если это исполнимый CUI- или GUI-файл, а также необязательную сигнатуру строгого имени (о строгих именах см. главу 3). Наконец, заголовок содержит размер и смещение некоторых таблиц метаданных, расположенных в модуле. Чтобы узнать точный формат заголовка CLR, изучите определение *IMAGE_COR20_HEADER* в файле CorHdr.h.

Метаданные — это блок двоичных данных, состоящий из нескольких таблиц. Существуют три категории таблиц: определений, ссылок и деклараций. В табл. 2-1 приводится описание некоторых наиболее распространенных таблиц определений, существующих в блоке метаданных модуля.

Табл. 2-1. Стандартные таблицы определений, входящие 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 GUID, созданного компилятором). Это позволяет переименовывать файл, не теряя сведений о его исходном имени. Однако настоятельно рекомендуется не переименовывать файл, иначе 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 (public, private и т. д.) и указывает на записи таблиц MethodDef, PropertyDef и EventDef, содержащие соответственно сведения о методах, свойствах и событиях этого типа
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 и тип
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 ссылается исходный текст. 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.

Табл. 2-2. Стандартные 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 хеш. Этот хеш служит контрольной суммой битов, составляющих сборку, на которую ссылается код. CLR полностью игнорирует этот хеш и, вероятно, будет игнорировать его в будущем

Табл. 2-2. *(окончание)*

Имя таблицы ссылок	Описание
ModuleRef	Содержит по одной записи для каждого PE-модуля, в котором реализованы 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, где определен 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. Выше я упоминал, что в состав метаданных также входят таблицы декларации. Их мы обсудим чуть позже.

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

```
ILDasm Program.exe
```

Запустится ILDasm.exe, и загрузится сборка Program.exe. Чтобы вывести метаданные в читабельном виде, выберите в меню команду View/MetaInfo/Show! (или нажмите Ctrl+M). В результате появится такая 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] [AnsiClass]
               [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)
      CallCnvntn: [DEFAULT]
      ReturnType: Void
      No arguments.

   Method #2 (06000002)
   ---------------------------
      MethodName: .ctor (06000002)
      Flags     : [Public] [HideBySig] [ReuseSlot] [SpecialName]
                  [RTSpecialName] [.ctor]  (00001886)
      RVA       : 0x0000205c
      ImplFlags : [IL] [Managed]  (00000000)
      CallCnvntn: [DEFAULT]
      hasThis
      ReturnType: Void
      No arguments.

TypeRef #1 (01000001)
---------------------------
Token:            0x01000001
ResolutionScope:  0x23000001
TypeRefName:      System.Object
   MemberRef #1 (0a000004)
   ---------------------------
      Member: (0a000004) .ctor:
      CallCnvntn: [DEFAULT]
      hasThis
      ReturnType: Void
      No arguments.

TypeRef #2 (01000002)
---------------------------
Token:            0x01000002
ResolutionScope:  0x23000001
TypeRefName:      System.Runtime.CompilerServices.CompilationRelaxationsAttribute
   MemberRef #1 (0a000001)
   ---------------------------
      Member: (0a000001) .ctor:
      CallCnvntn: [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:
      CallCnvntn: [DEFAULT]
      hasThis
      ReturnType: Void
      No arguments.
TypeRef #4 (01000004)
---------------------------
Token:            0x01000004
ResolutionScope:  0x23000001
TypeRefName:      System.Console
   MemberRef #1 (0a000003)
   ---------------------------
      Member: (0a000003) WriteLine:
      CallCnvntn: [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 сам обрабатывает таблицы метаданных и комбинирует информацию, поэтому пользователю не приходится заниматься синтаксическим разбором необработанных табличных данных. Например, в приведенном выше дампе видно, что, показывая строку таблицы TypeDef, ILDasm выводит перед первой записью таблицы TypeRef определение соответствующего члена.

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

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

лучше поймете общеязыковую исполняющую среду CLR и ее возможности. В этой книге я еще не раз буду использовать ILDasm.

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

```
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 meta-data 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 размеры как самого файла (в байтах), так и 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 в ILDasm.exe есть ошибка, искажающая отображаемую информацию о размере файла. В частности, нельзя доверять сведениям в строке Unaccounted.

Объединение модулей для создания сборки

Файл Program.exe — не просто PE-файл с метаданными, а еще и *сборка* (assembly), то есть набор из одного или нескольких файлов с определениями типов и файлами ресурсов. Один из файлов сборки выбирают для хранения ее декларации. *Декларация* (manifest) — это еще один набор таблиц метаданных, которые в основном содержат имена файлов, составляющих сборку. Эти таблицы также 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 декларации, а затем получает из декларации имена остальных файлов сборки. Некоторые характеристики сборки стоит запомнить:

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

У отдельных файлов сборки, кроме файла с таблицами метаданных декларации, 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 примере выше, 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 Microsoft понадобилось вводить новое понятие — «сборка». Дело в том, что сборка позволяет разграничить логическое и физическое понятия повторно используемых типов. Допустим, сборка состоит из нескольких типов. При этом типы, используемые чаще всех, можно поместить в один файл, а используемые 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 использует. Так, компания, разработчик ПО, специализирующаяся на элементах управления пользовательского интерфейса, может реализовать в отдельном модуле активно используемые типы Active Accessibility (необходимые для соответствия требованиям логотипа Microsoft). Загружать этот модуль потребуется лишь тем, кому нужны специальные возможности.

Можно настроить приложение так, чтобы оно загружало файлы сборки, определив в его конфигурационном файле элемент codeBase (см. о нем 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 кеше загруженных файлов. Если он там есть, то загружается, нет — CLR использует для загрузки файла в кеш адрес, указанный URL. Если не удается найти нужный файл, CLR генерирует исключение *FileNotFoundException*.

Я нашел три аргумента в пользу применения многофайловых сборок. Они позволяют следующее.

- *Распределять типы на несколько файлов* Значит, можно избирательно загружать нужные файлы согласно сценарию загрузки из Интернета, а также частично упаковывать и развертывать типы, варьируя функциональность приложения.
- *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 мы обсудим ниже). Можно включать в сборки данные в любом формате: текстовом, в виде таблиц Microsoft Excel или Microsoft Word, а также любом другом при условии, что приложение способно анализировать содержимое этого файла.
- *Создавать сборки, состоящие из типов, написанных на разных языках программирования* При компиляции исходного текста на 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, а остальные — на других языках программирования. Затем при помощи соответствующего инструмента все эти модули можно объединить в одну сборку. Использующий такую сборку разработчик видит 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 ILAsm.exe и передать ему полученные файлы, и утилита выдаст файл, содержащий все типы. Для этого компилятор исходного текста должен генерировать только IL-код, поэтому эту методику нельзя использовать, скажем, с Visual C++.

Внимание! Подводя итог, можно сказать, что сборка — это единица повторного 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 декларацией, CLR может определить, какие файлы сборки содержат типы и ресурсы, на которые ссылается приложение. Любому потребителю сборки надо узнать лишь имя файла, содержащего декларацию, после чего он сможет, не нарушая работы приложения, абстрагироваться от особенностей распределения содержимого сборки по файлам, которое со временем может изменяться.

При работе с многими типами, совместно использующими одну версию и набор параметров безопасности, рекомендуется размещать все типы в одном файле, не распределяя их на несколько файлов, не говоря уже о разных сборках. Причина — производительность. 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 уменьшению фрагментации адресного пространства процесса. Ну, и наконец, nGen.exe лучше оптимизирует код, если обрабатываемые файлы крупнее по размеру.

Чтобы скомпоновать сборку, нужно выбрать один из PE-файлов, который станет хранителем декларации. Можно также создать отдельный PE-файл, в котором не будет ничего, кроме декларации. В табл. 2-3 перечислены таблицы метаданных декларации, наличие которых превращает управляемый модуль в сборку.

Табл. 2-3. Таблицы метаданных декларации

Имя таблицы метаданных декларации	Описание
AssemblyDef	Состоит из единственной записи, если модуль идентифицирует сборку. Запись включает имя сборки (без указания расширения и пути), сведения о версии (старший и младший номера версии, номер компоновки и редакции), региональные стандарты, флаги, хеш-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), хеш 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 (скажем, .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

Декларация позволяет потребителям сборки абстрагироваться от особенностей распределения ее содержимого и делает сборку самоописываемой. Заметьте также: файл, содержащий декларацию, «знает», какие файлы составляют сборку, но отдельные файлы «не знают», что они включены в сборку.

Примечание Файл сборки, содержащий декларацию, также содержит таблицу AssemblyRef. В ней хранятся записи с описанием всех сборок, на которые ссылаются файлы данной сборки. Это позволяет инструментам, открыв декларацию 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 заставляет компилятор генерировать единый PE-файл с таблицами метаданных декларации. 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 создавать PE-файл без таблиц метаданных. При использовании этого параметра всегда получается DLL-файл в формате PE. Чтобы получить доступ к типам такого файла, его нужно поместить в сборку. При указании параметра */t:module* компилятор C# по умолчанию присваивает выходному файлу расширение *.netmodule*.

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

Есть несколько способов добавления модуля в сборку. Если PE-файл с декларацией собирается при помощи компилятора C#, можно применить параметр */addmodule*. Чтобы понять, как создают многофайловые сборки, рассмотрим пример. Допустим, есть два файла с исходным текстом:

- RUT.cs, содержащий редко используемые типы;
- FUT.cs, содержащий 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 библиотеку PE DLL, но CLR не сможет просто загрузить ее.

Теперь скомпилируем в отдельном модуле часто используемые типы и сделаем его хранителем декларации сборки, так как к расположенным в нем типам обращаются довольно часто. Фактически теперь этот модуль представляет 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 файл PE DLL с таблицами метаданных декларации 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 декларации сведения об этом файле, а также занести в таблицу ExportedTypesDef сведения об открытых экспортируемых типах этого файла.

Завершив работу, компилятор создаст пару файлов (рис. 2-1). Модуль справа содержит декларацию.

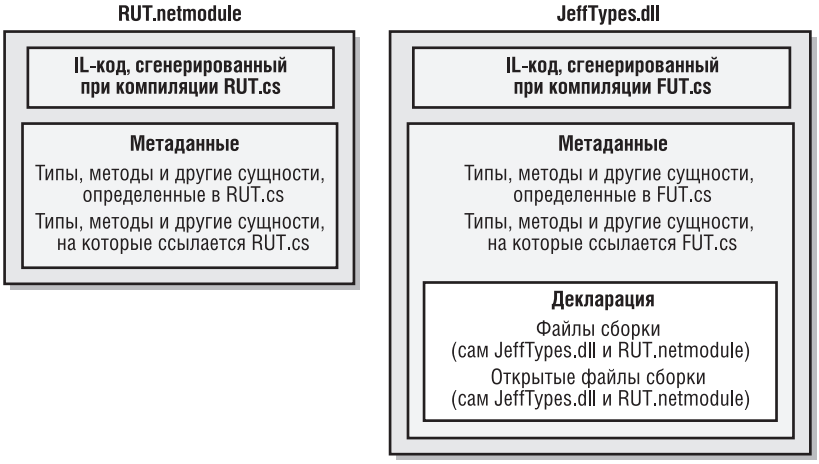


Рис. 2-1. *Многофайловая сборка из двух управляемых модулей и декларации*

Файл RUT.netmodule содержит IL-код, сгенерированный при компиляции RUT.cs. Этот файл также 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 JeffTypes.dll и RUT.netmodule). Таблицы метаданных декларации также включают описание всех открытых типов, экспортируемых файлами JeffTypes.dll и RUT.netmodule.

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

Скомпоновав сборку JeffTypes.dll, можно изучить ее таблицы метаданных декларации 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 ILDasm.exe, в выводимой утилитой информации вы увидите таблицы FileDef и ExportedTypesDef. Вот как они выглядят:

```
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[eference]: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# создал бы 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 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 декларацией. Если этот файл описывает типы, к которым обращается вызванный метод, срабатывают внутренние механизмы CLR, и нужные типы становятся доступными. Если в декларации 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, *необязательно* для его работы.

Добавление сборок в проект в среде Visual Studio

Если проект создается в Visual Studio, потребуется добавить в проект все сборки, на которые он ссылается. Для этого откройте окно **Solution Explorer**, щелкните правой кнопкой проект, на который надо добавить ссылку, и выберите **Add Reference**. Откроется диалоговое окно **Add Reference** (рис. 2-2).

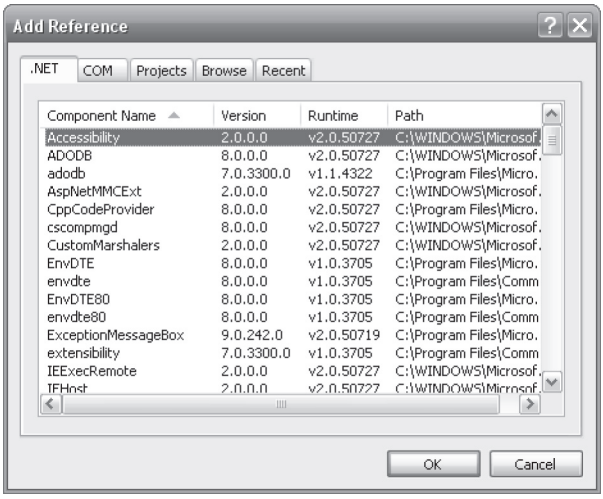


Рис. 2-2. *Диалоговое окно Add Reference в Visual Studio*

Чтобы добавить в проект ссылки на сборку, выберите ее в списке. Если в списке нет нужной сборки, щелкните кнопку **Browse**, чтобы найти ее (файл с декларацией). Вкладка **COM** в диалоговом окне **Add Reference** позволяет получить доступ к неуправляемому COM-серверу из управляемого кода через прокси-класс, 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*).

Использование утилиты Assembly Linker

Вместо компилятора C# для создания сборки можно задействовать Assembly Linker (компоновщик сборок), 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 когда требования к упаковке сборки просто не известны на момент компоновки. 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 EXE или DLL PE, которые не содержат ничего, кроме декларации, описывающей 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.

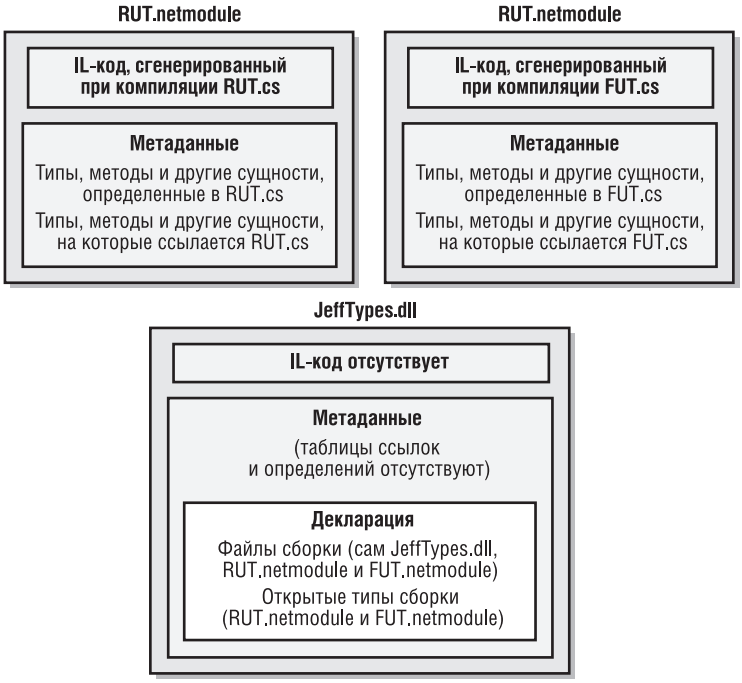


Рис. 2-3. *Многофайловая сборка из трех управляемых модулей и декларации*

В этом примере два из трех отдельных модулей, RUT.netmodule и FUT.netmodule, сборками не являются (так как не содержат таблиц метаданных декларации). Третий же — JeffTypes.dll — это небольшая PE DLL (поскольку она скомпонована с параметром */t[arget]:library*), в которой нет IL-кода, а только таблицы метаданных декларации, 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 Assembly Linker не «умеет» объединять несколько файлов в один.

Утилита AL.exe может генерировать консольные PE-файлы и PE-файлы с графическим интерфейсом (используя параметры */t[arget]:exe* или */t[arget]: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 Assembly Linker параметр командной строки */main*. Вот 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 компонует App.cs в модуль, а вторая генерирует небольшой PE-файл App.exe с таблицами метаданных декларации. В нем также находится небольшая глобальная функция, сгенерированная AL.exe из-за наличия параметра */main:-ProgramMain*. Эта функция, __EntryPoint, содержит такой 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 App.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 декларации. Я упомянул здесь этот параметр, лишь чтобы вы знали о его существовании.

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 декларации обновляется сведениями, отражающими наличие нового ресурса.

AL.exe также поддерживает параметр */link[resource]*, который принимает файл с ресурсами. Однако параметр только обновляет таблицы декларации 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 добавляют, чтобы Проводник (Windows Explorer) мог отображать значок для управляемого исполняемого файла.

Ресурсы со сведениями о версии сборки

Когда AL.exe или CSC.exe генерирует сборку в виде PE-файла, он также включает в этот файл стандартный ресурс Win32 — Version. 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 Version диалогового окна свойств файла JeffTypes.dll.

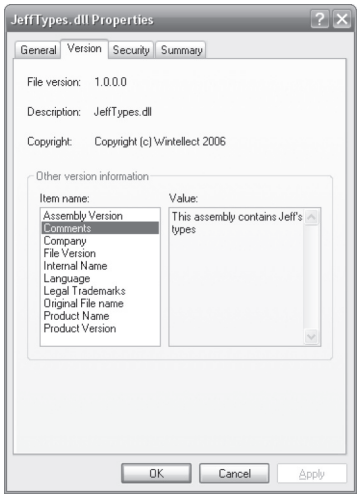


Рис. 2-4. *Вкладка Version диалогового окна свойств файла JeffTypes.dll*

При компоновке сборки следует задавать значения полей ресурса Version 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:

```
using System.Reflection;

// Задать значения полей CompanyName, LegalCopyright и LegalTrademarks
[assembly:AssemblyCompany("Wintellect")]
[assembly:AssemblyCopyright("Copyright (c) Wintellect 2006")]
[assembly:AssemblyTrademark(
             "JeffTypes is a registered trademark of Wintellect")]

// Задать значения полей ProductName и ProductVersion.
[assembly:AssemblyProduct("Wintellect (R) Jeff's Type Library")]
[assembly:AssemblyInformationalVersion("2.0.0.0")]

// Задать значения полей FileVersion, FileDescription и Comments.
[assembly:AssemblyFileVersion("1.0.0.0")]
[assembly:AssemblyTitle("JeffTypes.dll")]
[assembly:AssemblyDescription("This assembly contains Jeff's types")]

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

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

В табл. 2-4 перечислены поля ресурса Version 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 — параметры командной строки для каждого поля ресурса Version. Обратите внимание на отсутствие аналогичных параметров у компилятора C#; поэтому сведения о версии обычно задают, применяя специализированные атрибуты.

Табл. 2-4. Поля ресурса Version и соответствующие им параметры AL.exe и пользовательские атрибуты

Поле ресурса Version	Параметр AL.exe	Атрибут/комментарий
FILEVERSION	*/fileversion*	*SystemReflectionAssemblyFileVersion-Attribute*
PRODUCTVERSION	*/productversion*	*SystemReflectionAssemblyInformational VersionAttribute*
FILEFLAGSMASK	Нет	Всегда задается равным *VS_FFI_FILEFLAGSMASK* (определяется в WinVer.h как 0x0000003F)
FILEFLAGS	Нет	Всегда равен 0
FILEOS	Нет	В настоящее время всегда равен *VOS__WINDOWS32*
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*	*SystemReflectionAssemblyVersionAttribute*
Comments	*/description*	*SystemReflectionAssemblyDescriptionAttribute*
CompanyName	*/company*	*SystemReflectionAssemblyCompanyAttribute*
FileDescription	*/title*	*SystemReflectionAssemblyTitleAttribute*
FileVersion	*/version*	*SystemReflectionAssemblyVersionAttribute*
InternalName	*/out*	Задается равным заданному имени выходного файла (без расширения)
LegalCopyright *LegalTrademarks*	*/copyright* */trademark*	*SystemReflectionAssemblyCopyrightAttribute* *SystemReflectionAssemblyTrademarkAttribute*
OriginalFilename	*/out*	Задается равным заданному имени выходного файла (без пути)
PrivateBuild	*Нет*	Всегда остается пустым
ProductName	*/product*	*SystemReflectionAssemblyProductAttribute*
ProductVersion	*/productversion*	*SystemReflectionAssemblyInformational-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 пойдет речь в главе 3. Можно просто открыть файл AssemblyInfo.cs и изменить относящиеся к конкретной сборке сведения.

Номера версии

Выше я показал, что сборка может идентифицироваться по номеру версии. У этого номера одинаковый формат: он состоит из 4 частей, разделенных точками.

Табл. 2-5. Формат номеров версии

	Старший номер версии	Младший номер версии	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 Microsoft, и я настоятельно рекомендую ей следовать. В грядущих версиях CLR предполагается обеспечить более совершенную поддержку загрузки новых версий сборок и отката на предыдущую версию сборки, если новая не совместима с имеющимся приложением. В рамках механизма поддержки версий 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 является причиной большой путаницы. Позвольте 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 Microsoft (скажем, 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. Так, 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 из старшего и младшего номеров версии. Затем номера компоновки и редакции увеличивают при каждой упаковке всех сборок готового продукта.
- **AssemblyVersion** — этот номер версии хранится в декларации, в таблице метаданных AssemblyDef. CLR использует этот номер версии для привязки к сборкам, имеющим строгие имена (о них см. главу 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 определенной версии той сборки, на которую она ссылается.

Региональные стандарты

Помимо номера версии, сборки идентифицируют *региональными стандартами* (culture). Так, 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.

Табл. 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	Британский английский
en	US	Американский 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 определена культура, называют сборкой с *нейтральными региональными стандартами* (culture neutral).

При создании приложения, ресурсы которого привязаны к региональным стандартам, Microsoft настоятельно рекомендует создать одну сборку, в которой объединить код и ресурсы приложения по умолчанию 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 кода. Сборки, помеченные для применения в определенных региональных стандартах, называют *сопутствующими сборками* (satellite assemblies). Региональные стандарты, назначенные таким сборкам, в точности отражают региональные стандарты размещенного в ней ресурса. Следует создавать отдельные сборки для каждого регионального стандарта, который планируется поддерживать.

Обычно сопутствующие сборки я компоную с помощью утилиты AL.exe. Использовать для этого компилятор не стоит — ведь в сопутствующей сборке не должно быть кода. Применяя 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. Так, если базовый каталог приложения *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 кодом. При желании вместо параметра */culture* утилиты AL.exe культуру можно указать в атрибуте *System.Reflection.AssemblyCulture-Attribute,* определяемом пользователем, например, так:

```
// 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 нужно получить доступ к типам или членам, расположенным в сопутствующей сборке, следует воспользоваться методом отражения (см. главу 22).

Развертывание простых приложений (закрытое развертывание сборок)

На протяжении этой главы я рассказывал, как компоновать модули и объединять их в сборки. Теперь мы готовы к упаковке и развертыванию всех сборок, чтобы пользователь мог работать с приложением.

Особых средств для упаковки сборки не требуется. Легче всего упаковать набор сборок, просто скопировав все их файлы. Так, 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, от которых зависит ее работа, 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 уменьшения времени загрузки). Можно также упаковать файлы сборки в MSI-файл, предназначенный для службы установщика Windows Installer (MSIExec.exe). MSI позволяет установить сборку по требованию при первой попытке CLR загрузить эту сборку. 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 (Desktop), 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. Надеюсь, что в будущих версиях Windows с этим станет получше.

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 Web-сайт, FTP-сайт или в заданную папку на диск. 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, используя технологию 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 — большое 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 я расскажу о развертывании совместно используемых сборок, доступных нескольким приложениям.

Простое средство администрирования (конфигурационный файл)

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

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 могут сами создавать 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 создавать резервную копию файла, а администратору — без труда копировать файлы с машины на машину: достаточно скопировать нужные файлы, и административная политика также будет скопирована.

В главе 3 мы подробно изучим такой конфигурационный файл, а пока лишь коснемся его. Допустим, издатель хочет развернуть приложение вместе с файлами сборки JeffTypes, но в отдельном каталоге. Желаемая структура каталогов с файлами выглядит так:

```
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 *SystemIO.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 нашем случае — App.exe.config. Содержимое этого конфигурационного файла должно быть примерно таким:

```
<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 вложенными подкаталогами, но не может управлять другими каталогами.

Алгоритм поиска файлов сборки

В поиске сборки CLR просматривает несколько подкаталогов. Порядок при поиске сборки с нейтральными региональными стандартами таков (при условии, что *firstPrivatePath* и *secondPrivatePath* 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 вышеупомянутых 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. Так, если для файла AsmName.dll назначен региональный стандарт «en-US», порядок просмотра каталогов будет 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 в поисках 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 Web-приложения и называться Web.config. Кроме того, в каждом вложенном каталоге может быть собственный файл Web.config с унаследованными параметрами конфигурации. Например, Web-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*.
- Для сборок, которые содержат клиентские элементы управления, работающие в Microsoft Internet Explorer, HTML-страница должна содержать тег со ссылкой, у которой значение атрибута *rel* равно «Configuration», а в атрибуте href находится URL-адрес конфигурационного файла с каким угодно именем. Пример: *<LINK REL=Configuration HREF=http://www.Wintellect.com/Controls.config>*. Дополнительные сведения об этом см. в документации по .NET Framework, страница *http://msdn.microsoft.com/library/en-us/cpguide/html/cpcondeployingcommonlanguageruntimeapplicationusingie55.asp*

Как я уже говорил, параметры конфигурации применяют к конкретному приложению и конкретному компьютеру. При установке .NET Framework создает файл Machine.config. Существует по одному файлу Machine.config на каждую версию 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 Windows (обычно это C:\Windows), а <версия> — номер версии, идентифицирующий определенную версию .NET Framework (например, v2.0.50727).

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 параметров конфигурации приложения, что возможно лишь при использовании конфигурационных файлов, специфичных для приложения.

ГЛАВА 3

Совместно используемые сборки и сборки со строгим именем

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

В этой главе я сосредоточусь 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 сказано в главе 2, Windows получила репутацию нестабильной ОС 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 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. Ранее установленное и до сих пор прекрасно работавшее пользовательское приложение теперь уже не использует тот код, с которым оно создавалось и тестировалось. В итоге поведение такого приложения становится непредсказуемым, что в свою очередь расшатывает 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 использует ошибки в других программах. Если в более поздней версии кода будет исправлена какая-либо ошибка, то использующее ее приложение станет работать непредсказуемо.

Итак, вопрос в следующем: как, устраняя ошибки и добавляя к программам новые функции, в то же время гарантировать, что 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, который позволит надеяться, что все приложения при этом будут работать замечательно, а если это окажется не так, *легко* вернуть приложение в последнее состояние, в котором оно прекрасно работало.

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

Примечание Мне часто приходится консультировать людей из Microsoft, в частности пришлось много поработать с командой, работающей над архитектурой управления версиями. Эта команда отвечает за совершенствование управления версиями в CLR. Эти парни планируют изменить будущие версии CLR, чтобы управление версиями стало намного проще. К сожалению, эти изменения не попали в CLR версии 2.0 — они отложены до следующих версий. Материал этой главы по большей части не утратит актуальности и после этих изменений — просто некоторые процессы упростятся.

Два вида сборок — два вида развертывания

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

Внимание! Вы не найдете 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. Поэтому я решил обозначить такие сборки специальным термином, чтобы потом можно было недвусмысленно указать, о каких сборках идет речь.

Сборки со строгими и нестрогими именами идентичны по структуре, то есть в них используется один файловый формат — portable executable (PE) и они состоят из заголовка PE32(+), CLR-заголовка, метаданных, таблиц декларации, а также

IL-кода, который мы рассмотрели в главах 1 и 2. Оба типа сборок компонуются 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 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 я объясню, как создают и развертывают сборки со строгим именем. Сведения о типах сборок и способах их развертывания суммированы в табл. 3-1.

Табл. 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 будут поставлены вне закона. Проблема с нестрогими сборками в том, что можно создать несколько разных сборок с одним нестрогим именем. С другой стороны, 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 CLR должна автоматически проверять, обнаружив ссылку на сборку. Однако с этим связана проблема — что, если две (или больше) компаний сделают сборки с одинаковыми именами? В таком случае, если обе эти сборки будут скопированы в один общеизвестный каталог, «победит» последняя из них, а работа приложений, использовавших пер-

вую, нарушится — ведь первая была затерта второй при копировании (это и стало причиной «ада DLL» в современных Windows-системах — все DLL-библиотеки копировались в папку System32).

Очевидно, одного имени файла мало, чтобы различать две сборки. 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 хеш открытого ключа, который называют *маркером открытого ключа* (public key token). Следующие четыре строки (иногда их 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, зависимых 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 Microsoft предпочла другим способам идентификации (при помощи GUID, URL и URN) использование стандартных криптографических технологий, основанных на паре из закрытого и открытого ключей. В частности, криптографические технологии позволяют проверять целостность данных сборки при установке ее на жесткий диск, а также обеспечивают предоставление прав доступа для сборки в зависимости от ее издателя. Все эти методики мы обсудим ниже.

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 могут создавать сборки с одинаковым именем, версией и региональными стандартами, не опасаясь возникновения конфликтов.

Примечание Вспомогательный класс *System.ReflectionAssemblyName* 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*, *GetPublicKeyToken*, *SetPublicKey* и *SetPublicKeyToken*.

В главе 2 я продемонстрировал назначение 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 декларации. 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, указанных 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. Она также 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 команду:

```
SN -k MyCompany.keys
```

Эта команда заставит SN.exe создать файл MyCompany.keys, 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 выполнить SN.exe с параметром *–tp* и указать файл, содержащий только 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
00240000048000009400000006020000002400005253413100040000010001003f9d621b702111
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 хеш 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 создание сборки со строгим именем не должно вызывать затруднений. При компиляции сборки нужно задать компилятору параметр */keyfile:*<имя_файла>:

```
csc /keyfile:MyCompany.keys app.cs
```

Обнаружив в исходном тексте этот параметр, компилятор открывает заданный файл (MyCompany.keys), подписывает сборку закрытым ключом и встраивает открытый ключ в декларацию сборки. Заметьте: подписывается лишь файл сборки, содержащий декларацию, 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 **<New...>** (см. рис. 3-1).

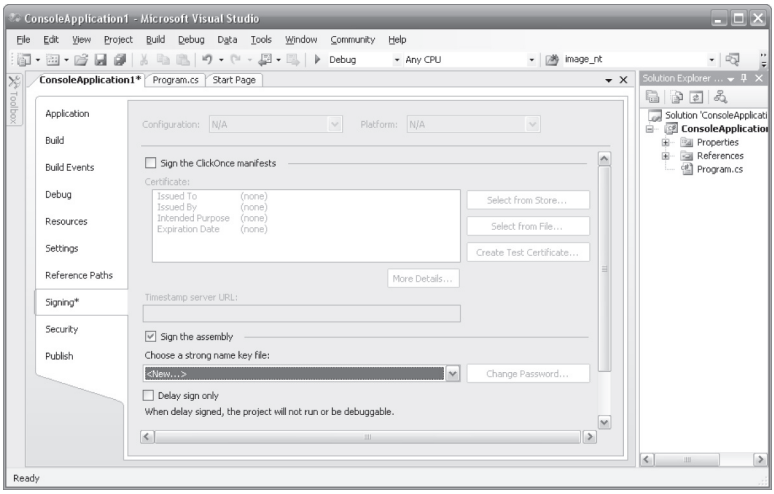


***Рис. 3-1**. Создание файла ключей и подписание сборки в Visual Studio*

Слова «подписание файла» означают здесь следующее: при компоновке сборки со строгим именем в таблицу метаданных декларации FileDef заносится список всех файлов, составляющих эту сборку. Каждый раз, когда к декларации добавляется имя файла, рассчитывается хеш содержимого этого файла, и полученное значение сохраняется вместе с именем файла в таблице FileDef. Можно заменить алгоритм расчета хеша, 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 хеша используется алгоритм SHA-1, возможностей которого должно хватать практически для любого приложения.

После компоновки PE-файла с декларацией рассчитывается хеш всего содержимого этого файла (за исключением подписи Authenticode Signature, строгого име-

ни сборки и контрольной суммы заголовка PE) (рис. 3-2). Для этой операции применяется алгоритм SHA-1, здесь его нельзя заменить никаким другим. Значение хеша подписывается закрытым ключом издателя, а полученная в результате цифровая подпись RSA заносится в зарезервированный раздел PE-файла (при расчете хеша PE-файла этот раздел исключается) и в CLR-заголовок PE-файла записывается адрес, по которому встроенная цифровая подпись находится в файле.

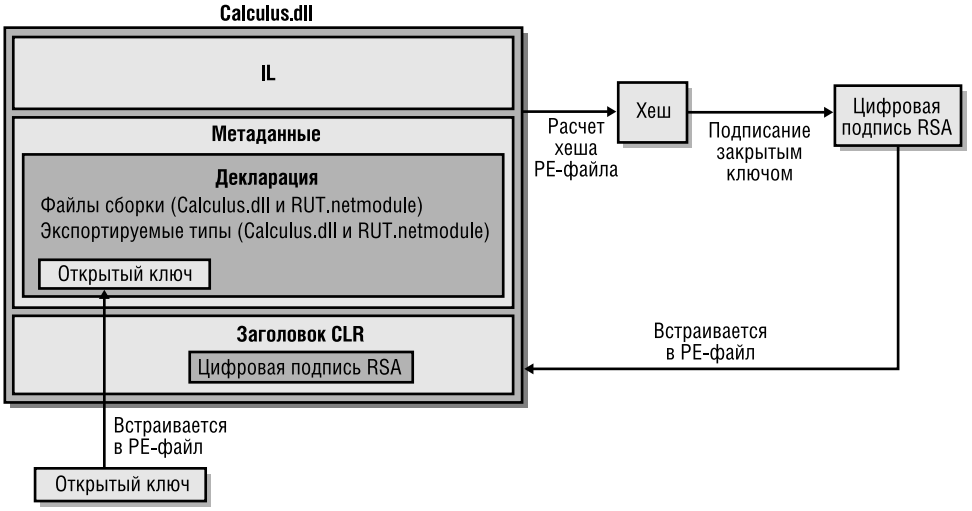


Рис. 3-2. *Подписание сборки*

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

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

Как сказано в главе 2, при компиляции исходного текста компилятор обнаруживает все типы и члены, на которые ссылается исходный текст; компилятору также 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 Microsoft рассчитывает хеш открытого ключа и берет последние 8 байт полученного хеша. В таблице AssemblyRef на самом деле хранятся именно такие, усеченные, значения открытого ключа — маркеры отрытого ключа. В общем случае разработчики и конечные пользователи намного чаще будут встречаться с маркерами, чем с полными значениями ключа.

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

Ниже показаны 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: 2.0.0.0
Major Version: 0x00000002
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)
```

Из этих сведений видно, что JeffTypes.dll ссылается на тип, расположенный в сборке с такими атрибутами:

```
"MSCorLib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

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

Взглянув на содержимое таблицы метаданных AssemblyDef файла JeffTypes.dll, 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 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 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. Ниже я объясню принцип, лежащий в основе устойчивости к несанкционированной модификации сборок со строгими именами.

Глобальный кеш 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 CLR должна автоматически проверять, обнаружив ссылку на сборку. Место, где располагаются совместно используемые сборки, называют *глобальным кешем сборок* (global assembly cache, GAC), обычно это каталог:

```
C:\Windows\Assembly
```

Каталог 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 задачи. 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 GAC чаще всего применяют инструмент GACUtil.exe. Запущенный без параметров, он отобразит такие сведения о его использовании:

```
Microsoft (R) .NET Global Assembly Cache Utility. Version 2.0.50727.42
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
```

Как видите, вызвав GACUtil.exe с параметром */i*, можно установить сборку в GAC, а если задать параметр */u*, сборка будет удалена из GAC. Заметьте: сборку с нестрогим именем даже нельзя поместить в GAC. Если передать GACUtil.exe файл сборки с нестрогим именем, утилита покажет сообщение об ошибке: «Failure adding assembly to the cache: Attempt to install an assembly without a strong name» («Ошибка при добавлении сборки в кеш: попытка установить сборку без строгого имени»).

Примечание По умолчанию манипуляции над 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 использовать 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 файл формата cabinet (.cab-файл) или сжата иным способом, то, прежде чем устанавливать файл сборки в GAC при помощи GACUtil.exe, следует распаковать его во временный файл, который следует удалить по завершении 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 GAC, используйте программу Windows Installer (MSI) версии 3 и выше, так как это единственный инструмент, способный установить сборки в GAC и гарантированно присутствующий на машине конечного пользователя. (Чтобы определить версию Windows Installer на компьютере, используйте программу MSIExec.exe.)

Внимание! Глобальное развертывание сборки путем размещения ее в GAC — это один из видов регистрации сборки в системе, хотя это никак не затрагивает реестр Windows. Установка сборок в GAC делает невозможным простое удаление, копирование, восстановление и удаление приложения. Поэтому рекомендуется избегать глобального развертывания и использовать закрытое развертывание сборок всюду, где это только возможно.

Зачем «регистрировать» сборку в 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. И все же, забавы ради, я приведу описание внутренней структуры GAC в следующем разделе.

При установке .NET Framework также устанавливается расширение проводника (ShFusion.dll). Эта программа-расширение тоже «знает» структуру GAC и показывает его содержимое в виде, понятном пользователю. Просматривая каталог *C:\Windows\Assembly* в проводнике на своем компьютере, я вижу сборки, установленные в GAC (рис. 3-3). Каждая строка списка содержит имя сборки, ее тип, номер версии, региональные стандарты (если есть), маркер открытого ключа и процессорную архитектуру.

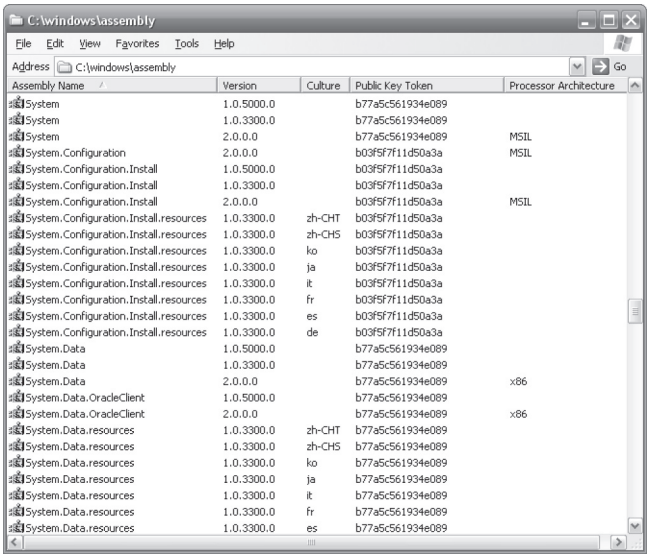


Рис. 3-3. *Просмотр сборок, установленных в GAC, при помощи расширения проводника*

Выбрав сборку, можно щелкнуть правой кнопкой мыши и вывести контекстное меню с командами **Delete** и **Properties**. Ясно, что первая удаляет файлы выбранной сборки из GAC, должным образом модифицируя при этом его структуру. Выбор второй выводит диалоговое окно вроде показанного на рис. 3-4. Отметка времени Last Modified указывает, когда сборка была добавлена в GAC. Перейдя на вкладку Version, можно увидеть диалоговое окно, представленное на рис. 3-5.

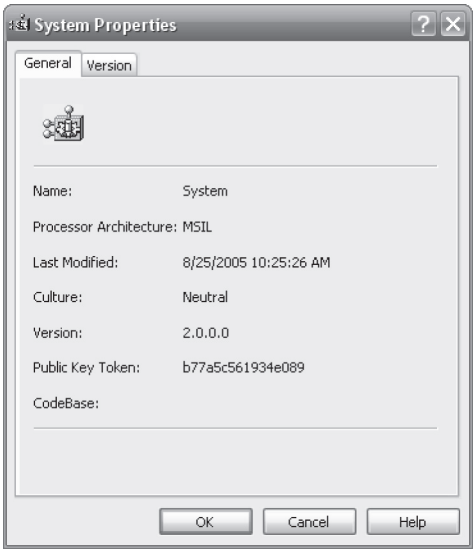


Рис. 3-4. *Вкладка General страницы свойств сборки*

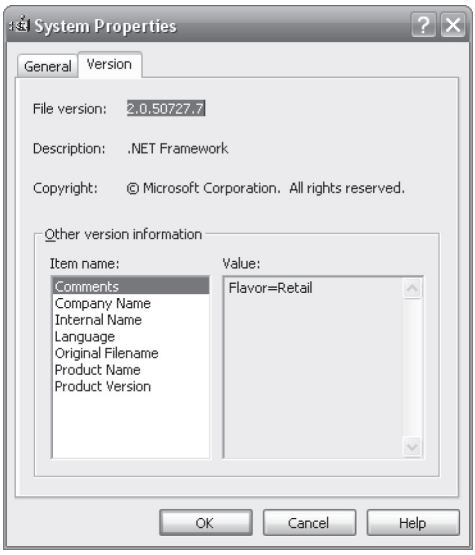


Рис. 3-5. *Вкладка Version страницы свойств сборки*

И последнее, но не менее важное замечание: если перетащить файл «строгой» сборки, содержащий декларацию, в окно проводника, расширение установит файлы этой сборки в GAC. В период тестирования некоторым разработчикам легче устанавливать сборки в GAC именно так, а не при помощи утилиты GACUtil.exe.

Примечание Можно отключить расширение просмотра кеша сборок Assembly Cache Viewer — для этого нужно изменить реестр: в разделе *HKEY_LOCAL_MACHINE\Software\Microsoft\Fusion* создать параметр типа DWORD по имени *DisableCacheViewer* и присвоить ему значение 1. Есть другой вариант: в каталоге *C:\Windows\Assembly* удалить скрытый файл Desktop.ini.

Внутренняя структура GAC

Роль GAC, попросту говоря, заключается в поддержании связи между сборкой со строгим именем и некоторым подкаталогом. По сути, внутренняя функция CLR принимает имя сборки, ее версию, региональные стандарты и маркер открытого ключа, возвращая путь к подкаталогу, в котором хранятся файлы указанной сборки.

Если в командной строке перейти в каталог *%SystemRoot%\Assembly*, можно увидеть несколько вложенных каталогов. Вот как выглядит дерево каталогов GAC на моем компьютере:

```
C:\Windows\Assembly\GAC
C:\Windows\Assembly\GAC_MSIL
C:\Windows\Assembly\GAC_32
C:\Windows\Assembly\GAC_64
```

Каталог *C:\Windows\Assembly\GAC* содержит сборки, созданные в версиях 1.0 и 1.1 среды CLR. Они все состоят из управляемого (IL) кода или содержат управля-

емый и машинный x86-код (например, когда сборка была создана компилятором Microsoft C++ с Managed Extensions). Сборки этого каталога могут выполняться только в 32-разрядном адресном пространстве, поэтому способны работать только в версии x86 Windows или, с применением технологии WoW64, — на 64-разрядной версии Windows.

Каталог *C:\Windows\Assembly\GAC_MSIL* содержит сборки, созданные для версии 2.0 среды CLR. Они полностью состоят из управляемого IL-кода и способны работать в 32- и 64-разрядном адресном пространстве, поэтому могут работать в 32- и 64-разрядных версиях Windows. Также они способны выполняться в 32-разрядном адресном пространстве в 64-разрядных версиях Windows, но с применением технологии WoW64.

Каталог *C:\Windows\Assembly\GAC_32* содержит сборки, созданные для версии 2.0 среды CLR. Они содержат управляемый IL-код и машинный x86-код (например, если сборка создана компилятором Microsoft C++/CLI). Этим сборкам разрешено выполнение только в 32-разрядном адресном пространстве, поэтому они могут выполняться в x86-версии Windows или, с использованием технологии WoW64, — в 64-разрядной версии Windows.

Каталог *C:\Windows\Assembly\GAC_64* содержит сборки, созданные для версии 2.0 среды CLR. Они содержат управляемый IL-код и машинный код x64 или IA64. У меня x64-компьютер, поэтому на моей машине в этом каталоге хранятся сборки с x64-кодом. Нельзя установить в *GAC* код IA64 (Intel Itanium) на x64-машине, и наоборот. Сборкам этого каталога разрешается работа в 64-разрядном адресном пространстве, поэтому они могут работать только в 64-разрядной версии Windows с соответствующей процессорной архитектурой. На машине с 32-разрядной версией Windows этого подкаталога попросту нет.

У всех перечисленных каталогов одинаковая внутренняя структура, поэтому достаточно познакомиться с иерархией подкаталогов в одном из них. Войдя в один из подкаталогов каталога GAC_MSIL, мы увидим вложенные каталоги — по одному на каждую сборку, установленную в GAC. Содержимое каталога GAC_MSIL на моем компьютере выглядит так (несколько каталогов удалено, чтобы сохранить структуру деревьев):

```
 Volume in drive C has no label.
 Volume Serial Number is 2450-178A

 Directory of C:\WINDOWS\assembly\GAC_MSIL

09/22/2005   07:38 AM     <DIR>        .
09/22/2005   07:38 AM     <DIR>        ..
08/25/2005   10:25 AM     <DIR>        Accessibility
08/25/2005   10:25 AM     <DIR>        ADODB
08/25/2005   10:25 AM     <DIR>        AspNetMMCExt
08/25/2005   10:29 AM     <DIR>        CppCodeProvider
08/25/2005   10:25 AM     <DIR>        cscompmgd
08/25/2005   10:25 AM     <DIR>        IEExecRemote
08/25/2005   10:25 AM     <DIR>        IEHost
08/25/2005   10:25 AM     <DIR>        IIEHost
08/25/2005   10:29 AM     <DIR>        MFCMIFC80
```

```
08/25/2005   10:32 AM     <DIR>         Microsoft.AnalysisServices
08/25/2005   10:32 AM     <DIR>         Microsoft.AnalysisServices.DeploymentEngine
08/25/2005   10:29 AM     <DIR>         Microsoft.Build.Conversion
08/25/2005   10:25 AM     <DIR>         Microsoft.Build.Engine
08/25/2005   10:25 AM     <DIR>         Microsoft.Build.Framework
08/25/2005   10:25 AM     <DIR>         Microsoft.Build.Tasks
08/25/2005   10:25 AM     <DIR>         Microsoft.Build.Utilities
08/25/2005   10:32 AM     <DIR>         Microsoft.DataWarehouse.Interfaces
08/25/2005   10:33 AM     <DIR>         Microsoft.ExceptionMessageBox
08/25/2005   10:25 AM     <DIR>         Microsoft.JScript
...
08/25/2005   10:25 AM     <DIR>         System
08/25/2005   10:25 AM     <DIR>         System.Configuration
08/25/2005   10:25 AM     <DIR>         System.Configuration.Install
08/25/2005   10:25 AM     <DIR>         System.Data.SqlXml
08/25/2005   10:25 AM     <DIR>         System.Deployment
08/25/2005   10:25 AM     <DIR>         System.Design
08/25/2005   10:25 AM     <DIR>         System.DirectoryServices
08/25/2005   10:25 AM     <DIR>         System.DirectoryServices.Protocols
08/25/2005   10:25 AM     <DIR>         System.Drawing
08/25/2005   10:25 AM     <DIR>         System.Drawing.Design
08/25/2005   10:25 AM     <DIR>         System.Management
08/25/2005   10:25 AM     <DIR>         System.Messaging
08/25/2005   10:25 AM     <DIR>         System.Runtime.Remoting
08/25/2005   10:25 AM     <DIR>         System.Runtime.Serialization.Formatters.Soap
08/25/2005   10:25 AM     <DIR>         System.Security
08/25/2005   10:25 AM     <DIR>         System.ServiceProcess
08/25/2005   10:25 AM     <DIR>         System.Web.Mobile
08/25/2005   10:25 AM     <DIR>         System.Web.RegularExpressions
08/25/2005   10:25 AM     <DIR>         System.Web.Services
08/25/2005   10:25 AM     <DIR>         System.Windows.Forms
08/25/2005   10:25 AM     <DIR>         System.Xml
               1 File(s)                  0 bytes
             110 Dir(s)  13,211,459,584 bytes free
```

Если перейти к одному из этих каталогов, видно один или несколько подкаталогов. Каталог System выглядит так:

```
Volume in drive C has no label.
 Volume Serial Number is 2450-178A

 Directory of C:\WINDOWS\assembly\GAC_MSIL\System

08/25/2005  10:25 AM    <DIR>         .
08/25/2005  10:25 AM    <DIR>         ..
08/25/2005  10:25 AM    <DIR>         2.0.0.0__b77a5c561934e089
               0 File(s)              0 bytes
               3 Dir(s)  13,211,467,776 bytes free
```

В каталоге System имеется по одному подкаталогу для каждой сборки System.dll, установленной на машине. У меня установлена лишь одна версия сборки System.dll:

```
"System, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

Атрибуты сборки разделены символом подчеркивания и выводятся так: «*(Версия)_ (Идентификатор_региональных_стандартов)_(Маркер_Открытого_Ключа)*». В примере сведений о региональных стандартах нет, так что это сборка с нейтральными региональными стандартами. В каталоге расположены файлы (например, System.dll), составляющие версию сборки System со строгим именем.

Внимание! Очевидно, что вся идея GAC — исключительно в способности хранить нескольких версий сборки. Так, в GAC могут присутствовать версии 1.0.0.0 и 2.0.0.0 сборки Calculus.dll. Если приложение скомпоновано и протестировано с использованием версии 1.0.0.0 файла Calculus.dll, то для этого приложения CLR загрузит именно версию 1.0.0.0, несмотря на наличие в GAC более поздней версии этой сборки. Эта политика загрузки разных версий сборки применяется CLR по умолчанию, ее преимущество в том, что при установке новой сборки работа уже установленных приложений не нарушается. Способы изменения этой политики мы обсудим ниже.

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 я показал, как использовать компилятор CSC.exe с параметром */reference* для определения сборки, на которую должна ссылаться компонуемая сборка. Если вместе с именем файла задан полный путь к нему, CSC.exe загружает указанный файл и использует его метаданные для компоновки сборки. Как говорилось 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 CLR**. 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 от Microsoft, можно задать параметр */reference:System.Drawing.dll* при вызове CSC.exe. Компилятор проверит показанные выше каталоги и обнаружит файл System.Drawing.dll в одном каталоге с CLR, которую сам использует для создания сборки. Но несмотря на то, что при компиляции сборка находится в этом каталоге, во время выполнения эта сборка загружается из другого каталога.

Видите ли, 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 GAC. Файлы в каталоге CLR облегчают компоновку пользовательских сборок, а их копии в GAC предназначены для загрузки во время выполнения.

CSC.exe не ищет нужные для компоновки сборки в GAC, чтобы вам не пришлось задавать громоздкие пути к файлам сборки вроде C:\WINDOWS\Assembly\GAC_MSIL\System.Drawing\2.0.0.0__b03f5f7f11d50a3a\System.Drawing.dll. CSC.exe также позволяет задавать сборки при помощи не менее длинной, но чуть более изящной строки вида «System.Drawing, Version=2.0.0.0, 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 *GAC_64*. Чуть позже я расскажу, 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 хеш содержимого файла с декларацией 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 что открытый ключ подписи соответствует закрытому ключу издателя. Кроме того, система рассчитывает хеш содержимого других файлов сборки и сравнивает полученные значения с таковыми из таблицы декларации FileDef. Если хоть одно из значений не совпадает, значит, хотя бы один из файлов сборки был модифицирован и установка сборки в GAC окончится 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 найдена, возвращается путь к каталогу, в котором она находится, и загружается файл с ее декларацией. Такой механизм поиска сборок гарантирует вызывающей стороне, что во время выполнения будет загружена сборка того же издателя, который создал 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, затем, если приложение установлено при помощи MSI, CLR просит MSI найти нужную сборку. Если ни по одному из этих адресов сборка не найдена, привязка заканчивается неудачей и генерируется исключение *System.IO.FileNotFoundException*.

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 хеш. Иначе говоря, расчет хеша для файла выполняется в данном случае 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 хеша, CLR генерирует исключение *System.IO.FileLoadException*.

Отложенное подписание

Выше мы обсуждали способ получения криптографической пары ключей при помощи 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. Так, 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 подписание* (partial signing).

Отложенное подписание позволяет компоновать сборку с открытым ключом компании, не требуя закрытого ключа. В этом случае в записи таблицы AssemblyRef сборок, ссылающихся на вашу сборку, встраивается правильное значение открытого ключа, а также эти сборки корректно размещаются во внутренней структуре GAC. Не подписывая файл закрытым ключом, вы полностью лишаетесь защиты от несанкционированной модификации, так как при этом не рассчитывается

хеш 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 будет подписана закрытым ключом.

Обычно открытый ключ компании получают в виде файла и передают его утилитам, компонующим сборку. (Как говорилось в этой главе, чтобы извлечь открытый ключ из файла, содержащего пару ключей, можно вызвать утилиту SN.exe с параметром *–tp*.) Следует также указать компонующей 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** (см. рис. 3-1). При использовании AL.exe нужно задать параметр */delay[sign]*.

Обнаружив, что подписание сборки откладывается, компилятор или 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 AssembyRef 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 хеш файла не рассчитывается.

На этом этапе результирующая сборка не имеет действительной цифровой подписи. Попытка установки такой сборки в GAC окончится неудачей, так как хеш содержимого файла не был рассчитан, что создает видимость повреждения файла. 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 хеша для всех файлов сборки при ее загрузке во время выполнения. С точки зрения внутренних механизмов системы, параметр *–Vr* утилиты SN.exe размещает идентификационную информацию сборки в разделе реестра — *HKEY_LOCAL_ MACHINE\SOFTWARE\Microsoft\StrongName\Verification*.

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* и с указанием имени файла, содержащего настоящий закрытый ключ. Параметр *–R* заставляет SN.exe рассчитать хеш 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*.

Для удобства привожу полную последовательность действий по созданию сборки с отложенным подписанием.

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 ниже команду, чтобы получить возможность тестирования этой сборки, установки ее в 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 ниже команду. При желании можно установить новую версию в GAC, но не пытайтесь это сделать до выполнения п. 4.

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

4. Чтобы снова включить проверку для тестирования сборки в реальных условиях, выполните команду:

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

В начале раздела я сказал о хранении ключей организации на аппаратных носителях, например на смарт-картах. 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, нужно использовать другие параметры при обращении к CSC.exe, AL.exe и SN.exe. При компиляции (CSC.exe) вместо */keyfile* нужно задействовать параметр */keycontainer*, при компоновке (AL.exe) нужно применять */keyname* вместо */keyfile*, а при вызове SN.exe для добавления закрытого ключа к сборке, подписание которой было отложено, укажите параметр *–Rc* вместо *–R*. SN.exe поддерживает дополнительные параметры для работы с CSP.

Внимание! Отложенное подписание удобно, когда нужно выполнить какие-либо действия над сборкой до ее развертывания. Так, может понадобиться применить к сборке защитные утилиты, модифицирующие до неузнаваемости код. После подписания сборки это сделать будет нельзя, так как хеш 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 *–Rc*, чтобы завершить подписание сборки и рассчитать все необходимые хеш-значения.

Закрытое развертывание сборок со строгими именами

Установка сборок в GAC дает несколько преимуществ. GAC позволяет нескольким приложениям совместно использовать сборки, снижая в целом обращение к физической памяти. Кроме того, при помощи GAC легче развертывать новую версию сборки и заставлять все приложения использовать новую версию сборки посред-

ством реализации политики издателя (которая будет описана ниже). 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 развертывание сборки путем простого копирования невозможным.

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 «простого» 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.

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. Теперь при выполнении CLR будет знать, что совместно используемую сборку надо искать в каталоге, содержащем сборку со строгим именем. Замечу, что эту методику используют 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.

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 Web-адреса CLR автоматически загрузит указанный файл и сохранит его в кеше 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 я покажу ниже.

При установке в GAC сборки со строгим именем система проверяет, не подвергался ли файл с декларацией несанкционированной модификации. Эта проверка производится лишь раз, во время установки сборки. Напротив, когда сборка со строгим именем загружается не из GAC, а из другого каталога, CLR всегда проверяет файл с декларацией сборки, чтобы гарантировать неприкосновенность содержимого файла, что несколько снижает быстродействие.

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, скажем, 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. Ниже показан IL-код метода *Main*. Чтобы получить его, я запустил ILDasm.exe, выбрал в меню **View** команду **Show Bytes** и дважды щелкнул метод *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 записи в другой таблице, AssemblyRef. Эта запись такова: «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 (этот 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 декларации текущей сборки. При этом исполняющая среда ищет его в каталоге, откуда был загружен файл, содержащий декларацию сборки. Файл загружается, проверяется его хеш, 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 декларацией этой сборки. Если в файле с декларацией нужного 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 перечисляет сборки только по имени, без расширения. Во время привязки к сборке система автоматически добавляет к имени файла расширение .dll или .exe, пытаясь найти файл, проверяя каталоги по алгоритму, описанному в главе 2.

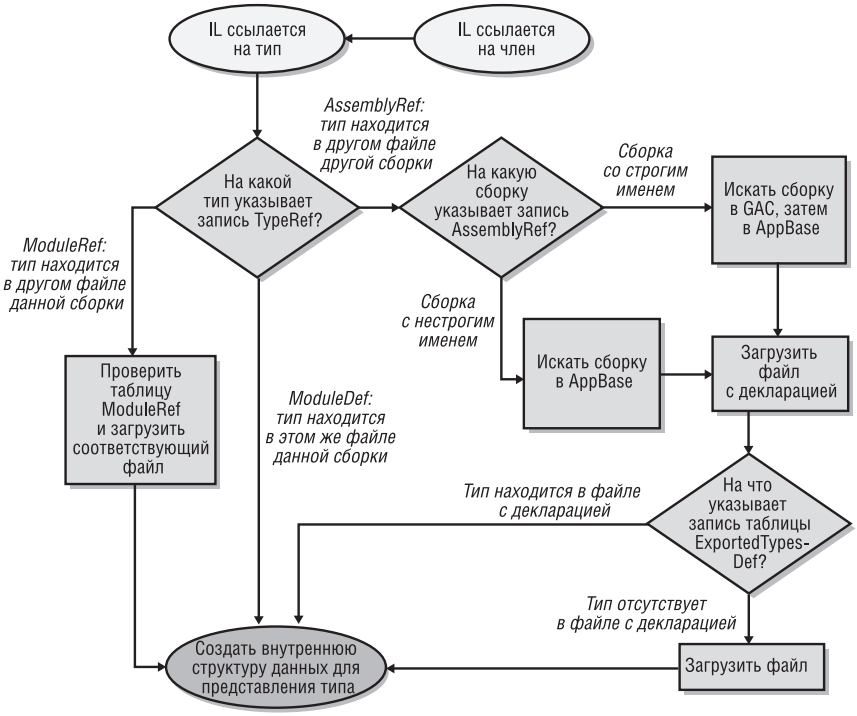


Рис. 3-6. *Блок-схема алгоритма поиска на основе метаданных, используемых CLR, файла сборки, где определен тип или метод, на который ссылается IL-код*

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

В предыдущем примере CLR обнаруживала, что тип *System.Console* реализован в файле другой сборки. CLR должна найти эту сборку и загрузить PE-файл, содержащий ее декларацию. После этого декларация сканируется в поисках сведений о PE-файле, в котором реализован искомый тип. Если нужный тип содержится в том же файле, что и декларация, все замечательно, а если в другом файле, то CLR загружает этот файл и посматривает 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*.

Рис. 3-6 иллюстрирует процесс привязки к типам.

Внимание! Строго говоря, приведенный пример не является верным на все сто. 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.

Так что в нашем 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 следующим проверяется каталог *C:\Windows\Assembly\GAC_MSIL*. Если и там сборку найти не удается, проверяется каталог *C:\Windows\Assembly\GAC* на предмет наличия сборок версии 1.x.

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

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 обсудим 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="FredTypes"
              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 bindingRedirect** 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 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 для сборок с нестрогими именами. При этом номер версии сборки игнорируется, и его следует опустить при определении элемента codeBase. URL, заданный элементом codeBase, также должен ссылаться на подкаталог базового каталога приложения.
- **Второй набор элементов dependentAssembly, assemblyIdentity и bindingRedirect** При попытке найти сборки FredTypes с номерами версии 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 publisherPolicy** Если организация, производитель сборки FredTypes, развернула файл политики издателя (описание этого файла см. в следующем разделе), 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; далее CLR ищет именно эту сборку/версию. О политике издателя я расскажу в следующем разделе. Наконец CLR просматривает сборку/версию в файле Machine.config и применяет все указанные в нем перенаправления на другие версии.

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 по URL-адресу, заданному этим элементом.

Эти конфигурационные файлы обеспечивают администратору настоящий контроль над решением, принимаемым CLR относительно загрузки той или иной сборки. Если в приложении оказывается ошибка, администратор может связаться с издателем сборки, содержащей ошибку, после чего издатель пришлет новую

сборку. Однако администратор может заставить CLR загрузить новую сборку, модифицировав конфигурационный XML-файл приложения. Для простоты издатель может создать XML-файл средствами административной утилиты Microsoft .NET Framework Configuration, поставляемой в составе .NET Framework SDK. Имейте в виду, что она не поставляется в составе свободно распространяемых компонентов .NET Framework. Чтобы воспользоваться утилитой, вызовите **Панель управления** (Control Panel), выберите **Администрирование** (Administrative Tools), а затем **Microsoft .NET Framework Configuration**.

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 присылал новую версию сборки администратору, который устанавливал ее 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 CLR, работающую у 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 я покажу, 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 файла JeffTypes.config, конфигурационного файла 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 AL.exe с такими 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 таков.

- ***/out*** приказывает AL.exe создать новый PE-файл с именем Policy.1.0.JeffTypes.dll, в котором нет ничего, кроме декларации. 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.
- ***/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.
- ***/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.

- ***/linkresource*** заставляет AL.exe считать конфигурационный XML-файл отдельным файлом сборки. При этом в результате компоновки получается сборка из двух файлов. Оба следует упаковать 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 сопровождаться установкой политики 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. Тогда администратору нужно, чтобы CLR игнорировала сборку с политикой издателя. Для этого он может отредактировать конфигурационный файл приложения, добавив в него элемент publisherPolicy:

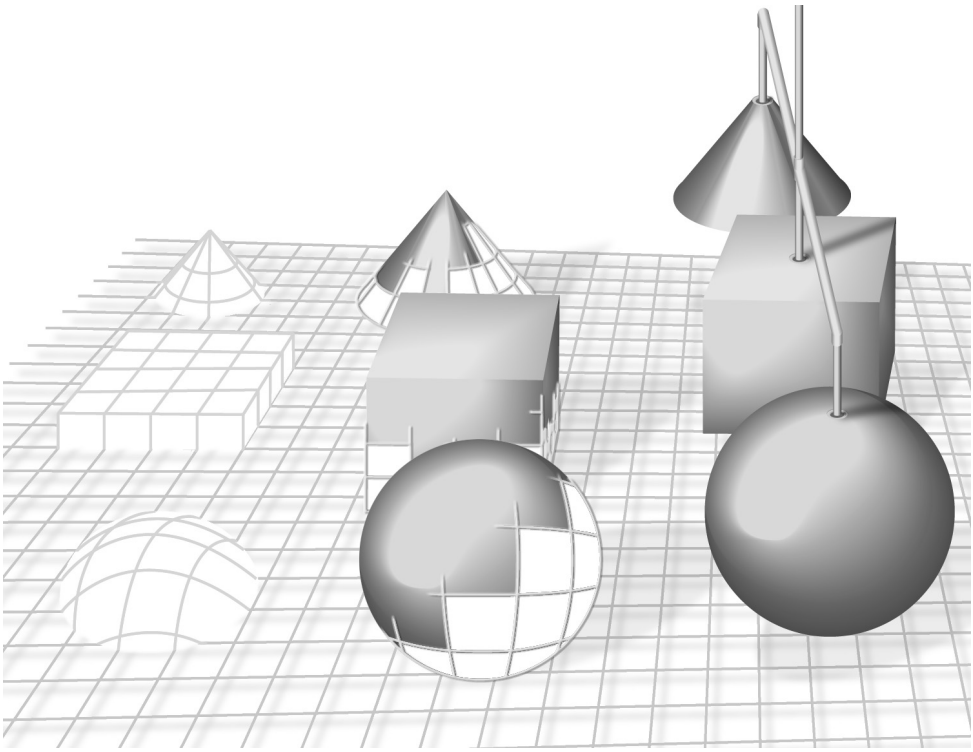
```
<publisherPolicy apply="no"/>
```

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. Обрабатывая 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 наличие 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 не совместима с прежней, 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. С другой стороны, 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

РАБОТАЕМ С ТИПАМИ



ГЛАВА 4

Основы типов

В этой главе мы познакомимся с основами работы с типами и общеязыковой исполняющей средой (Common Language Runtime, CLR). В частности, я представлю минимальную функциональность, присущую всем типам, расскажу о контроле типов, пространствах имен, сборках и различных способах приведения типов объектов. В конце главы я объясню, 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.

Табл. 4-1. Открытые методы *System.Object*

Открытый метод	Описание
Equals	Возвращает true, если два объекта имеют одинаковые значения. Подробнее об этом методе см. главу 5
GetHashCode	Возвращает хеш-код для значения данного объекта. Этот метод следует переопределить, если объекты типа используются в качестве ключа в хеш-таблице. Очень неудачно, что этот метод определен в *Object*, потому что большинство типов не служат ключами в хеш-таблице; этот метод уместнее было бы определить в интерфейсе. Подробнее об этом методе см. главу 5

Табл. 4-1. *(окончание)*

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, переопределение метода часто применяют при отладке: вызов такого метода позволяет получить строку, содержащую значения полей объекта. Считается, что *ToString* «знает» о *CultureInfo*, связанном с вызывающим потоком. Подробнее о *ToString* см. главу 11
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 22. Метод *GetType* невиртуальный, 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 (см. табл. 4-2).

Табл. 4-2. Защищенные методы *System.Object*

Защищенный метод	Описание
MemberwiseClone	Этот невиртуальный метод создает новый экземпляр типа и присваивает полям нового объекта соответствующие значения объекта *this*. Возвращается ссылка на созданный экземпляр
Finalize	Этот виртуальный 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 20

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

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

Оператор *new* делает следующее.

1. Вычисляет число байт, необходимых всем экземплярным полям типа и всем его базовым типам вплоть до и включая *System.Object* (в котором отсутствуют собственные экземплярные поля). Каждый объект кучи требует дополнительных членов, они называются *указатель на объект-тип* (type object pointer) и индекс блока синхронизации (*SyncBlockIndex*) и используются 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 байт в управляемой куче и обнуляя все эти байты.
3. Инициализирует указатель на объект-тип и *SyncBlockIndex*.
4. Вызывает конструктор экземпляра типа с параметрами, указанными при вызове new (в предыдущем примере это строка «ConstructorParam1»). Хотя мно-

гие компиляторы помещают в конструктор вызов конструктора базового типа. 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 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 CLR (см. главу 20), автоматически находя объекты, ставшие ненужными или недоступными и освобождая занимаемую ими память.

Приведение типов

Одна из важнейших особенностей CLR — *безопасность типов* (type safety). В период выполнения тип объекта всегда известен CLR. Точно определить тип объекта позволяет *GetType*. Поскольку это невиртуальный метод, никакой тип не сможет сообщить о себе ложные сведения. Так, тип *Employee* не может переопределить метод *GetType*, чтобы тот вернул тип *SpaceShuttle*.

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. Каждый язык программирования по-своему осуществляет приведение типов. 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 — такое преобразование может привести к ошибке. Вот пример приведения к базовому и производному типам:

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

public sealed class Program {
   public static void Main() {
      // Приведение типа не требуется, так как new возвращает объект Employee,
      // а Object – это базовый тип для Employee.
      Object o = new Employee();

      // Приведение типа обязательно, так как Employee – производный от Object.
      // В других языках (таких как Visual Basic) компилятор не потребует
      // явного приведения.
      Employee e = (Employee) o;
   }
}
```

Этот пример демонстрирует, что необходимо компилятору для компиляции кода. А что произойдет в период выполнения? CLR проверит операции приведения, чтобы преобразования типов осуществлялись либо к фактическому типу объекта, либо к одному из его базовых типов. Вот код, который успешно компилируется, но в период выполнения вызывает 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 генерирует исключение System.InvalidCastException.
      DateTime newYears = new DateTime(2007, 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 (при каждом приведении типа),
      // 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 объект типа, производного от *Employee*. Поскольку *Manager* — производный от *Employee*, CLR выполняет преобразование, и *PromoteEmployee* продолжает работу.

После того как *PromoteEmployee* возвращает управление, *Main* создает объект *DateTime*, который передает в *PromoteEmployee*. *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 *Employee*, ни другим типом, производным от *Employee*. В этот момент CLR не в состоянии выполнить приведение типов и генерирует исключение *SystemInvalidCastException*.

Если разрешить подобное преобразование, работа с типами станет небезопасной. При этом последствия могут быть непредсказуемы: увеличится вероятность краха приложения или возникнет брешь в защите, обусловленная возможностью типов выдавать себя за другие типы. Последнее обстоятельство подвергает большому риску устойчивую работу 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*. Я же использовал *Object*, только чтобы показать, как обрабатывают операции приведения типов компилятор C# и CLR.

Приведение типов в C# с помощью операторов *is* и *as*

В C# есть другие механизмы приведения типов. Так, 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 код:

```
Object o = new Object();
Boolean b1 = (o is Object);   // b1 равна true.
Boolean b2 = (o is Employee); // b2 равна 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 так:

```
if (o is Employee) {
   Employee e = (Employee) o;
   // Используем e внутри оператора 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 происходит анализ, является ли *o* ссылкой на *Employee*. Контроль типов в CLR укрепляет безопасность, но при этом приходится жертвовать производительностью, так как CLR должна выяснять фактический тип объекта, на который ссылается переменная (*o*), а затем проверить 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 оператора 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* несовмести-

мы, оператор *as* возвращает null. Заметьте: оператор *as* заставляет CLR верифицировать тип объекта только раз, а *if* лишь сравнивает *e* с null — такая проверка намного эффективнее, чем определение типа объекта.

Оператор *as* отличается от приведения типа по сути 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, как показано здесь:

```
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.
```

Чтобы убедиться, что вы усвоили материал, выполните упражнение. Допустим, существуют описания таких классов:

```
internal class B {     // Базовый класс.
}

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

Табл. 4-3 содержит в первом столбце код на C# — определите, каков будет результат обработки этих строк компилятором и CLR. Если код компилируется и выполняется без ошибок, поставьте отметку в графу OK, если вызывает ошибку компиляции — в графу CTE (compile-time error), а если приводит к ошибке в период выполнения — в графу RTE (run-time error).

Табл. 4-3. Тест на знание контроля типов

Оператор	OK	CTE	RTE
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		

Пространства имен и сборки

Пространства имен позволяют объединять родственные типы в логические группы, в них проще найти нужный разработчику 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. Так, в компиляторе C# предусмотрена директива *using*, а в Visual Basic — оператор *Imports*. Этот 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 — просто способ, позволяющий расширить имя типа и сделать его уникальным за счет добавления к началу имени групп символов, разделенных точками. Так, в нашем примере компилятор интерпретирует *FileStream* как *System.IO.FileStream*, а *StringBuilder* — как *System.Text.StringBuilder*.

Применять директиву *using* в C# и оператор *Imports* в Visual Basic не обязательно; можно набирать и 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 очень длинная строка с точками) и сборку, содержащую описание типа, чтобы в период 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 в 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 обнаружено, он попробует повторить поиск уже с приставкой *System.Text*. Благодаря двум директивам *using*, показанным выше, я смог ограничиться именами *FileStream* и *StringBuilder* — компилятор автоматически расширит ссылки до *System.IO.FileStream* и *System.Collections.StringBuilder*. Полагаю, вам понятно, что вводить и читать такой код намного проще.

Компилятору надо сообщить с помощью 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 найдена, сведения о ней и типе помещаются в метаданные результирующего управляемого модуля. 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 другие.

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 одинаковым названием, например *Widget*. В этом случае процесс формирования имен типов становится неуправляем, 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.
   }
}
```

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 таков:

```
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 создадут пространства имен *ABC*, в которые и включат тип *BuyProduct*. Тот, кто намерен разработать приложение, оперирующее обоими типами, не сдвинется с места, если в языке программирования не окажется способа различать программными средствами не только пространства имен, но и сборки. К счастью в компиляторе C# поддерживается функция *внешние псевдонимы* (extern aliases), позволяющая справиться с такой проблемой. Внешние псевдонимы также позволяют 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 пространство имен «Microsoft» для своих типов (к примеру, пространства имен *Microsoft.CSharp*, *Microsoft.VisualBasic* и *Microsoft.Win32*).

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; это настоящее имя типа с точки зрения CLR.

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

Как связаны пространства имен и сборки

Пространство имен и сборка (файл, содержащий реализацию типа) могут быть не связаны. В частности, различные типы, принадлежащие одному пространству имен, могут быть реализованы в нескольких сборках. Например, тип *System.IO.FileStream* реализован в сборке MSCorLib.dll, а *System.IO.FileSystemWatcher* — в System.dll. В действительности сборка System.dll даже не поставляется в составе .NET Framework.

В одной сборке могут содержаться типы из разных пространств имен. Так, в сборке MSCorLib.dll находятся типы *System.Int32* и *System.Text.StringBuilder*.

В документации .NET Framework SDK четко показано, к каким пространствам имен принадлежат те или иные типы и в каких сборках находятся реализации типов. На рис. 4-1 справа от раздела Syntax показано, что тип *ResXFileRef* относится к пространству имен *System.Resources*, однако его реализация находится в сборке System.Windows.Forms.dll. Чтобы скомпилировать код, ссылающийся на тип *ResXFileRef*, следует добавить в код директиву *using System.Resources;* а также использовать параметр */r:System.Windows.Forms.dll* компилятора.

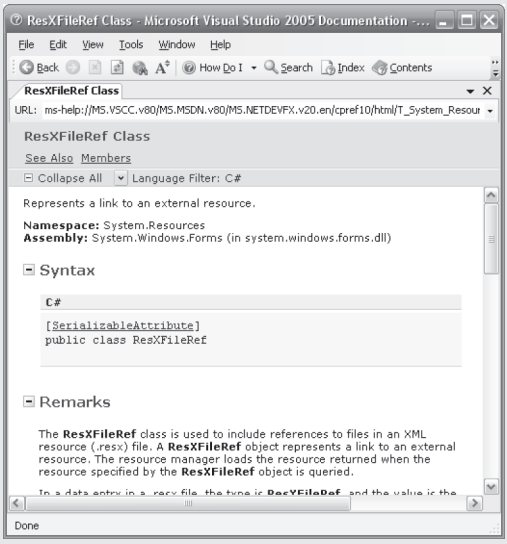


Рис. 4-1. *Сведения о пространстве имен и сборке для конкретного типа в документации к .NET Framework SDK*

Как разные компоненты взаимодействуют во время выполнения

В этом разделе я объясню, как во время выполнения взаимодействуют типы, объекты, стек потока и управляемая куча. Кроме того, я объясню, в чем различие между вызовом статических, экземплярных и виртуальных методов. А начнем 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 показан 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*.

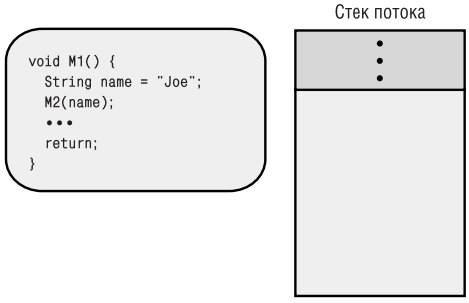


Рис. 4-2. *Стек потока перед вызовом метода M1*

Все методы, кроме самых простых, содержат некоторый *входной код* (prologue code), инициализирующий метод до начала его работы. Такие методы также содержат *выходной код* (epilogue code), выполняющий очистку после того, как метод выполнит свою основную работу, чтобы возвратить управление вызывающей программе. В начале выполнения метода *M1* его вводный код выделяет в стеке потока память для локальной переменной *name* (рис. 4-3).

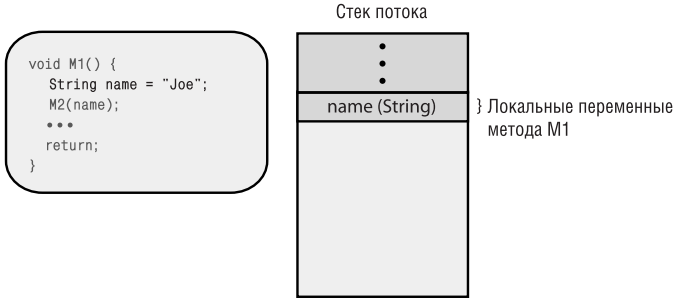


Рис. 4-3. *Размещение локальной переменной метода M1 в стеке потока*

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 будет храниться в переменной-параметре *s*. (Кстати, в некоторых процессорных архитектурах для повышения производительности аргументы передаются через регистры, но это различие для нашего обсуждения несущественно.) Также, при вызове метода адрес возврата в вызывающий метод заталкивается в стек (также показано на рис. 4-4).

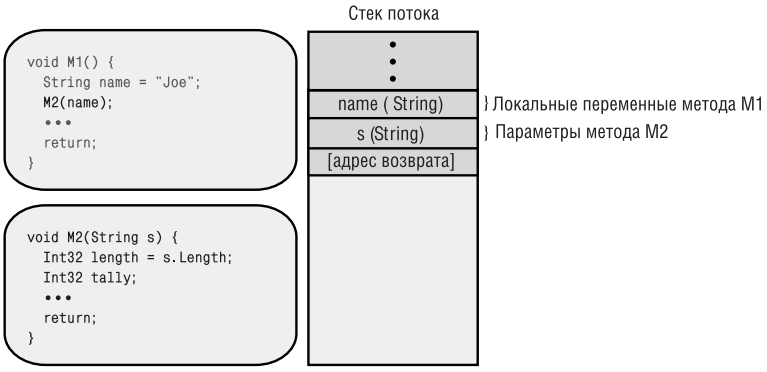


Рис. 4-4. *При вызове M2 метод M1 заталкивает аргументы и адрес возврата в стек потока*

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 оператора возврата, который записывает в указатель команд процессора адрес возврата из стека, и стековый фрейм *M2* возвращается до состояния, показанного 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*.

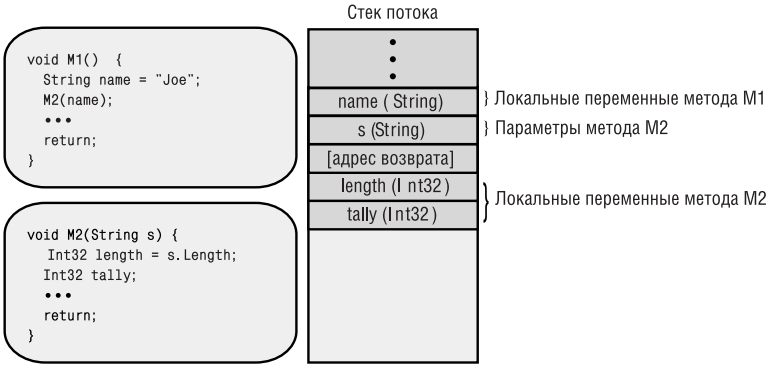


Рис. 4-5*. Выделение в стеке потока памяти для локальных переменных метода M2*

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 до состояния, показанного 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 теперь направим обсуждение в русло исполняющей среды CLR. Допустим, есть определения двух классов:

```
internal class Employee {
   public       Int32        GetYearsEmployed()  { ... }
   public virtual String     GenProgressReport() { ... }
   public static  Employee   Lookup(String name) { ... }
}

internal sealed class Manager : Employee {
   public override String GenProgressReport() { ... }
}
```

Процесс Windows запустился, в него загружена CLR, инициализирована управляемая куча, и создан поток (вместе с его 1 Мбайтом 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.

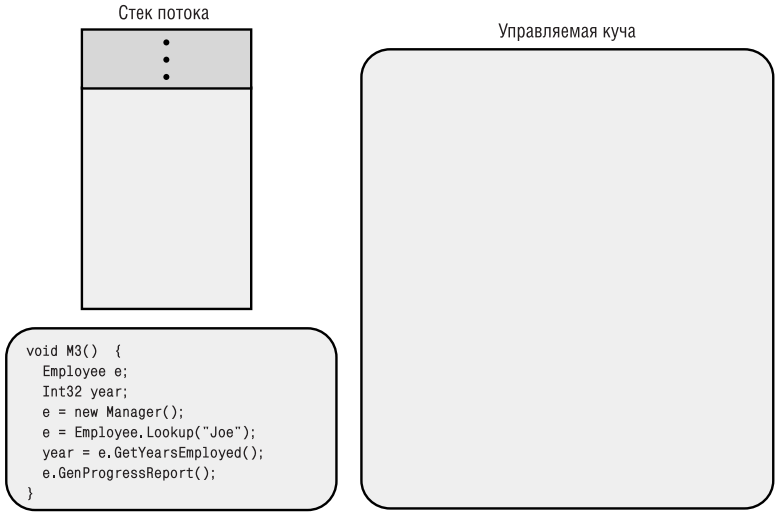


Рис. 4-6. *CLR загружена в процесс, его куча инициализирована, и готовится вызов стека потока, в который загружен метод M3*

В процессе преобразования IL-кода метода *M3* в машинные команды JIT-компилятор выявляет все типы, на которые есть ссылки в *M3* — *Employee*, *Int32*, *Manager* и *String* (из-за наличия строки «Joe»). На этом этапе CLR обеспечивает загрузку в домен AppDomain всех сборок, в которых определены все эти типы. Затем, используя метаданные сборки, CLR получает информацию о типах и создает структуры данных, собственно представляющие 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), и я не буду показывать их на рисунке.

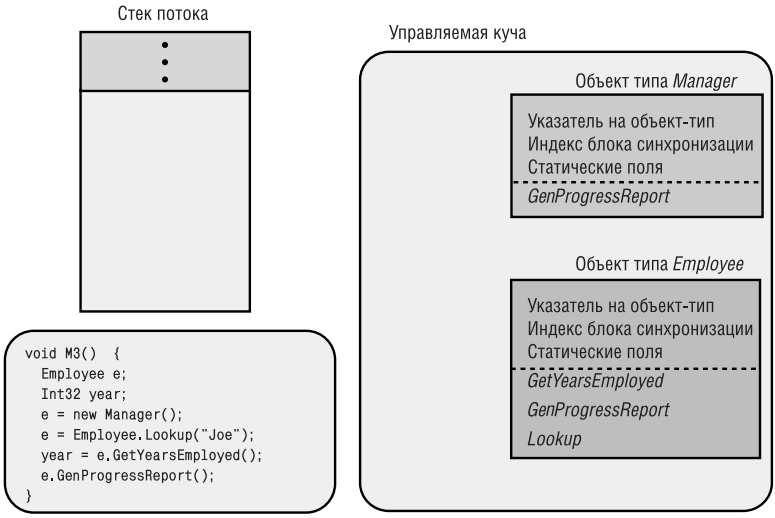


Рис. 4-7. *При вызове* M3 *создаются объекты типа Employee и Manager*

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 *GenProgressReport*), который и представлен в таблице методов этого типа.

После того как CLR позаботилась о создании всех необходимых для метода объектов-типов и компиляции кода метода *M3*, исполняющая среда приступает к выполнению машинного кода *M3*. При выполнении входного кода *M3* в стеке потока выделяется память для локальных переменных (см. рис. 4-8). Между прочим, CLR автоматически инициализирует все локальные переменные значением null или 0 (нулем) — это делается в рамках выполнения входного кода метода.

Далее, *M3* выполняет код создания объекта *Manager*. При этом в управляемой куче создается экземпляр типа *Manager*, объект *Manager* (рис. 4-9). У объекта *Manager* — также, как и всех остальных объектов — есть указатель на объект-тип и индекс блока синхронизации. У этого объекта также 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 это *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 это 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).

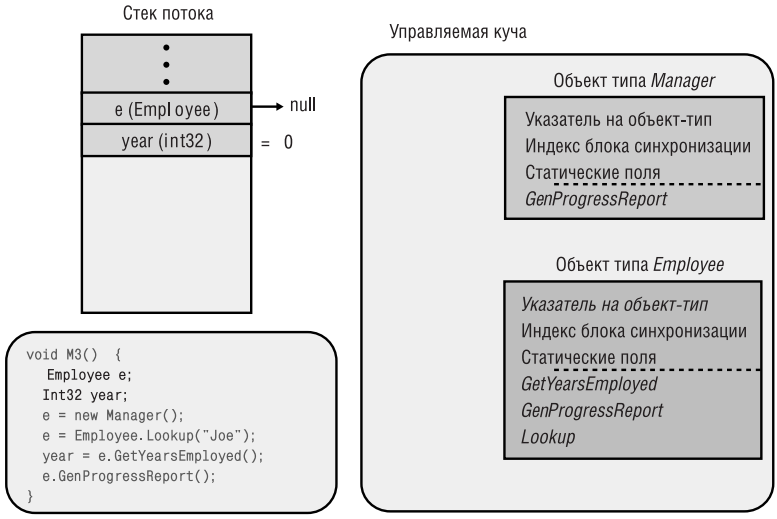


Рис. 4-8. *Выделение памяти в стеке потока для локальных переменных метода M3*

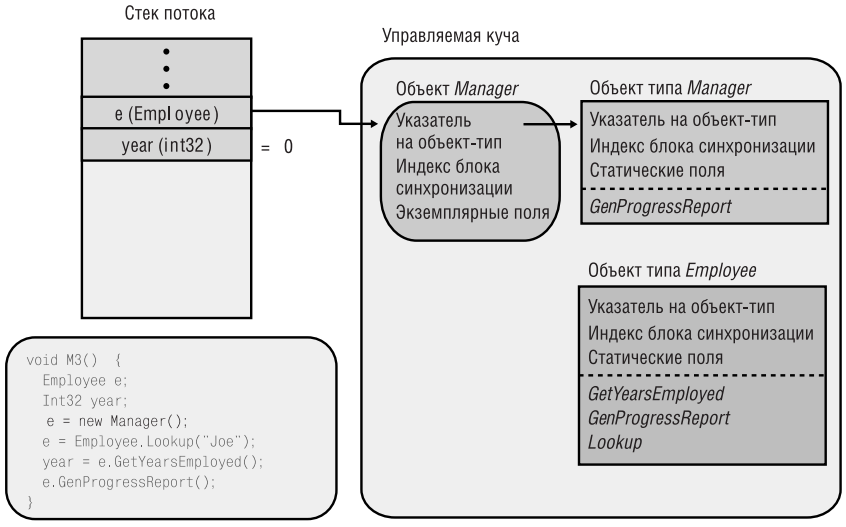


Рис. 4-9. *Создание и инициализация объекта Manager*

Следующая строка метода *M3* 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 предположения, что метод *Lookup* объекта *Employee* выполняет запрос базы 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 возвращает адрес готового объекта. Адрес размещается в локальной переменной *e*. Результат этой операции показан на рис. 4-10.

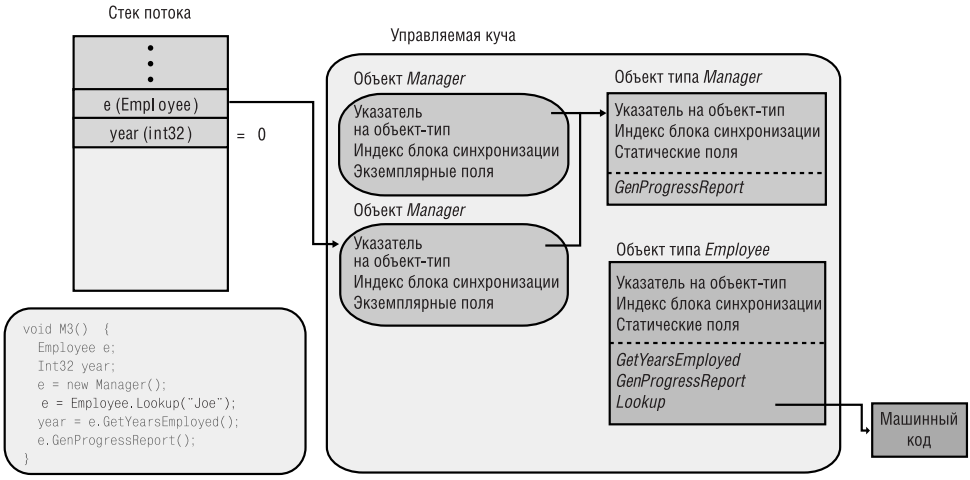


Рис. 4-10. *Статический метод Lookup в Employee выделяет память и инициализирует объект Manager для Joe*

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

Следующая строка метода *M3* вызывает невиртуальный экземпляр метода *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 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).

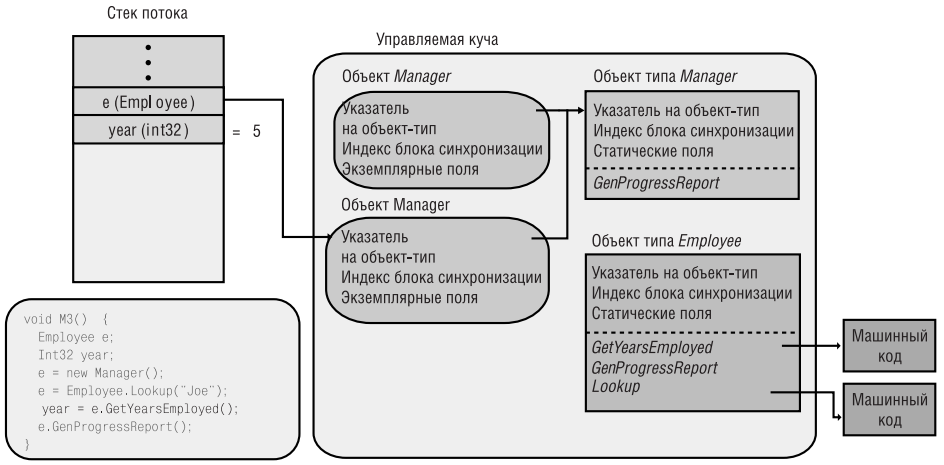


Рис. 4-11. *Невиртуальный экземплярный метод GetYearsEmployeed в Employee возвращает «5»*

Следующая строка метода *M3* вызывает виртуальный экземплярный *GenProgressReport* метод 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 *e* ссылается на объект *Manager*. Результат этой операции показан на рис. 4-12.

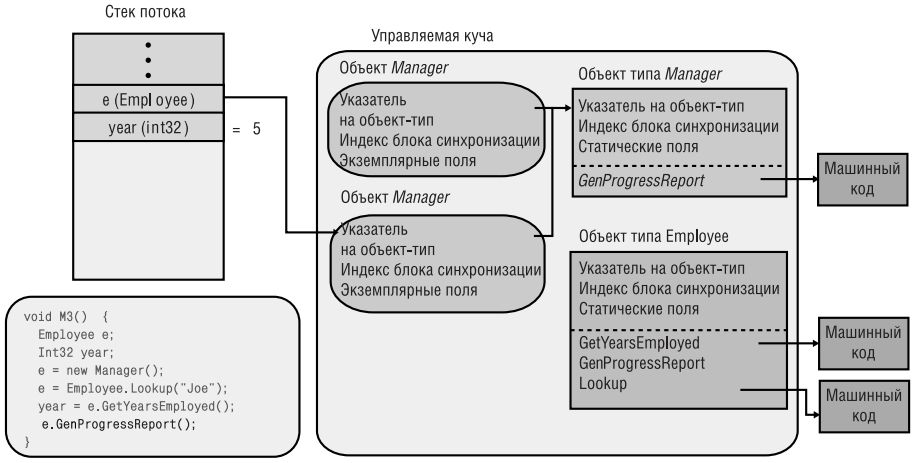


Рис. 4-12. *При вызове виртуального метода GenProgressReport экземпляра Employee будет вызвана переопределенная реализация этого метода в Manager*

Заметьте, если метод *Lookup* в *Employee* обнаружит, что *Joe* всего лишь *Employee*, а не *Manager*, то *Lookup* создаст объект *Employee*, член-указатель на объект-тип которого ссылается на объект типа *Employee*, это приведет к тому, что выполнится реализация *GenProgressReport* в *Employee*, а не в *Manager*.

Итак, мы обсудили взаимоотношения между исходным текстом, IL и машинным JIT-кодом, поговорили о стеке потока, аргументах и локальных переменных и о том, как эти аргументы и переменные ссылаются на объекты в управляемой куче. Также мы узнали, что объекты хранят указатель на свой объект-тип (содержащий статические поля и таблицу методов). Мы также обсудили, как CLR вызывает статические методы, невиртуальные и виртуальные экземплярные методы. Все рассказанное призвано дать вам более полную картину работы CLR и помочь при создании архитектуры, проектировании и реализации типов, компонентов и приложений. Заканчивая главу, я хотел бы сказать еще несколько слов о происходящем внутри CLR.

Наверняка вы обратите внимание, что объекты типа *Employee* и *Manager* содержат члены-указатели на объекты-типы. Причина в том, что объекты-типы — это, по сути, и есть сами объекты. Создавая объекты-типы, CLR должна инициализировать эти члены. Законно спросить: «Какие значения присвоить при инициализации?». В общем, при своем запуске в процессе CLR сразу же создает специальный объект-тип для типа *System.Type* (он определен в MSCorLib.dll). Объекты типа *Employee* и *Manager* являются «экземплярами» этого типа, и по этой причине их указатели на объекты-типы инициализируются так, чтобы ссылаться на объект-тип *System.Type* (рис. 4-13).

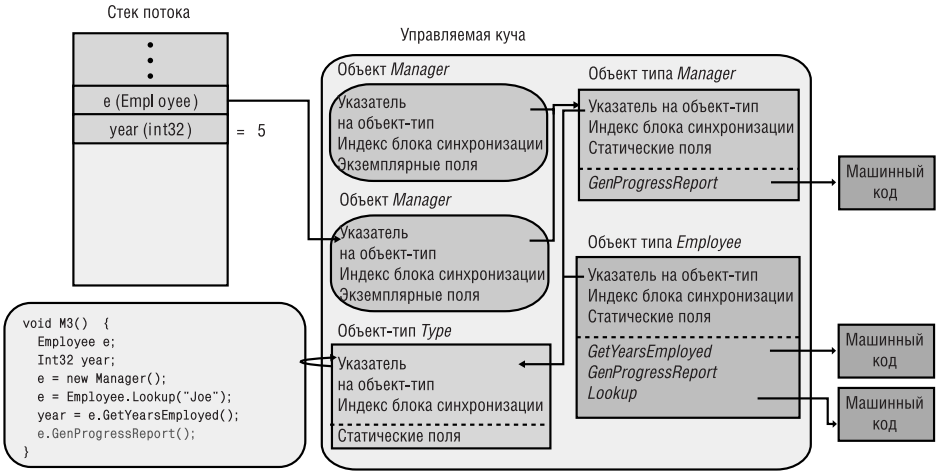


Рис. 4-13. *Объекты типа Manager и Employee как экземпляры типа System.Type*

Ясно, что объект-тип *System.Type* сам является объектом и поэтому также имеет член-указатель на объект-тип, и 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 указанного объекта. Иначе говоря, метод *GetType* возвращает указатель на объект-тип объекта, и именно поэтому возможно определить истинный тип любого объекта в системе (включая объекты-типы).

ГЛАВА 5

Элементарные, ссылочные и значимые типы

В этой главе речь идет о разновидностях типов, с которыми вы будете иметь дело при программировании для Microsoft .NET Framework. Важно, чтобы все разработчики четко осознавали разницу в их поведении. Приступая к изучению .NET Framework, я толком не понимал, в чем разница между элементарными, ссылочными и значимыми типами, и поэтому невольно напустил в свой код трудно вылавливаемых ошибок и снизил его эффективность. Надеюсь, мой опыт и объяснения разницы между этими типами помогут вам избавиться от лишней головной боли и повысить производительность работы.

Элементарные типы в языках программирования

Некоторые типы данных применяют так широко, что для работы с ними во многих компиляторах предусмотрен упрощенный синтаксис. Например, целую переменную можно создать так:

```
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 и более простые выражения, например:

```
int a = 0;
```

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

```
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 элементарные типы C#. В других языках типам, удовлетворяющим общеязыковой спецификации Common Language Specification (CLS), будут соответствовать аналогичные элементарные типы. Однако поддержка языком типов, не удовлетворяющих требованиям CLS, необязательна.

Табл. 5-1. Элементарные типы C# и соответствующие типы FCL

Элементарный тип C#	Тип FCL	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 это было в неуправляемом коде на C++)
float	*System.Single*	Да	32-разрядное *float* в стандарте IEEE
double	*System.Double*	Да	64-разрядное *float* 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
object	*System.Object*	Да	Базовый тип для всех типов
string	*System.String*	Да	Массив символов

Иначе говоря, можно полагать, 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». Я стараюсь использовать имена типов FCL и избегать имен элементарных типов. На самом деле мне бы хотелось, чтобы имен элементарных типов не было совсем, а разработчики употребляли только имена FCL-типов. И вот почему.

- Мне попадались разработчики, не понимавшие, что использовать в коде: *string* или *String*. В C# это не важно, так как ключевое слово *string* в точности преобразуется в FCL-тип *System.String*.
- В C# *long* отображается в *System.Int64*, но в другом языке это может быть *Int16* или *Int32*. Как известно, в C++ с управляемыми расширениями (C++/CLI) *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* и *ReadSingle* и т. д., а у типа *System.Convert* — методы *ToBoolean*, *ToInt32* и *ToSingle* и т. д. Вот вполне приемлемый код, в котором строка, содержащая *float*, кажется мне неестественной, и сразу не очевидно, что код корректный:

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

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.
```

Но, 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 элементарных типах и применяет свои правила при компиляции кода. Иначе говоря, он распознает наиболее распространенные шаблоны программирования и генерирует такие IL-команды, благодаря которым исходный код работает, как требуется. В первую очередь это относится к приведению типов, литералам и операторам, примеры которых мы рассмотрим ниже.

Начнем с того, что компилятор выполняет явное и неявное приведение между элементарными 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». Так, преобразование из *Int32* в *Byte* требует явного приведения к типу, так как при больших величинах *Int32* будет потеряна точность; требует приведения и преобразование из *Single* в *Int16*, поскольку число *Single* может оказаться больше, чем допустимо для *Int16*.

Разные компиляторы могут создавать различный код для выполнения приведения. 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 элементарных типов вы найдете в разделе спецификаций языка C#, посвященном преобразованиям («Conversions»).

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

```
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 элементарных типов

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

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

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

В каждом языке свои способы обработки переполнения. C и C++ не считают переполнение ошибкой и разрешают усекать значения; приложение не прервет

свою работу. А вот Visual Basic всегда рассматривает переполнение как ошибку и, обнаружив его, генерирует исключение.

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 делает неявного приведения типа, во вторую строку потребовалось ввести приведение к типу *Byte*.

В CLR есть IL-команды, позволяющие компилятору по-разному реагировать на переполнение. Так, суммирование двух чисел выполняет команда *add*, не реагирующая на переполнение, и 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 тогда разработчики должны быть уверены в отсутствии переполнения либо его возникновение должно быть специально предусмотрено.

Чтобы включить управление 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 переполнение. Когда оно возникает, CLR генерирует исключение *OverflowException*. Код приложения должен предусматривать корректную обработку этого исключения.

Однако программистам вряд ли подойдет включение или отключение режима проверки переполнения во всем коде. У них должна быть возможность самим решать, как реагировать на переполнение в каждом 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));   // Генерируется OverflowException.
```

Здесь *b* и 200 преобразуются в 32-разрядные числа и суммируются; результат равен 300. Затем преобразование 300 в *Byte* генерирует исключение *OverflowException*. Если приведение к *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 внутрь 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. Пример:

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

Используя *checked* и *unchecked*, учитывайте следующее.

- 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.

Внимание! Тип *System.Decimal* стоит особняком. В отличие от многих языков программирования (включая C# и Visual Basic) в CLR *Decimal* не относится к элементарным типам. В CLR нет IL-команд для работы со значениями *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 *Decimal* компилятор создает вызовы членов *Decimal*, которые и выполняют реальную работу. Поэтому значения *Decimal* обрабатываются медленнее элементарных типов CLR. Кроме того, раз нет IL-команд для манипуляции числами *Decimal*, то не будут иметь эффекта ни *checked* и *unchecked*, ни соответствующие параметры командной строки компилятора. И любая «небезопасная» операция над *Decimal* обязательно вызовет исключение *OverflowException*.

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 с 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 может инициировать сборку мусора.

Если бы все типы были ссылочными, эффективность приложения резко упала бы. Представьте, насколько замедлится выполнение приложения, если при каждом обращении к значению типа *Int32* будет выделяться 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 наборов (collections), требуемых приложению на протяжении его существования.

В документации по .NET Framework можно сразу увидеть, какие типы относят к ссылочным, а какие — к значимым. Если тип называют *классом* (class), речь идет о ссылочном типе. Так, классы *System.Object*, *System.Exception*, *System.IO.FileStream* и *System.Random* — это ссылочные типы. В свою очередь значимые типы в доку-

ментации 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 12.

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 не может служить базовым типом для какого-либо другого ссылочного или значимого типа. Поэтому, например, нельзя в описании нового типа указать в качестве базовых *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 поначалу 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 разместятся экземпляры данного типа, а разработчик, использующий тип в своем коде, управлять этим не может.

Ниже и на рис. 5-1 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. 5-1 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. 5-1 показан результат
   // выполнения ВСЕХ предыдущих строк.
}
```

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. Разница в поведении ссылочных и значимых типов значительна. Поэтому так важно представлять, к какому семейству относится тот или иной тип — к ссылочному или значимому: ведь это может существенно повлиять на поведение кода.

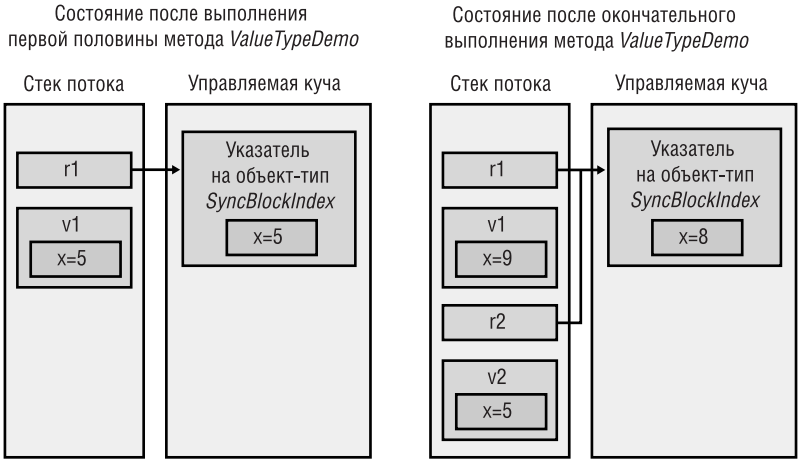


Рис. 5-1. *Различие между размещением в памяти значимых и ссылочных типов*

В предыдущем примере есть строка:

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

Может показаться, что экземпляр *SomeVal* будет помещен в управляемую кучу. Но, 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 на 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, посмотрите, не использовать ли вместо ссылочного типа значимый. Иногда это позволяет повысить эффективность кода. Особенно это справедливо для типа, удовлетворяющего *всем* перечисленным ниже условиям.

- Тип ведет себя подобно элементарному. В частности, это значит, что тип достаточно простой и у него нет членов, способных изменить экземплярные поля типа, в этом случае говорят, что тип *неизменяемый* (immutable).
- 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 одно и перечисленных ниже условий верно:

- размер экземпляров типа мал (примерно 16 байт или меньше);
- размер экземпляров типа велик (более 16 байт) и 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 с ссылочными типами, у значимых типов есть и недостатки. Вот некоторые особенности, отличающие значимые и ссылочные типы.

- Объекты значимого типа существуют в двух формах: *неупакованной* (unboxed) и *упакованной* (boxed) (см. следующий раздел). 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 хеш-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* чуть ниже.
- Поскольку в объявлении нового значимого или ссылочного типа нельзя указывать значимый тип в качестве базового класса, то 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 особого вида значимого типа, допускающего присваивание null, (nullable types). Этот тип обсуждается в главе 18.
- 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. С другой стороны, каждая переменная значимого типа имеет собственную копию данных «объекта», и операции с одной переменной значимого типа не повлияют на другую переменную.
- Так как неупакованные значимые типы не размещаются в куче, память, отведенная для них, 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 включить 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 разрешает включать в описание значимого типа метод *Finalize*, 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. Но при описании типа можно указать, сохранить ли порядок полей данного типа, определенный программистом, или разрешить CLR выполнить эту работу.

Чтобы сообщить CLR способ управления полями, укажите в описании класса или структуры атрибут *System.Runtime.InteropServices.StructLayoutAttribute*. Чтобы порядок полей устанавливался CLR, нужно передать конст-

руктору атрибута параметр *LayoutKindAuto,* чтобы сохранить установленный программистом порядок — *LayoutKind.Sequential*, а параметр *LayoutKind.Explicit* позволяет разместить в памяти, явно задав смещения. Если в описании типа не применен *StructLayoutAttribute*, порядок полей выберет компилятор.

Для ссылочных типов (классов) компилятор Microsoft C# выбирает *LayoutKindAuto*, 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 потребуется изменить поведение компилятора по умолчанию, например:

```
using System;
using System.Runtime.InteropServices;
// Для повышения производительности разрешим CLR
// установить порядок полей для этого типа.
[StructLayout(LayoutKind.Auto)]
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 использует для имитации того, что в неуправляемом C/C++ называлось *объединением* (union), потому что несколько полей могут начинаться с одного смещения в памяти. Вот пример:

```
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 и к ним нельзя обратиться через указатель. Однако часто нужно получить ссылку на экземпляр значимого типа. Скажем, вы хотите сохранить структуры *Point* в объекте типа *ArrayList* (определен в пространстве имен *System.Collections*). В коде это выглядит примерно так:

```
// 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 так:

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

Отсюда видно, что в качестве параметра *Add* нужен *Object*, то есть ссылка (или указатель) на объект в управляемой куче. Но 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.

Преобразование значимого типа в ссылочный позволяет выполнить *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, необходимыми для всех объектов в управляемой куче, — указателем на объект-тип и индексом *SyncBlockIndex*.
2. Поля значимого типа копируются в память, выделенную только что в куче.
3. Возвращается адрес объекта. Этот адрес является ссылкой на объект; значимый тип превратился в ссылочный.

Некоторые компиляторы, например компилятор C#, автоматически создают IL-код, необходимый для упаковки экземпляра значимого типа, 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 скопируются во вновь созданный в куче объект *Point*. Полученный адрес упакованного объекта *Point* (теперь это ссылочный тип) будет передан методу *Add*. Объект *Point* останется 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.

Многие языки, разработанные для CLR (например, C# и Visual Basic), автоматически создают код для упаковки значимых типов в ссылочные, когда это необходимо. Однако другие языки (вроде C++ с Managed Extensions) требуют, чтобы программист сам писал код упаковки значимых типов там, где это требуется.

Примечание Следует заметить, что в состав FCL входит новое множество обобщенных классов наборов, из-за чего устарели необобщенные классы наборов. Так, вместо *System.Collections.ArrayList* следует использовать класс *System.Collections.Generic.List<T>*. Обобщенные классы наборов во многих отношениях совершеннее своих необобщенных аналогов. В частности, API-интерфейс стал яснее и совершеннее, а также повышена производительность классов наборов. Но одно из самых ценных улучшений заключается в предоставляемой обобщенными классами наборов возможности работать с наборами значимых типов, не прибегая к их упаковке/распаковке. Одна эта особенность позволяет значительно повысить производительность, так как значительно 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 компилирования, а код становится понятнее за счет сокращения числа приведений типов. Все это мы обсудим в главе 16.

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). 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 (как в плане замедления, так и дополнительной памяти), поэтому нужно знать, когда компилятор сам создает код для выполнения этих операций, и стараться минимизировать создание такого кода.

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*. Но, 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.
}
```

Как я уже говорил, распаковка часто сопровождается копированием полей. Код на C# 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 C# 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.

---

[1] CLR также позволяет распаковывать значимые типы в версию этого же типа, поддерживающую присвоение null (см. главу 18).

Теперь такой 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 не могут не сказаться на производительности приложения.

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

Внимание! Если вас беспокоит производительность вашего приложения, надо 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, чтобы просмотреть IL-код готовых методов на предмет команд *box*.

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 voidMain() 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 сохраним 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 сохраним 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*. Но, 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 неупако-

ванный значимый тип *v*, но это не влияет на упакованное *Int32*, величина которого остается равной 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* — неупакованное значение, а *arg0* — это *Object*, поэтому *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 *Int32* внутри упакованного *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 *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 тем, что я убрал операцию приведения типа *(Int32)* для переменной *o*. Этот код выполняется быстрее, так как *o* уже является ссылочным типом *Object*, а его адрес можно сразу передать методу *Concat*. Удалив приведение 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 voidMain() 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 использование им 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 выше 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 применительно к наиболее часто используемым значимым типам.

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 определить 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 16).

И последнее, что касается упаковки: если вы знаете, что ваш код будет периодически заставлять компилятор упаковывать один какой-то значимый тип, можно уменьшить и ускорить свой код, выполнив упаковку этого типа вручную. Взгляните на пример:

```
using System;

public sealed class Program {
   public static void Main() {
      Int32 v = 5; // Создаем переменную неупакованного 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*, *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 экземпляр значимого типа. Простое правило: если нужна ссылка на экземпляр значимого типа, этот экземпляр должен быть упакован. Обычно это случается, 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 индекс *SyncBlockIndex*.

Поскольку неупакованные значимые типы не имеют *SyncBlockIndex*, то не может быть и нескольких потоков, синхронизирующих свой доступ к экземпляру через методы типа *System.Threading.Monitor* (или оператор *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 невиртуально, а *System.ValueType* переопределяет эти виртуальные методы 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 вызывать невиртуально виртуальные методы значимого типа.

Вместе с тем, вызов невиртуального 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 14.) 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)"

      // p 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, связанных с упаковкой/распаковкой.

- **Вызов *ToString*** При вызове ToString упаковка *p1* не требуется. Сначала я бы решил, что *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 невиртуального *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* будет передана 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 *Point* и, по этой причине, адрес памяти из *c* может передаваться в *CompareTo*, при этом никакой дополнительной упаковки не требуется.
- **Третий вызов *CompareTo*** При третьем вызове *CompareTo c* уже ссылается на упакованный *Point* в куче. Поскольку *c* сама по себе является интерфейсным типом *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 *c* к типу *Point*, объект в куче, на который указывает *c*, 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 размерных типах посредством интерфейсов

Посмотрим, насколько хорошо вы усвоили размерные типы, упаковку и распаковку. Взгляните: можете ли вы сказать, что будет выведено на консоль:

```
using System;

// Point - размерный 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 инициализирует его поля *m_x* и *m_y* равными 1. Затем *p* пакуется до первого обращения к *WriteLine*, который вызывает *ToString* для упакованного *Point*, и 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)».

Затем *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 временно-

го 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 размерном 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 - размерный 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 конец *Main* я добавил пару примеров.

В первом примере *p* — неупакованный объект типа *Point* — приводится к *IChangeBoxedPoint*. Такое приведение типа вызывает упаковку *p*. *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 *IChangeBoxedPoint*. Упаковка здесь не производится, поскольку *o* уже упакован. Затем вызывается *Change*, который изменяет поля *m_x* и *m_y* упакованного *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 размерном типе. В C# сделать это без интерфейсов нельзя.

Внимание! Ранее в этой главе я говорил, что в значимых типах нельзя определять члены, которые изменяют какие-либо поля экземпляра. 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 размерных типах. Должен сказать, что эти незначительные нюансы размерных типов стоили мне многодневной отладки; поэтому я и описываю их в этой книге. Надеюсь, вы не забудете об этих нюансах, и они не застигнут вас врасплох. Не бойтесь размерных типов — они полезны и занимают свою нишу. Просто не забывайте, что ссылочные и размерные 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.

Равенство и тождество объектов

Часто разработчикам приходится создавать код сравнения объектов. В частности, это требуется, когда объекты размещаются в наборы и требуется писать код для сортировки, поиска и сравнения отдельных элементов набора. В этом разделе рассказывается о равенстве и тождестве объектов, а также говорится 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. Но, если аргументы ссылаются на разные объекты, *Equals* сложнее определить, содержат ли объекты одинаковые значение, 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 выше стандартная реализация не годится: нужно переопределить *Equals* и написать его 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 разного типа, вернуть false. Не нужно быть семи пядей во лбу, чтобы понять, что объект *String* не равен объекту *FileStream*.
3. Сравнить все определенные в типе экземплярные поля объектов *obj* и *this*. Если хотя бы одна пара полей не равна, вернуть false.
4. Вызвать метод *Equals* базового класса, чтобы сравнить определенные в нем поля. Если метод *Equals* базового класса вернул false, тоже вернуть false, в противном случае — true.

Поэтому Microsoft должна была реализовать метод *Equals* типа *Object* так.

```
public class Object {
   public virtual Boolean Equals(Object obj) {
      // Сравниваемый объект не может быть равным null.
      if (obj == null) return false;
```

```
      // 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;
   }
}
```

Но, поскольку Microsoft реализовала метод *Equals* по-другому, правила 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 также означает, 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* с таким 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 может отличаться от понятия «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* переопределяется 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 п. 3 в методе *Equals* типа *ValueType* используется отражение (см. главу 22). Так как отражение в CLR работает медленно, при создании собственного значимого типа нужно переопределить *Equals* и создать собственную 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 соблюдались 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 *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 «==» and «!=»** — обычно код реализации этих методов операторов 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 *SystemIComparable<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, чтобы он вызывал безопасный в отношении типов метод *CompareTo*.

Хеш-коды объектов

Разработчики FCL решили, что при формировании хеш-таблиц полезно применять любые экземпляры любых типов. С этой целью в *System.Object* включен виртуальный метод *GetHashCode*, позволяющий получить для любого объекта целочисленный (*Int32*) хеш-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 Microsoft C# выдаст предупреждение. Так, при компиляции кода, представленного ниже, 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*, в том, что реализация типа *System.Collections.Hashtable* и *System.Collections.Generic.Dictionary* требует, чтобы два равных объекта имели одинаковые значения хеш-кодов. Поэтому, переопределяя *Equals*, нужно переопределить *GetHashCode* и гарантировать тем самым соответствие алгоритма, применяемого для вычисления равенства, алгоритму, используемому для вычисления хеш-кода объекта.

По сути, когда вы добавляете пару «ключ — значение» в объект *Hashtable* или *Dictionary*, первым вычисляется хеш-код для ключа. Этот хеш-код указывает, в каком «сегменте» будет храниться пара «ключ — значение». Когда объекту *Hashtable* (или *Dictionary*) нужно найти некий ключ, он вычисляет для него хеш-код. Хеш-код определяет «сегмент» поиска имеющегося в таблице ключа, равного заданному. Применение этого алгоритма хранения и поиска ключей означает, что, если вы измените хранящийся в *Hashtable/Dictionary* ключ, *Hashtable/Dictionary* не сможет найти этот объект. Если вы намерены изменить ключ в хеш-таблице, то сначала удалите имеющуюся пару «ключ — значение», модифицируйте ключ, а затем добавьте в хеш-таблицу новую пару «ключ — значение».

В описании метода *GetHashCode* нет особых хитростей. Но для некоторых типов данных и их распределения в памяти бывает непросто подобрать алгоритм хеширования, который выдавал бы хорошо распределенный диапазон значений. Вот простой алгоритм, хорошо 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 хеш-кодов для экземпляров своего типа, старайтесь следовать таким правилам:

- используйте алгоритм, который дает случайное распределение, повышающее производительность хеш-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 хеширования;
- 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 хеш-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 AppDomain; 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 хеш-кодом для другого объекта.

Примечание Если нужно получить уникальный (в рамках домена AppDomain) идентификатор объекта, не следует вызывать метод *GetHashCode* типа *Object*. Этот метод возвращает то же значение, которое возвратил бы метод *GetHashCode* типа *Object*; однако, если в типе метод *GetHashCode* типа *Object* переопределен, не удастся получить уникальный идентификатор объекта.

Тем не менее в FCL есть метод, который нужно вызывать для получения уникального идентификатора объекта. В пространстве имен *System.Runtime.CompilerServices* обратитесь к открытому статическому методу *GetHashCode* класса *RuntimeHelpers*, который принимает в качестве аргумента ссылку на *Object*. Этот метод возвращает уникальный идентификатор для объекта, даже если в типе объекта метод *GetHashCode* переопределен. Название этого метода обусловлено историческими причинами, но было бы лучше, если бы Microsoft назвала его иначе, например *GetUniqueObjectID*.

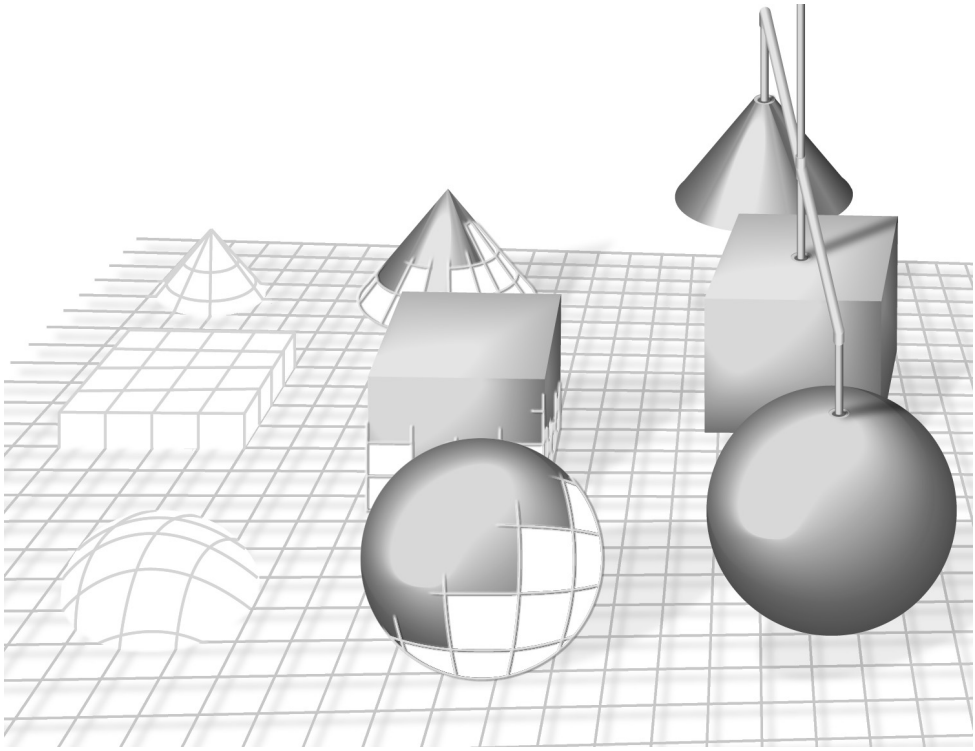
Реализация *GetHashCode* для *System.ValueType* использует механизм отражения (который отличается медленной работой) и обрабатывает некоторые поля экземпляра типа операцией исключающего «или» (XOR). Такой простой способ может быть полезен для некоторых размерных типов, но я бы посоветовал вам сделать свою реализацию *GetHashCode*, поскольку вы будете уверены в своем методе и ваша версия окажется быстрее базовой.

Внимание! Если вы взялись за реализацию собственной хеш-таблицы или пишете код, в котором будет вызываться *GetHashCode*, никогда не храните значения хеш-кодов. Они подвержены изменениям в силу своей природы. Так, при переходе к следующей версии типа может просто измениться алгоритм вычисления хеш-кода для объекта.

Я знаю компанию, которая проигнорировала это предупреждение. Посетители ее Web-сайта создавали новые учетные записи, выбирая имя пользователя и пароль. Строка (*String*) пароля передавалась методу *GetHashCode*, а полученный хеш-код сохранялся в базе данных. В дальнейшем при входе на Web-сайт посетители указывали свой пароль, который снова обрабатывался методом *GetHashCode,* и полученный хеш-код сравнивался с сохраненным в базе данных. При совпадении пользователю предоставлялся доступ. К несчастью, после обновления до новой версии CLR метод *GetHashCode* типа *String* изменился и стал возвращать другой хеш-код. Результат оказался плачевным — все пользователи потеряли доступ к Web-сайту!

ЧАСТЬ III

ПРОЕКТИРОВАНИЕ ТИПОВ



ГЛАВА 6

Основные сведения о членах и типах

В части 2 рассматривались типы и операции, применимые ко всем экземплярам любого типа. Я также пояснил, почему все типы делятся на две категории — ссылочные и значимые типы. В этой и последующих главах я покажу, как проектировать типы, используя различные виды членов. В главах с 7 по 10 различные члены будут рассмотрены подробнее.

Члены типа

В типе можно определить следующие члены.

- **Константа (глава 7)** — идентификатор, определяющий некую постоянную величину. Эти идентификаторы обычно используют для повышения читабельности кода и удобства сопровождения и поддержки. Константы всегда связаны с типом, а не экземпляром типа. В некотором смысле константы всегда статичны.
- **Поле (глава 7)** представляет неизменяемое или изменяемое значение. Поле может быть статическим — тогда оно является частью типа. Поле может быть и экземплярным (нестатическим) — тогда оно является частью объекта. Я настоятельно рекомендую делать поля закрытыми, чтобы внешний код не мог нарушить состояние типа или объекта.
- **Конструктор экземпляров (глава 8)** — метод, служащий для инициализации полей экземпляра при его создании.
- **Конструктор типа (глава 8)** — метод, используемый для инициализации статических полей типа.
- **Метод (глава 8)** 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.
- **Перегруженный оператор (глава 8)** определяет, что нужно проделать с объектом при применении к нему оператора. Перегрузка операторов отсутствует в общеязыковой спецификации CLS, поскольку не все языки программирования ее поддерживают.
- **Оператор преобразования (глава 8)** — это метод, определяющий порядок явного или неявного приведения/преобразования объекта из одного типа в дру-

гой. Операторы преобразования не входят в спецификацию CLS по той же причине, что и перегруженные операторы.

- **Свойство (глава 9)** представляет собой механизм, позволяющий применить простой (напоминающий обращение к полям) синтаксис 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-наборах.
- **Событие (глава 10)**. Статическое событие — это механизм, позволяющий типу посылать уведомление слушающему типу/объекту. Экземплярное (нестатическое) событие является механизмом, позволяющим объекту посылать уведомление слушающему типу/объекту. События обычно инициируются в ответ на изменение состояния типа или объекта, порождающего событие. Событие состоит из двух методов, позволяющих типам или объектам («слушателям») регистрировать и отменять регистрацию-подписку на событие. Помимо этих двух методов, в событиях обычно используется поле-делегат для управления набором зарегистрированных слушателей.
- **Тип** позволяет определять другие вложенные в него типы. Он применяется обычно для разбиения большого, сложного типа на небольшие блоки для упрощения его реализации.

Итак, цель данной главы 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 используемого языка программирования — именно поэтому CLR называют *общеязыковой* 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 транслирует такой тип и его члены в метаданные. Еще раз напомню, что каждый из членов в отдельности будет 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 свойство экземпляра.
   public Int32 SomeProp {                                          // 11
      get { return 0; }                                             // 12
      set { }                                                       // 13
   }

   // Параметризованное свойство экземпляра.
   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 ILDasm.exe (см. рис. 6-1).

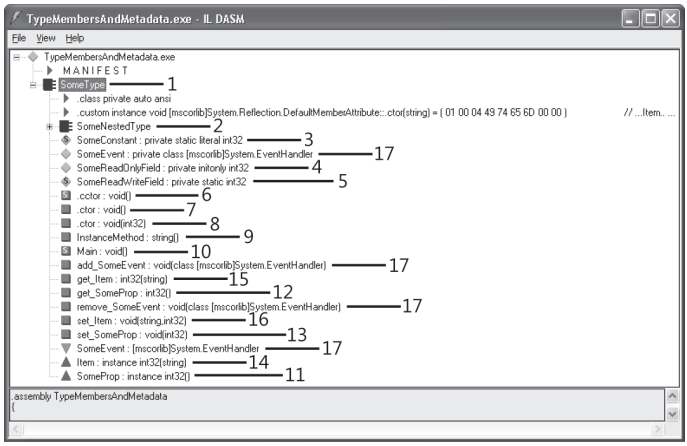


Рис. 6-1. *Выходные данные ILDasm.exe с метаданными, созданными на основе примера*

Заметьте, что компилятор генерирует метаданные для всех членов типа. На самом деле для некоторых членов (например, событие, член 17) компилятор создает дополнительные члены (поле и два метода) и метаданные. Сейчас не требуется полностью 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.

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. 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 бюджетирования или структуры подотчетности, эти группы не могут разместить все свои типы в единую сборку; 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 (public). Однако это означает, что эти типы будут доступны абсолютно всем сборкам; 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 (internal), 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.ComplierServices*. У атрибута есть строковый параметр, определяющий имя дружественной сборки и открытый ключ (переда-

ваемая 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.ComplierServices; // 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 к внутренним типам приведенной выше сборки из дружественной сборки просто. Например, дружественная сборка Wintellect с открытым ключом 12345678...90abcdef может обратиться к внутреннему типу *SomeInternalType* приведенной выше сборки так:

```
using System;

internal sealed class Foo {
   private static Object SomeMethod() {
      // 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 сборки (то есть не содержащей атрибута *InternalsVisibleTo*) 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 определить это имя, 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 с параметром C# */t:module* модуля (в противоположность сборке), 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 с совместимостью. Если сборки все же нужно поставлять в разное время, следует решить проблему созданием открытых классов, доступных из любой сборки, но ограничить доступ посредством *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 определен свой набор возможных модификаторов доступа, но в каждом языке программирования существует свой синтаксис и термины. Например, термин *Assembly* в CLR указывает, что член доступен изнутри сборки, тогда как в C# для этого используется *internal*.

В табл. 6-1 приведены шесть модификаторов доступа, определяющие уровень ограничения — от максимального (*Private*) до минимального (*Public*).

Табл. 6-1. Модификаторы доступа к члену

Термин CLR	Термин 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
Family or Assembly (Родовой или Сборочный)	*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 А определен внутренний тип, имеющий открытый метод, то код сборки Б не сможет вызвать открытый метод, поскольку внутренний тип сборки А не доступен из Б.

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, см. 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 типа был одинаковый модификатор доступа. То есть, если член базового класса является защищенным, то и член производного класса также 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 как открытый, но только не как закрытый. Это необходимо, чтобы пользователь производного типа всегда мог получить доступ к методу базового класса путем приведения к базовому типу. Если бы 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 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.

Вот 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 AStaticMethod() { }

   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 в ILDasm.exe библиотечной сборки (DLL), полученной при компиляции приведенного выше кода. Как видите, определение класса с ключевым словом *static* заставляет компилятор C# сделать этот класс абстрактным (*abstract*) и изолированным (*sealed*). Более того, компилятор не создает в классе метод конструктора экземпляра (*.ctor*).

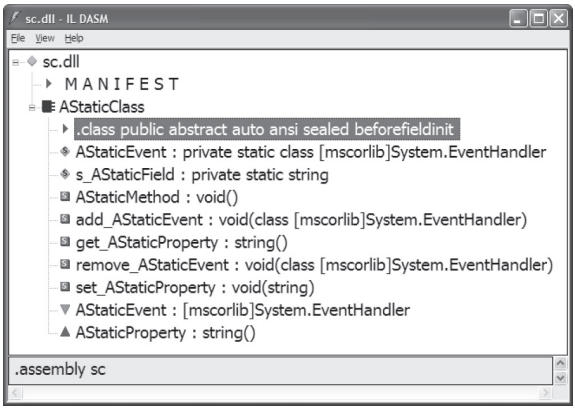


Рис. 6-2. *Судя по метаданным, отображаемым в ILDasm.exe, класс является абстрактным и изолированным*

Частичные классы, структуры и интерфейсы

Частичные классы, структуры и интерфейсы поддерживаются исключительно компиляторами C# и некоторых других языков, но 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. Существуют две основные причины, по которым исходный код разбивается на несколько файлов.

- **Управление версиями** Представьте, что определение типа содержит большое количество 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.
- **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 Visual Studio автоматически генерирует весь необходимый код и помещает его в отдельные файлы. Это значительно повышает продуктивность работы. Раньше автоматически генерируемый код помещался 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 Windows Form, Web Form, User Control и других, — Visual Studio создает два исходных файла: один предназначен для программиста, а в другой помещается код, создаваемый редактором форм. Теперь вероятность случайного изменения генерируемого кода существенно меньше.

Ключевое слово *partial* должно указываться во всех файлах, содержащих определение типа. При компиляции компилятор объединяет эти файлы, и готовый тип помещается в результирующий файл сборки с расширением .exe или .dll (или файл модуля .netmodule). Как уже говорилось, 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 пользуется неизменно высокой популярностью. В 70-80 годы объектно-ориентированные приложения были намного меньшего размера и весь код приложения разрабатывался в одной компании. Разумеется, в то время уже были операционные системы и приложения по максимуму использовали их возможности, но современные ОС предлагают намного больше функций.

Сложность программного обеспечения существенно возросла, и пользователи требуют от приложений богатых функциональных возможностей — графический интерфейс, меню, поддержку различных устройств ввода-вывода (мышь, прин-

тер, планшет), сетевые функции 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.

В компонентной разработке приложений (Component Software Programming) 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 C# для этого 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).
- 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 и поставляют в разное время. Управление версиями существенно повышает сложность компонентного программирования по сравнению с ООП, где все приложение пишет, тестирует и поставляет одна компания.

В .NET номер версии состоит из четырех частей: старший (major) и младший (minor) номера версии, номер компоновки (build) и номер редакции (revision). Например, у сборки с номером 1.2.3.4 старший номер версии — 1, младший номер — 2, компоновка — 3 и редакция — 4. Старший и младший номера обычно ослужат для идентификации сборки, а компоновка и редакция 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).

С другой стороны, если компания выпустит новую версию сборки, в которую внесены значительные изменения и 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 перенаправление связывания). Однако Microsoft планирует в будущем изменить загрузчик CLR, чтобы он мог загружать последнюю версию сборки. Например, если загрузчику потребуется версия 1.2.3.4, то, обнаружив версию 1.2.5.0, он загрузит именно ее. Что ж, с нетерпением будем ждать появления такой возможности.

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 повлиять 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.

Табл. 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 в качестве базового	Член нельзя переопределить в производном типе. Это ключевое слово применяется 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		

Назначение и использование этих ключевых слов будет показано 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 будет 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 определить и два метода с одним и тем же именем и параметрами, но с разным типом 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, при определении операторов преобразования язык C# смягчает это ограничение — см. главу 8.

Определим класс *Employee* с тремя различными видами методов.

```
internal class Employee {
   // Невиртуальный экземплярный метод.
   public       Int32     GetYearsEmployed() { ... }

   // Виртуальный метод (виртуальный, значит, экземплярный).
   public virtual  String GenProgressReport() { ... }

   // Статический метод.
   public staticEmployee  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 используется для невиртуального 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 невиртуальный метод экземпляра, тип переменной указывает, где определен необходимый метод. При использовании *callvirt* для вызова виртуального метода экземпляра CLR опре-

деляет 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 *call*. Проверка на *null* выполняется даже при вызове невиртуального метода экземпляра.

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 метода экземпляра.
      o.GetType();        // Вызов невиртуального метода экземпляра.
   }
}
```

После компиляции результирующий IL-код выглядит так.

```
.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 *callvirt*. Это выглядит странным, поскольку метод *GetType* невиртуальный. Тем не менее это работает, потому что во время JIT-компиляции CLR распознает, что *GetType* не виртуальный метод, и вызовет его невиртуально.

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 невиртуальных методов экземпляра 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 невиртуального метода *GetFive* среде CLR необходимо узнать только тип *p*, а это *Program*. При вызове *GetFive* аргумент *this* будет равен *null*, но в методе *GetFive* он не используется, поэтому исключения не будет. Однако компилятор C# вместо инструкции *call* вставляет *callvirt*, поэтому код вызовет исключение *NullReferenceException*.

Внимание! Если метод определен как невиртуальный, не рекомендуется в дальнейшем делать его виртуальным. Причина в том, что некоторые компиляторы для вызова невиртуального метода используют *call* вместо *callvirt*. Если метод сделать виртуальным и не перекомпилировать ссылающийся на него код, виртуальный метод будет вызван невиртуально и приложение может повести себя непредсказуемо. Если ссылающийся код написан на C#, проблем не будет, поскольку в C# все экземплярные методы вызываются с помощью *callvirt*. Но проблемы вполне 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 невиртуального 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 невиртуально. 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 невиртуального. Во-вторых, JIT-компилятор не может встраивать (inline) виртуальные методы, что также ударяет по производительности. В-третьих, виртуальные методы затрудняют управление версиями компонентов, как будет показано далее. В-четвертых, при определении базового типа часто создают набор перегруженных методов. Чтобы сделать их полиморфными, лучше всего сделать наиболее сложный метод виртуальным, оставив другие методы невиртуальными. Кстати, следование этому правилу поможет управлять версиями компонентов, не нарушая работу производных типов. Вот пример:

```
public class Set {
   private Int32 m_length = 0;

   // Этот перегруженный метод – невиртуальный.
   public Int32 Find(Object value) {
      return Find(value, 0, m_length);
   }

   // Этот перегруженный метод – невиртуальный.
   public Int32 Find(Object value, Int32 startIndex) {
      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 раз-

работчика нет доступа к исходным кодам компонентов (он может даже не знать, на каком языке они написаны), кроме того версии компонентов обновляют в разное время. Более того, из-за наличия полиморфизма и 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 использовать видимость типов и модификаторы доступа к членам.

Во-первых, при определении нового типа компиляторам следовало бы по умолчанию делать его изолированным. Вместо этого, большинство компиляторов (в том числе и 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 по умолчанию изменилось в следующих версиях компиляторов. 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 вызовов виртуальных методов в новых версиях, иначе в будущем возникнут проблемы с производными типами.
- **Производительность** Как уже говорилось, невиртуальные 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 невиртуальный вызов. Это возможно потому, что у изолированного класса не может быть производных классов.

  Например, в следующем коде JIT-компилятор может вызвать виртуальный метод *ToString* невиртуально.

```
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 ToString() {
      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 невиртуального 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.

Проблема 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. Но 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. Это позволит изменять базовый класс, не опасаясь за работоспособность производного класса. Идеальным будет, если разработчики компиляторов сделают *closed* модификатором доступа 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. Вот пример.

```
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, C# 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 говорилось, C# и другие компиляторы по умолчанию делают класс неизолированным. Если нет необходимости в предоставлении другим сборкам доступа к классу, его следует сделать внутренним. К счастью, именно так ведет себя по умолчанию компилятор C#. 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 C# поступает именно так. Вообще говоря, я бы предпочел, чтобы в C# 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 невиртуальными. К счастью, C# по умолчанию делает именно так. Разумеется, чтобы типом можно было воспользоваться, некоторые методы, свойства или 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, рекомендующее 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 говорилось, 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). Но 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.

Разработчиками компании А 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 Б спроектировали другой тип, *BetterPhone*, использующий *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 установлению соединения.
      }
   }
}
```

При попытке скомпилировать свой код разработчики компании Б 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 А 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 не имеет ничего общего 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 Б может использовать в своем приложении тип *BetterPhone*, например так:

```
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 исполнения свидетельствует о том, что код выполняет именно те действия, что нужны компании Б. 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 А. Допустим также, что все они сочли полезным установление соединения в самом методе *Dial*. Эти отзывы заставили разработчиков компании А 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.
        }
    }
}
```

Но теперь разработчики компании Б при компиляции своего типа *BetterPhone* (производного от новой версии *Phone*), получают 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» [предупреждение CS0114: 'BetterPhone.EstablishConnection()' скрывает унаследованный член 'Phone.EstablishConnection()'.

Чтобы текущий член переопределил реализацию, поставьте ключевое слово *override*, в противном случае добавьте ключевое слово *new*].

Компилятор предупреждает о том, 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 Б 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 Б 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* вызывает метод *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* из *Phone*, вызывающий *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* из типа *Phone* вызывает метод *EstablishConnection*, определенный в типе *Phone*, что и требовалось от программы.

Примечание Без ключевого слова *new* разработчики типа *BetterPhone* не смогут использовать в нем имена методов *Dial* и *EstablishConnection*. Если 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. Но, если изменение имени метода приведет лишь к ограниченному обновлению исходного текста, следует пойти на это, чтобы одинаковые имена методов *Dial* и *EstablishConnection*, обладающих разной семантикой в разных типах, не вводили в заблуждение других разработчиков.

Альтернативный вариант таков: компания Б, получив от компании А новую версию типа *Phone*, решает, что текущая семантика методов *Dial* и *EstablishConnection* типа *Phone* — это именно то, что нужно. В этом случае в компании Б полностью удаляют метод *Dial* из типа *BetterPhone*. Кроме того, поскольку теперь разработчикам компании Б нужно указать компилятору, что метод *EstablishConnection* из типа *BetterPhone* связан с одноименным методом из типа *Phone*, нужно удалить из его определения ключевое слово *new*. Но простого удаления ключевого слова здесь недостаточно, так как компилятор не поймет предназначения метода *EstablishConnection* типа *BetterPhone*. Чтобы выразить свои намерения явно, разработчик из компании Б должен, помимо прочего, изменить модификатор определенного в типе *BetterPhone* метода *EstablishConnection* с *virtual* на *override*. Вот 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 изменен
      // 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 из типа *BetterPhone*, так как он переопределяет виртуальный метод *EstablishConnection*, определяемый типом *Phone*.

ГЛАВА 7

Константы и поля

В этой главе я покажу, как добавить к типу члены, являющиеся данными. В частности, мы рассмотрим константы и поля.

Константы

Константа — 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*.

Другой важный момент: поскольку значение констант никогда не меняется, константы всегда считаются частью типа. Иначе говоря, константы считают 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 сборку DLL:

```
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 скомпонуем сборку приложения из такого 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 с типом *Int32* прямо в IL-код приложения (рис. 7-1). В сущности, после компоновки кода приложения сборка DLL даже не будет загружаться в период выполнения, поэтому ее можно просто удалить с диска.

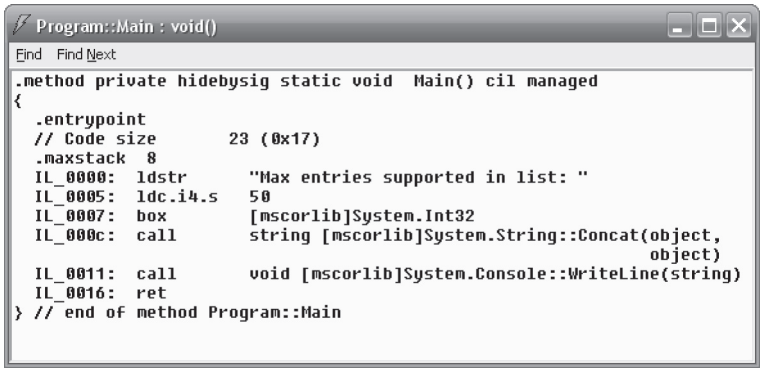


Рис. 7-1. *Представленный в окне ILDasm.exe IL-код метода с литералом константы, внедренным непосредственно в код метода*

Думаю, теперь проблема с управлением версиями при использовании констант должна стать очевидной. Если разработчик изменит значение константы *MaxEntriesInList* на 1000 и заново опубликует сборку DLL, это не повлияет на код самого приложения. Чтобы в приложении использовалось новое значение константы, его придется перекомпилировать. Нельзя применять константы, если модуль должен задействовать значение, определенное в другом модуле, во время выполнения (а не во время компиляции). В этом случае вместо констант следует использовать неизменяемые поля.

Поля

Поле (field) — это член данных, который хранит экземпляр размерного типа или ссылку на ссылочный тип. В табл. 7-1 приведены модификаторы, применяемые по отношению к полям.

Как видно из таблицы, общеязыковая 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 AppDomain (см. главу 21), что обычно происходит при JIT-компиляции любого метода, ссылающегося на этот тип. Динамическая память для хранения экземплярных полей выделяется при создании экземпляра данного типа.

Табл. 7-1. Модификаторы полей

Термин CLR	Термин C#	Описание
Static	*static*	Поле является частью состояния типа, а не объекта
Instance	*(default)*	Поле связано с экземпляром типа, а не самим типом
InitOnly	*readonly*	Запись в поле разрешается только из кода метода конструктора
Volatile	*volatile*	Код, обращающийся к полю, не обязательно специально должен оптимизироваться в отношении управления типами компилятором, CLR или оборудованием. Только следующие типы могут объявляться как *volatile*: все ссылочные типы, *Single*, *Boolean*, *Byte*, *SByte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Char*, а также все перечислимые типы, основанные на следующих типах: *Byte*, *SByte*, *Int16*, *UInt16*, *Int32* или *UInt32*

Поскольку поля хранятся в динамической памяти, их значения можно получить лишь в период выполнения. Поля также решают проблему с управлением 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 изменяемые (read/write) и неизменяемые (readonly) поля. Большинство полей — изменяемые. Это значит, что во время исполнения кода значение таких полей может многократно меняться. Однако данные в неизменяемые поля можно записывать только при исполнении метода-конструктора (который вызывается лишь раз — при создании объекта). Компилятор и механизм верификации гарантируют, что ни один метод, кроме конструктора, не сможет записать данные в неизменяемое поле. Заметим, что для изменения неизменяемого поля можно задействовать отражение.

Попробуем решить проблему с управлением версиями в примере из раздела «Константы», используя статические неизменяемые поля. Вот новая версия кода сборки DLL:

```
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 загрузит сборку DLL (так как она требуется во время выполнения) и извлекает значение поля *MaxEntriesInList* из динамической памяти, выделенной для его хранения. Естественно, это значение будет 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 — он просто работает в том виде, в каком был (хотя и чуть медленнее). Но 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 во время 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 ниже 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 объявлении (inline). C# позволяет использовать этот удобный синтаксис для инициализации констант, а также изменяемых и неизменяемых полей. Как я покажу в главе 8, C# считает, что инициализация поля при объявлении — это краткий синтаксис, позволяющий инициализировать поле во время исполнения конструктора. Вместе с тем, в C# возможны проблемы с производительностью, которые нужно учитывать при инициализация поля с использованием синтаксиса встраивания, а не присвоения в конструкторе. Они также обсуждаются в главе 8.

Внимание! Неизменность поля ссылочного типа означает неизменность ссылки, которую он содержит, но только не объекта, на которую эта ссылка указывает. Вот пример:

```
public sealed class AType {
   // InvalidChars должно всегда ссылаться 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 как то, на что ссылается InvalidChars, изменить нельзя.
      AType.InvalidChars = new Char[] { 'X', 'Y', 'Z' };
   }
}
```

ГЛАВА 8

Методы: конструкторы, операторы, преобразования и параметры

В этой главе мы обсудим разновидности методов, которые могут определяться в типе, и разберем ряд вопросов, касающихся методов. В частности, я покажу, как определяют методы-конструкторы (создающие экземпляры типов и сами типы), методы перегрузки операторов и методы операторов преобразования (выполняющие явное и неявное приведение типов). Кроме того, я расскажу, как передать методу параметры ссылками, а также 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 экземпляра не наследуются. Иначе говоря, в классе есть экземплярные конструкторы, которые определены в самом классе. Невозможность наследования означает, что к конструктору экземпляра нельзя применить следующие 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.

К примеру, такое определение класса:

```
public class SomeType {
}
```

идентично определению:

```
public class SomeType {
   public SomeType() : base() { }
}
```

Для абстрактных классов компилятор создаст конструктор по умолчанию с модификатором *protected*, в противном случае область действия будет *public*. Если в базовом классе нет конструктора без параметров, производный класс должен явно вызвать конструктор базового класса, иначе компилятор вернет ошибку. Для статических классов (*sealed* и *abstract*) 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 экземпляра 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 и поэтому конструктору просто нечего делать.

В редких ситуациях экземпляр типа может создаваться без вызова конструктора экземпляра. 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.

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, выполнится 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 код (рис. 8-1).

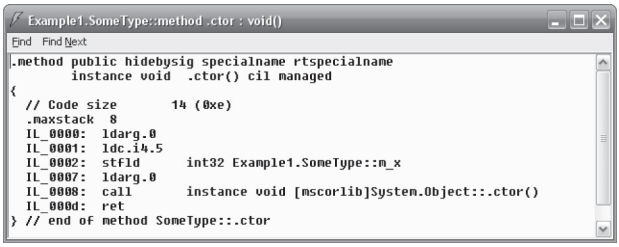


***Рис. 8-1**. IL-код метода-конструктора объекта SomeType*

Как видите, конструктор объекта *SomeType* содержит код, записывающий в поле *m_x* значение 5 и вызывающий конструктор базового класса. Иначе говоря, компилятор C# допускает 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. Вот пример. Представьте себе такой класс:

```
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; }
}
```

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 выше классе определены три конструктора, компилятор трижды генерирует код, инициализирующий поля *m_x*, *m_s* и *m_d*: по 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. Вот пример использования способности C# явно заставлять один конструктор вызывать другой конструктор за счет использования 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;

   // Этот конструктор содержит код, инициализирующий 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 значение 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 конструкторы ссылочных типов (*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 такой код:

```
internal struct Point {
   public Int32 m_x, m_y;
}
internal sealed class Rectangle {
   public Point m_topLeft, m_bottomRight;
}
```

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 внутри объекта ссылочного типа. Но, как сказано выше, поля значимого типа инициализируются нулевыми или пустыми значениями.

CLR действительно позволяет программистам определять конструкторы для значимых типов, но эти конструкторы будут выполнены 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 будет исполнен, только если вызвать его явно. Так что, если конструктор объекта *Rectangle* не инициализировал его поля *m_topLeft* и *m_bottomRight* вызовом конструктора *Point* оператором *new*, 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. Но 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 скажите: 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++) будут ожидать, что компилятор C# поместит в конструктор *Rectangle* код, автоматически вызывающий конструктор структуры *Point* по умолчанию, не имеющий параметров, для двух полей *Rectangle*. Но, чтобы увеличить 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 если у него есть конструктор без параметров. Чтобы принудительно исполнить конструктор значимого типа, не имеющий параметров, разработчик должен добавить код для явного вызова конструктора значимого типа.

С учетом сказанного можно ожидать, что поля *m_x* и *m_y* обеих структур *Point* из объекта *Rectangle* в показанном выше коде будут инициализированы нулевыми значениями, так как в этой программе нет явного вызова конструктора *Point*.

Однако, как я сказал, мой первый вопрос был с подвохом. Подвох в том, что C# не позволяет определять для значимого типа конструкторы без параметров. Поэтому показанный выше код на самом деле даже не компилируется. При попытке скомпилировать его компилятор C# генерирует сообщение об ошибке: «error CS0568: Structs cannot contain explicit parameterless constructors» («ошибка CS0568: структура не может содержать явные конструкторы без параметров»).

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 является вложенным в объект ссылочного типа. Однако где гарантия, что поля значимых типов, работающих со стеком, будут инициализированы значениями 0 или null! 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 в них хотя бы записываются некоторые значения. Поэтому при верификации во время выполнения исключение сгенерировано не будет. Но 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 скрытые особен-

ности работы системы, описанные выше, 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: «error CS0573: 'SomeValType.m_x': cannot have instance field initializers in structs»(ошибка CS0573: 'SomeValType.m_x': нельзя создавать конструкторы экземплярных полей в структурах).

```
internal struct SomeValType {
   // В значимом типе нельзя встраивать 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: «error CS0171: Field 'SomeValType.y' must be fully assigned before control leaves the constructor» (ошибка CS0171: поле 'SomeValType.y' должно быть полностью определено до возвращения управления конструктором). Чтобы разрешить эту проблему, в поле *m_y* надо занести значение (обычно это 0) в конструкторе.

Конструкторы типов

Помимо конструкторов экземпляров, CLR поддерживает конструкторы типов (также известные как *статические конструкторы*, *конструкторы классов* или *инициализаторы типов*). Конструкторы типа можно применять и к интерфейсам (хотя C# этого не допускает), ссылочным и значимым типам. Подобно тому, как кон-структоры экземпляров используются для установки первоначального состояния экземпляра типа, конструкторы типов применяются для установки первоначального состояния типа. По умолчанию у типа не определен ни один конструктор. У типа может быть один и только один конструктор. Кроме того, у конструкторов типа 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, не имеющие параметров.
   static SomeValType() {
      // Исполняется при первом обращении к значимому типу SomeValType.
   }
}
```

Заметьте: конструкторы типа определяют так же, как конструкторы экземпляров, не имеющие параметров, за исключением того, что их помечают как статические. Кроме того, конструкторы типа всегда должны быть закрытыми (C# делает их закрытыми автоматически). Но, если явно пометить в исходном тексте программы конструктор типа как закрытый (или как-то иначе), компилятор C# выведет сообщение об ошибке: «error CS0515: 'SomeValType.SomeValType()': access modifiers are not allowed on static constructors» («ошибка CS0515: 'SomeValType.SomeValType()': в статических конструкторах нельзя использовать модификаторы уровня доступа»). Конструкторы типа всегда должны быть закрытыми, чтобы код разработчика не смог их вызвать, — напротив, 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. Вот пример:

```
internal struct SomeValType {
   static SomeValType() {
      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);  // Отображаем 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 AppDomain. 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 типа».)

Далее, после JIT-компиляции метода, начинается выполнение потока, и в конечном итоге очередь доходит до выполнения 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 AppDomain. Для этого при вызове конструктора типа вызывающий поток получает исключающую блокировку синхронизации потоков. Поэтому, если несколько потоков одновременно попытаются вызывать конструктор типа, только один получит блокировку, а 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 выполнят 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 еще одно его выполнение.

Примечание Поскольку CLR гарантирует, что конструктор типа выполняется только однажды в каждом домене AppDomain, а также обеспечивает его безопасность по отношению к потокам, конструктор типа лучше всего подходит для инициализации всех Singleton-объектов, необходимых для существования типа.

В рамках одного потока возможна неприятная ситуация, когда существуют два конструктора типов, 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 позаботится, чтобы код конструкторов типа выполнился лишь однажды, но исполняющая среда не в состоянии обеспечить, чтобы конструктор типа *ClassA* завершился 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 возникает исключение *System.TypeInitializationException*.

Код конструктора типа может обращаться только к статическим полям типа, обычно это делается, чтобы инициализировать их. Как и в случае экземплярных полей, C# предлагает простой синтаксис, позволяющий инициализировать статические поля типа:

```
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 изменить приведенный выше тип с *class* на *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 исходно написан так:

```
internal sealed class SomeType {
   private static Int32 s_x;
   static SomeType() { s_x = 5; }
}
```

При помощи ILDasm.exe нетрудно проверить, какой код на самом деле сгенерировал компилятор. Для этого нужно изучить IL-код конструктора типа (рис. 8-2). В таблице определений методов, составляющей метаданные модуля, метод-конструктор типа всегда называется .cctor («конструктор класса»).

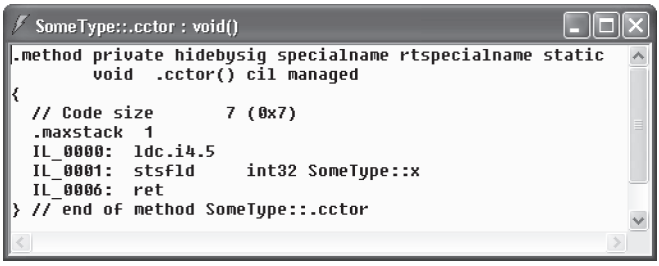


Рис. 8-2. *IL-код метода-конструктора типа SomeType*

Из показанного 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.

Конструктор типа не должен вызывать конструктор базового класса. Этот вызов необязателен, так как ни одно статическое поле типа не используется совместно с базовым типом и не наследуется от него.

Примечание Ряд языков, таких как Java, ожидает, 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*, поддерживаемый 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 код:

```
internal sealed class SomeType {
   private static Int32 s_x = 5;
```

```
    static SomeType() {
        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 выгрузки типа? Во-первых, следует знать, что типы выгружаются только при закрытии домена AppDomain. Когда AppDomain 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 поддерживает статические методы *Finalize*. Однако не все потеряно: если при закрытии AppDomain нужно исполнить некоторый код, можно зарегистрировать метод обратного вызова для события *DomainUnload* типа *SystemAppDomain*.

Производительность конструкторов типа

В предыдущем разделе я говорил о сложностях вызова конструкторов типов и рассказал, как JIT-компилятор принимает решение, нужно ли создавать IL-код конструктора, а CLR обеспечивает безопасный с точки зрения потоков вызов конструктора. Как оказалось, это всего лишь начало этих сложностей — есть еще нюансы, связанные с производительностью.

При компиляции метода JIT-компилятор самостоятельно решает, нужно ли создавать вызов на исполнение 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 нужно.
- JIT-компилятор может создать вызов в месте, предшествующем коду, обращающемуся к ненаследуемому статическому полю. Это называют семантикой вызова *до инициации поля* (before-field-init), потому что в этом случае CLR обеспечивает выполнение статического конструктора в какое-то время до обращения с статическому полю; 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 AppDomain 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 в метаданных 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 выполнился 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, из-за чего он исполняется 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 такой 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 используется 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, который может выполнять определенную видимую работу, поэтому его нужно выполнять в заданное разработчиком время.

Как видно из выходных данных программы, решение значительно сказывается на производительности работы кода. В PerfTest1 цикл выполняется за 2,2 секунды, что сильно отличается от последней строки — целых 7,62 секунды, то есть в три раза быстрее. Времена выполнения PerfTest2 намного «кучнее», так как JIT-ком-

пилятор «знал», что конструкторы типов уже вызывались, поэтому изъял 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 она даже не знает, что такое оператор. Смысл знаков операторов и код, который должен быть сгенерирован, когда этот знак встречается в исходном тексте, определяется языком программирования.

Так, если в C#-программе поставить между обычными числами знак «+», компилятор сгенерирует код, выполняющий сложение двух чисел. Когда знак «+» применяют к строкам, компилятор C# генерирует код, выполняющий конкатенацию этих строк. Для обозначения неравенства в C# используется знак «!=», а в Visual Basic — «<>». Наконец, знак «^» в C# означает операцию «исключающее или» (XOR), тогда как в Visual Basic это возведение в степень.

Хотя CLR находится в неведении относительно операторов, она 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 будет задействован 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 методы оператора были открытыми и статическими. Дополнительно C# (и многие другие языки) требует, чтобы у оператора-метода тип по крайней мере одного из параметров или возвращаемого значения совпадал с типом, в котором определен оператор-метод. Причина этого ограничения в том, что это позволяет компилятору C# в разумное время находить кандидатуры операторов-методов для привязки.

Вот 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 компилятор 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 приводится набор унарных и бинарных операторов, которые C# позволяет перегружать, их знаки и рекомендованные имена соответствующих методов, которые должен генерировать компилятор. Третий столбец я прокомментирую в следующем разделе.

Табл. 8-1. Унарные операторы C# и CLS-совместимые имена соответствующих методов

Знак оператора 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*
Нет	*op_True*	*IsTrue { get; }*
Нет	*op_False*	*IsFalse { get; }*

Табл. 8-2. Бинарные операторы C# и CLS-совместимые имена соответствующих методов

Знак оператора 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*
!=	*op_Inequality*	*Compare*
<	*op_LessThan*	*Compare*
>	*op_GreaterThan*	*Compare*
<=	*op_LessThanOrEqual*	*Compare*
>=	*op_GreaterThanOrEqual*	*Compare*

В спецификации CLR определены многие дополнительные операторы, поддающиеся перегрузке, но C# их не поддерживает. Они не очень распространены, поэтому я их здесь не указал. Полный список см. в спецификации ECMA (www.ecma-international.org/publications/standards/Ecma-335.htm) общеязыковой 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 выполнить 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 полезный инструмент, позволяющий 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 этот 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 используете язык, не поддерживающий перегрузку оператора «+» путем определения в типе, наверняка этот тип может предоставлять метод *op_Addition*. Логично ожидать, что в C# можно вызвать этот метод *op_Addition*, используя оператор «+», но это не так. Обнаружив оператор «+», компилятор C# ищет метод *op_Addition* с флагом метаданных *specialname*, который информирует компилятор, что *op_Addition* — перегруженный метод-оператор. Поскольку метод *op_Addition* создан языком, не поддерживающим 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 знак «+» в вызов этого метода.

Мое особое мнение 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, под-

держивающими методы перегруженных операторов. Если бы языки, поддерживающие перегрузку операторов, поддерживали бы и синтаксис со знаками 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* также должен 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 выше 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 методов операторов 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 встраивать код в метод с дружественным именем. Встраивание кода позволит JIT-компилятору оптимизировать весь код путем удаления дополнительного вызова метода и тем самым повысить скорость при выполнении.

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 преобразовать 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 создаст 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 типа). 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, тип *Rational* должен определять 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, использующий 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.

Выше 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 оператор-метод. Причина этого ограничения в том, что это позволяет компилятору C# в разумное время находить кандидатуры операторов-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 операторов преобразования следует указать, должен ли компилятор генерировать код для неявного вызова метода оператора преобразования 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, лишь когда в исходном тексте имеется явное приведение типов.

После ключевого слова *implicit* или *explicit* следует поместить указание (ключевое слово *operator*) компилятору, что данный метод представляет оператор преобразования. После ключевого слова *operator* надо указать целевой тип, в который преобразуется объект, а в скобках — исходный тип объекта.

Определив в показанном выше 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 теряется в результате преобразования (например, при преобразовании объекта типа *Rational* в *Int32*), следует определять оператор явного преобразования. На случай сбоя явного преобразования следует предусмотреть в операторе-методе генерацию 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 возвращаемых значений. CLR в полном объеме поддерживает предоставление типам возможности определить несколько методов, отличающихся только типом 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 тип определяет 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 есть подходящий метод оператора неявного преобразования, компилятор вставляет в результирующий IL-код вызов этого метода. Обнаружив 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.

Чтобы по-настоящему разобраться в методах перегруженных операторов и операторов преобразования, я настоятельно рекомендую использовать тип *System.Decimal* как наглядное пособие. В типе *Decimal* определены несколько конструкторов, позволяющих преобразовывать в *Decimal* объекты различных типов. Он также поддерживает несколько методов *To*Xxx для преобразования объектов типа *Decimal* в объекты других типов. Наконец, этот тип определяет ряд методов операторов преобразования и перегруженных операторов.

Передача методу параметров по ссылке

По умолчанию CLR предполагает, что все параметры методов передаются по значению. При передаче объекта ссылочного типа методу передается (по значению) ссылка (или указатель) на этот объект. Это означает, что метод может изменить переданный объект и вызывающий код получит измененный объект. Если параметром является экземпляр значимого типа, методу передается его копия. Это означает, что метод получает собственную копию объекта значимого типа, не доступную никому, кроме него, а исходный экземпляр объекта остается неизменным.

Внимание! Следует знать тип (ссылочный или значимый) каждого объекта, передаваемого методу в качестве параметра, поскольку код, манипулирующий параметрами, может существенно отличаться в зависимости от их типа.

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, которые описывают параметр как переданный по ссылке. Компилятор использует эти метаданные для генерации кода, передающего вместо самого параметра его адрес.

С точки зрения CLR *out* и *ref* не различаются, то есть независимо от ключевого слова генерируются одинаковые метаданные и IL-код. Однако компилятор C# различает эти ключевые слова, и разница касается выбора метода, используемого при инициализации объекта, на который указывает переданная ссылка. Если параметр метода помечен ключевым словом *out*, вызывающий код может не инициализировать его, пока не вызван этот метод. В этом случае вызванный метод не может читать значение параметра и должен записать его, прежде чем вернуть управление. Если параметр метода помечен ключевым словом *ref*, вызывающий код должен инициализировать его перед вызовом этого метода, а вызванный метод может как читать, так и записывать значение параметра.

Использование ключевых слов *out* и *ref* со значимыми и ссылочными типами существенно различается. Сначала рассмотрим их использование со значимыми типами:

```
public sealed class Program {
   public static void Main() {
```

```
      Int32 x;               // Инициализация x.
      GetVal(out x);         // Инициализировать x не обязательно.
      Console.WriteLine(x);  // Отображаем 10.
   }

   private static void GetVal(out Int32 v) {
      v = 10;  // Этот метод должен инициализировать переменную V.
   }
}
```

В этом коде переменная *x* объявлена в стеке *Main*, ее адрес передается методу *GetVal*. Параметр этого метода *v* представляет собой указатель на значимый тип *Int32*. Внутри метода *GetVal* значение *Int32*, на которое указывает *v*, изменяется на 10. Когда *GetVal* возвращает управление, *x* равно 10, это же значение выводится на консоль. Использование *out* со значимыми типами эффективно, так как оно предотвращает копирование экземпляров значимого типа при вызовах методов.

А теперь взгляните на пример, в котором вместо *out* использовано ключевое слово *ref*:

```
public sealed class Program {
   public static void Main() {
      Int32 x = 5;           // Инициализация x.
      AddVal(ref x);         // Параметр x должен быть инициализирован.
      Console.WriteLine(x);  // Выводит на консоль "15".
   }

   private static void AddVal(ref Int32 v) {
      v += 10;  // Этот метод может использовать инициализированный параметр v.
   }
}
```

Здесь переменная *x* объявлена в стеке *Main* и инициализирована значением 5. Далее адрес *x* передается методу *AddVal*, чей параметр *v* является указателем на значимый тип *Int32* в стеке *Main*. Внутри метода *AddVal* должно быть уже инициализированное значение *Int32*, на которое указывает параметр *v*. Таким образом, *AddVal* может использовать первоначальное значение *v* в любом выражении. *AddVal* также может изменить это значение, тогда вызывающему коду «вернется» уже новое значение. В этом примере *AddVal* прибавляет к исходному значению 10. Когда *AddVal* возвращает управление, переменная *x* метода Main содержит значение «15», которое затем 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 отсутствие ошибок в коде. Следующий код пытается передать методу, ожидающему параметр *ref*, неинициализированное значение, чем вызывает ошибку компиляции «error CS0165: Use of unassigned local variable 'x'» («ошибка CS0165: Использование локальной переменной x, у которой не задано значение.»):

```
public sealed class Program {
   public static void Main() {
      Int32 x;               // x не инициализируется.
```

```
      // Компиляция следующей строки заканчивается неудачей и 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 вызове метода в программах на C# надо указывать ключевое слово *out* или *ref*, ведь компилятор «знает», требует ли вызываемый метод указать *out* или *ref*, значит, он должен быть способен корректно скомпилировать код. Оказывается, компилятор действительно может все сделать автоматически (причем правильно). Однако создатели C# сочли, что вызывающий код должен явно указывать свои намерения, чтобы в том месте программы, где вызывается метод, сразу было ясно, что вызываемый метод должен изменить значение передаваемой переменной.

Кроме того, CLR позволяет по-разному перегружать методы в зависимости от того, какие параметры в них используются — *out* или *ref*. Так, 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 в результате JIT-компиляции подобных методов генерируется идентичный код метаданных, представляющих сигнатуру методов. Поэтому в показанном выше типе *Point* я не могу определить метод:

```
static void Add(out Point p) { ... }
```

Если попытаться включить последний метод *Add* в тип *Point*, компилятор C# вернет ошибку «error CS0663: 'Add' cannot define overloaded methods that differ only on ref and out» («ошибка CS0663: в 'Add' нельзя определить перегруженных методов, отличных от ref и out»).

Использование ключевых слов *out* и *ref* со значимыми типами дает тот же результат, что и передача ссылочного типа по значению. Ключевые слова *out* и *ref* позволяют методу управлять единственным экземпляром значимого типа. Вызывающий код должен выделить память для этого экземпляра, а вызванный метод будет управлять 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;  // Инициализация fs.

       // Открыть первый файл для обработки.
       StartProcessingFiles(out fs);

       // Продолжать, пока остаются файлы для обработки.
       for (; fs != null; ContinueProcessingFiles(ref fs)) {

          // Обработать файл.
          fs.Read(...);
       }
   }

   private static void StartProcessingFiles(out FileStream fs) {
      fs = new FileStream(...); // fs должна инициализироваться в этом методе.
   }

   private static void ContinueProcessingFiles(ref FileStream fs) {
      fs.Close();  // Закрыть последний обрабатываемый файл.

      // 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 новый объект. Заметьте также, что метод *ContinueProcessingFiles* может управлять передаваемым ему объектом, прежде чем вернет новый объект. Это возможно, так как его параметр помечен ключевым словом *ref*. Показанный выше код можно немного упростить:

```
class App {
   static public void Main() {
   FileStream fs = null;   // Инициализируется пустым значением (обязательно).

   // Открыть первый файл для обработки.
   ProcessFiles(ref fs);

   // Продолжать, пока остаются файлы для обработки.
   for (; fs != null; ProcessFiles(ref fs)) {

      // Обрабатываем файл.
      fs.Read(...);
   }
}
```

```
   void ProcessingFiles(ref FileStream fs) {
      // Закрыть предыдущий файл, если он был открыт.
      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 (...);
   }
}
```

А вот другой пример, демонстрирующий использование ключевого слова *ref* для реализации метода, обменивающего пару объектов ссылочного типа:

```
using System;
using System.IO;

public sealed class Program {
   public static void Main() {
      FileStream fs = null;   // Инициализация значением null (обязательная операция).

      // Открыть первый файл для обработки.
      ProcessFiles(ref fs);

      // Продолжать, пока остаются файлы для обработки.
      for (; fs != null; ProcessFiles(ref fs)) {

         // Обрабатываем файл.
         fs.Read(...);
      }
   }

   private static void ProcessFiles(ref FileStream fs) {
      // Закрыть предыдущий файл, если он был открыт.
      if (fs != null) fs.Close();  // Закрытие последнего обрабатываемого файла.

      // 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, обменивающего два ссылочных типа:

```
public static void Swap(ref Object a, ref Object b) {
   Object t = b;
   b = a;
   a = t;
}
```

Возможно, вы бы написали такой код, чтобы обменять ссылки на два объекта типа *String*:

```
public static void SomeMethod() {
   String s1 = "Jeffrey";
   String s2 = "Richter";

   Swap(ref s1, ref s2);
   Console.WriteLine(s1);  // Отображает на экране "Richter".
   Console.WriteLine(s2);  // Отображает на экране "Jeffrey".
}
```

Но компилироваться этот код не будет: переменные, передаваемые методу по ссылке, должны быть одного типа. Иначе говоря, метод *Swap* ожидает две ссылки на тип *Object*, а не *String*. Чтобы обменять значения двух ссылок типа *String*, нужно сделать так:

```
public static void SomeMethod() {
   String s1 = "Jeffrey";
   String s2 = "Richter";

   // Типы переменных, передаваемых по ссылке,
   // должны соответствовать ожидаемым методом.
   Object o1 = s1, o2 = s2;
   Swap(ref o1, ref o2);

   // Теперь преобразуем объекты обратно в строки.
   s1 = (String) o1;
   s2 = (String) o2;

   Console.WriteLine(s1);  // Отображает на экране "Richter".
   Console.WriteLine(s2);  // Отображает на экране "Jeffrey".
}
```

Эта версия метода *SomeMethod* действительно компилируется и работает как надо. Причина необходимости соответствия типов передаваемых и ожидаемых параметров — обеспечение безопасности типов. Следующий код (который, к счастью, не компилируется) является примером нарушения безопасности типов:

```
internal sealed class SomeType {
   public Int32 m_val;
}

public sealed class Program {
   public static void Main() {
      SomeType st;

      // Следующая строка генерирует сообщение: error CS1503: Argument '1':
      // cannot convert from 'ref SomeType' to 'ref object' (ошибка CS1503:
      // Нельзя преобразовать аргумент "1"
      // из 'ref SomeType' в '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*, *GetAnObject* может инициализировать параметр *o* любым объектом любого типа. В этом примере параметр *st* при возврате управления методом *GetAnObject* методу *Main* ссылается на объект типа *String*, который никак не является объектом типа *SomeType*, поэтому вызов метода *Console.WriteLine* непременно закончится неудачей. К счастью, компилятор C# откажется компилировать этот код, так как *st* представляет собой ссылку на объект типа *SomeType*, тогда как *GetAnObject* требует ссылку на *Object*.

Заставить эти методы работать, как ожидалось, можно с использованием обобщений. Вот так исправить показанный ранее метод *Swap*:

```
public static void Swap<T>(ref T a, ref T b) {
   T t = b;
   b = a;
   a = t;
}
```

После исправления *Swap* следующий код (идентичный показанному выше) будет без проблем компилироваться и выполняться:

```
public static void SomeMethod() {
   String s1 = "Jeffrey";
   String s2 = "Richter";

   Swap(ref s1, ref s2);
   Console.WriteLine(s1);  // Отображает на экране "Richter".
   Console.WriteLine(s2);  // Отображает на экране "Jeffrey".
}
```

За другими примерами использования обобщений для решения этой задачи отсылаю вас к классу *System.Threading.Interlocked* с его обобщенными методами *CompareExchange* и *Exchange*.

Передача методу переменного числа параметров

Иногда разработчику удобно определить метод, способный принимать переменное число параметров. Например, тип *System.String* предлагает методы, позволяющие выполнить конкатенацию произвольного числа строк, а также есть методы, при вызове которых можно задать набор строк, которые должны форматироваться все вместе.

Метод, принимающий переменное число аргументов, объявляют так:

```
static Int32 Add(params Int32[] values) {
   // ПРИМЕЧАНИЕ: если нужно, этот массив
   // можно передать другим методам.
```

```
   Int32 sum = 0;
   for (Int32 x = 0; x < values.Length; x++)
      sum += values[x];
   return sum;
}
```

В этом методе нет ничего незнакомого, кроме ключевого слова *params*, примененного к последнему параметру в сигнатуре метода. Если до времени закрыть глаза на новое ключевое слово, станет ясно, что этот метод принимает массив значений *Int32*, по очереди обрабатывает все элементы массива, складывая их значения, и возвращает полученную сумму.

Очевидно, этот метод можно вызвать так:

```
public static void Main() {
   // Выводит в консоль "15".
   Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 } ));
}
```

Ясно, что этот массив легко инициализировать произвольным числом элементов и передать для обработки методу *Add*. Показанный выше код немного неуклюж, хотя он корректно компилируется и работает. Мы, программисты, конечно, предпочли бы написать вызов *Add* так:

```
public static void Main() {
   // Выводит в консоль "15".
   Console.WriteLine(Add(1, 2, 3, 4, 5));
}
```

Наверное, вам будет приятно узнать, что это возможно благодаря ключевому слову *params*. Оно заставляет компилятор рассматривать параметр как экземпляр нестандартного атрибута *System.ParamArrayAttribute*.

Обнаружив вызов метода, компилятор C# проверяет все методы с заданным именем, у которых ни один из параметров не помечен атрибутом *ParamArray*. Если метод, способный принять вызов, есть, компилятор генерирует вызывающий его код, иначе компилятор ищет методы с атрибутом *ParamArray* и проверяет, могут ли они принять вызов. Если компилятор находит подходящий метод, то прежде, чем сгенерировать код для его вызова, компилятор генерирует код, создающий и заполняющий массив.

В предыдущем примере вы не найдете метод *Add*, принимающий пять *Int32*-совместимых аргументов. Однако компилятор видит в исходном тексте вызов метода *Add*, которому передается список значений *Int32*, и *Add*, у которого параметр-массив *Int32* помечен атрибутом *ParamArray*. Компилятор считает данный метод подходящим для этого вызова и генерирует код, собирающий все параметры в массив *Int32* и вызывающий *Add*. Мораль: можно написать вызов, который без труда передает методу *Add* кучу параметров, но в этом случае компилятор генерирует тот же код, что и для первой версии вызова метода *Add*, где массив создается и инициализируется явно.

Только последний параметр метода можно помечать ключевым словом *params* (*ParamArrayAttribute*). Этот параметр должен указывать одномерный массив любого типа. В последнем параметре метода допустимо передавать null или ссылку на мас-

сив, состоящий из 0 элементов. Следующий вызов метода *Add* прекрасно компилируется, отлично работает и дает в результате сумму, равную 0 (как и ожидалось):

```
public static void Main() {
   // Выводит в консоль "0".
   Console.WriteLine(Add());
}
```

Все показанные до сих пор примеры демонстрировали написание метода, принимающего произвольное число параметров типа *Int32*. Но как написать метод, принимающий произвольное число параметров любого типа? Ответ прост: достаточно модифицировать прототип метода так, чтобы вместо *Int32[]* он принимал *Object[]*. Вот метод, который выводит 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) {
      foreach (Object o in objects)
         Console.WriteLine(o.GetType());
   }
}
```

Если исполнить этот код, результат будет таков:

```
System.Object
System.Random
System.String
System.Int32
```

Внимание! Вызов метода, принимающего переменное число аргументов, снижает производительность. В любом случае всем объектам массива нужно выделить место в куче, элементы массива нужно инициализировать, а по завершении работы занятая массивом память должна быть очищена сборщиком мусора. Чтобы снизить отрицательное 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, в методе *Concat* определены несколько вариантов перегрузки, в которых ключевое слово *params* не используется. Эти версии метода *Concat* представляют наиболее часто используемые варианты перегрузки, которые, собственно, и предназначены для повышения эффективности работы в стандартных ситуациях. Варианты перегрузки с ключевым словом *params* предназначены для более редких ситуаций, в которых производительность страдает; радует только, что такие ситуации случаются нечасто.

Объявление типов параметров метода

При объявлении типов 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 экземпляр, такой как *IEnumerable<T>*, а не жесткий тип данных, такой как *List<T>*, или даже более жесткий тип — интерфейс, например *ICollection<T>* или *IList<T>*:

```
// Рекомендуется: в этом методе используется параметр мягкого типа.
public void ManipulateItems<T>(IEnumerable<T> collection) { ... }

// Не рекомендуется: в этом методе используется параметр жесткого типа.
public void ManipulateItems<T>(List<T> collection) { ... }
```

Причина, конечно же, в том, что можно вызывать первый метод, передавая объект-массив, объект *List<T>*, объект *String* и т. д. — любой объект, в типе которого реализован *IEnumerable<T>*. Второй метод принимает только объекты *List<T>*, он не примет массив или объект *String*. Ясно, что первый метод предпочтительнее, так как он гибче и может использоваться в более разнообразных ситуациях.

Естественно, что, создавая метод, принимающий список (а не просто любой перечислимый объект), нужно объявлять тип параметра как *IList<T>*. Все равно следует объявлять тип параметра как *List<T>*. Использование *IList<T>* позволяет вызывающему коду передавать массивы и другие объекты, в типе которых реализован *IList<T>*.

Заметьте, что в моих примерах речь шла о наборах, которые созданы с использованием архитектуры интерфейсов. Этот же подход годится, если речь идет о классах, использующих архитектуру базовых классов. Потому, к примеру, при реализации метода, обрабатывающего байты из потока *a stream*, напишем:

```
// Рекомендуется: в этом методе используется параметр мягкого типа.
public void ProcessBytes(Stream someStream) { ... }

// Не рекомендуется: в этом методе используется параметр жесткого типа.
public void ProcessBytes(FileStream fileStream) { ... }
```

Первый метод «умеет» обрабатывать байты из потока любого вида: *FileStream*, *NetworkStream*, *MemoryStream* и т. п. Второй метод поддерживает только *FileStream*, то есть область его применения существенно уже.

С другой стороны, всегда предпочтительнее объявлять тип возвращаемого методом объекта, используя наиболее жесткий доступный тип (в попытке не завязываться на конкретный тип). Например, лучше объявлять метод, возвращающий объект *FileStream*, а не *Stream*:

```
// Рекомендуется: в этом методе используется
// жесткий тип возвращаемого объекта.
public FileStream OpenFile() { ... }

// Не рекомендуется: в этом методе используется
// 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 второму методу требуется, чтобы вызывающий код рассчитывал только на объект *Stream*. Ясно, что первый метод предпочтительнее для вызывающего кода и может использоваться в более разнообразных ситуациях.

Иногда требуется сохранить возможность изменять внутреннюю реализацию метода, не влияя на вызывающий код. В приведенном выше примере, изменение реализации метода *OpenFile* в будущем маловероятно и он вряд ли будет возвращать что-либо отличное от объекта типа *FileStream* (или типа, производного от *FileStream*). Однако, если у вас есть метод, возвращающий объект *List<String>*, вполне возможно, что в будущем потребуется изменить реализацию метода так, чтобы он возвращал *String[]*. В случаях, когда требуется обеспечить некий задел на изменение метода в будущем, следует выбирать более мягкий тип возвращаемого объекта. Например, так:

```
// Гибкий вариант: в этом методе используется
// мягкий тип возвращаемого объекта.
public IList<String> GetStringCollection() { ... }

// Негибкий вариант: в этом методе используется
// жесткий тип возвращаемого объекта.
public List<String> GetStringCollection() { ... }
```

В этом примере, хотя в коде метода *GetStringCollection* используется и возвращается объект *List<String>*, лучше в прототипе метода указать в качестве возвращаемого объекта *IList<String>*. В будущем может измениться используемый в коде метода набор на *String[]*, но вызывающий код менять не потребуется, даже не придется перекомпилировать код. Обратите внимание, что в этом примере я использую самый жесткий из самых мягких типов — *IEnumerable<String>* или даже *ICollection<String>*.

Методы-константы и параметры-константы

В некоторых языках, в том числе в неуправляемом C++, можно объявлять методы и параметры как константы, которые запрещают коду экземплярного метода менять какие-либо поля метода или объекты, передаваемые в метод. CLR не поддерживает этой возможности, и многие программисты жаловались на ее отсутствие.

Так как сама исполняющая среда не поддерживает такой функции, естественно, что она отсутствует во всех языках (в том числе в C#).

Во-первых, следует заметить, что в неуправляемом C++ пометка экземплярного метода или параметра ключевым словом *const* гарантировала, что программист не мог стандартными средствами кода изменить объект или параметр. Внутри метода всегда существовала возможность написать код, который изменяет объект или параметр путем игнорирования их «константной» природы или получения адреса объекта или параметра и записи по этому адресу. В определенном смысле неуправляемый C++ «врал» программистам, утверждая, что константы-объекты или константы-параметры нельзя изменить даже при желании.

Создавая реализацию типа, разработчик может просто избегать написания кода, который изменяет объекты или параметры. Например, строки неизменяемы, так как класс *String* не предоставляет методы, позволяющие изменять объект-строку.

Также компании Microsoft очень сложно предусмотреть в CLR возможность проверки неизменности констант-объектов или констант-параметров. CLR пришлось бы при каждой операции записи проверять, не выполняется ли запись в объект-константу, а это сильно снизит эффективность работы программы. Естественно, что обнаружение нарушения должно приводить к генерации 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.

Вот всего лишь часть причин, по которым CLR не поддерживает методы-константы и параметры-константы.

ГЛАВА 9

Свойства

В этой главе я расскажу о свойствах. Свойства позволяют обращаться к методу в исходном тексте программы, используя упрощенный синтаксис. CLR поддерживает два вида свойств: без параметров, их называют просто — *свойства*, и с параметрами — у них в разных языках разное название. Например, в C# свойства с параметрами называют индексаторами, а в Visual Basic — свойствами по умолчанию.

Свойства без параметров

Многие типы определяют 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 определить любые сведения о его состоянии при помощи такого примерно кода:

```
Employee e = new Employee();
e.Name = "Jeffrey Richter";   // Определить имя сотрудника.
e.Age  = 41;                  // Определить возраст сотрудника.

Console.WriteLine(e.Name);    //  Выводим на экран "Jeffrey Richter".
```

Этот способ запроса и определения информации о состоянии объекта очень распространен. Но я готов спорить, что предыдущий код ни в коем случае не следует писать так, как в 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 пользования, так как в этом случае слишком просто написать код, способный испортить сведения о состоянии объекта путем ненадлежащего применения полей. Например, таким кодом разработчик может легко повредить объект *Employee*:

```
e.Age = -5; // Можете вообразить человека, которому -5 лет?
```

Есть и другие причины инкапсуляции доступа к полям данных типа. Допустим, вам нужен доступ к полю, чтобы что-то сделать, разместить в кеше некоторое значение или выполнить отложенное создание какого-то внутреннего объекта, при этом обращение к полю не должно нарушать безопасность потоков. Или, скажем, поле — это логическое поле, значение которого представлено не байтами в памяти, а вычисляется по некоторому алгоритму.

Любая из этих причин заставляет меня рекомендовать при разработке типов, во-первых, помечать все поля как закрытые, и, во-вторых, чтобы дать пользователю вашего типа возможность получения и определения сведений о состоянии, следует создавать специальные методы, которые служат именно этой цели. Методы, выполняющие функции оболочки для доступа к полю, обычно называют *аксессорами* (accessor). Аксессоры могут выполнять дополнительную «зачистку», гарантируя, что сведения о состоянии объекта не нарушатся. Я переписал класс из примера так:

```
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 этого достаточно опустить один из аксессоров.

Как видите, у инкапсуляции данных два недостатка: во-первых, приходится писать более длинный код из-за необходимости реализации дополнительных методов, во-вторых, вместо простой ссылки на имя поля пользователям типа придется 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);                        // Генерируется исключение.
ArgumentOutOfRangeException.
Int32 EmployeeAge = e.GetAge();      // Получение возраста сотрудника.
```

Лично я считаю эти недостатки незначительными. И все же CLR поддерживает механизм, частично компенсирующий первый недостаток и полностью устраняющий второй. Этот механизм — свойства.

Этот класс функционально идентичен показанному выше, 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, тот факт, что они позволяют писать код следующим образом, более чем компенсирует дополнительную работу:

```
e.Name = "Jeffrey Richter";      // "Задать" имя сотрудника.
String EmployeeName = e.Name;    // "Получить" имя сотрудника.
e.Age = 41;                      // "Задать" возраст сотрудника.
e.Age = -5;                      // Генерируется исключение
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. о них главу 6) и определяться в интерфейсах (см. главу 14).

У каждого свойства есть имя и тип (который не может быть *void*). Нельзя перегружать свойства (то есть определять несколько свойств с одинаковыми именами, но с разным типом). Определяя свойство, обычно определяют пару методов: *get* и *set*. Однако, опустив *set*, можно определить свойство, доступное только для чтения, а если оставить только *get*, получится свойство, доступное только для записи.

Методы *get* и *set* свойства довольно часто манипулируют закрытым полем, определенным в типе. Это поле обычно называют *вспомогательным полем* (backing field). Однако методам *get* и *set* не приходится обращаться к вспомогательному полю. Так, 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, — 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:

- метод-аксессор *get* этого свойства — генерируется, только если для свойства определен аксессор *get*;
- метод-аксессор *set* этого свойства — генерируется, только если для свойства определен аксессор *set*;
- определение свойства в метаданных управляемого модуля — генерируется всегда.

Вернемся к показанному выше 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 так:

```
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.

В C# есть встроенная поддержка свойств. Обнаружив код, пытающийся получить или определить свойство, компилятор на самом деле генерирует вызов соответствующего метода. Если используемый язык не поддерживает свойства напрямую, к ним все равно можно обращаться через вызов нужного аксессора. Эффект тот же, только исходный текст выглядит менее изящным.

Помимо аксессоров, 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 аксессоры *get* и *set*. Эта информация существует лишь затем, чтобы провести связь между абстрактным понятием «свойства» и его методами-аксессорами. Компиляторы и другие инструменты могут использовать эти метаданные, которые можно получить через класс *System.Reflection.PropertyInfo*. И все же CLR не использует эти метаданные, требуя при выполнении только методы-аксессоры.

Осторожный подход к определению свойств

Лично мне свойства не нравятся, и я был бы рад, если бы их поддержку убрали из Microsoft .NET Framework и сопутствующих языков программирования. Причина в том, что свойства выглядят как поля, на самом деле являясь методами. Это порождает массу заблуждений и непонимания. Столкнувшись с кодом, обращающимся к полю, разработчик привычно предполагает наличие массы условий, которые просто не всегда верны, если речь идет о свойстве. В частности:

- свойства могут быть «только для чтения» или «только для записи», а поля всегда доступны и для чтения, и для записи. Определяя свойство, лучше всего создавать для него оба аксессора — *get* и *set*;
- метод свойства может привести к исключению, а при доступе к полям исключений не бывает;
- свойства нельзя передавать в метод как параметры 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);
   }
}
```

- метод свойства может выполняться довольно долго, а доступ к полям выполняется моментально. Часто свойства используют 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 *System.MarshalByRefObject*), вызов метода свойства выполняется очень медленно, поэтому предпочтение следует отдать методу. Я считаю, что в классах, производных от *MarshalByRefObject*, никогда не следует использовать свойства;
- при вызове несколько раз подряд метод свойства может возвращать разные значения, а поле возвращает одно и то же значение. В классе *System.DateTime* есть неизменяемое свойство *Now*, которое возвращает текущую 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;
- метод свойства может создавать наблюдаемые сторонние эффекты, а при доступе к полю это невозможно. Иначе говоря, порядок определения значений различных свойств типа никак не должен влиять на поведение типа, однако в действительности часто бывает не так;
- методу свойства 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 замешательства для разработчиков, причем это поведение 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 и удобно и не вызывает замешательства у разработчика. Единственная привлекательная черта свойств — упрощенный синтаксис, остальное — недостатки, в числе которых проигрыш в производительности и читабельности кода. Если бы я проектировал .NET Framework и компиляторы, я бы вообще отказался от свойств, вместо этого я предоставил бы разработчикам полную свободу реализации методов *Get*Xxx и *Set*Xxx. 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 происходит — вызов метода!

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 по ощущениям напоминает обращение к полю. Помимо таких «полеобразных» свойств, языки программирования поддерживают то, что я называю *свойствами с параметрами* (parameterful

properties), у которых 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# — *индексаторы*, а 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 способ, позволяющий разработчику на C# перегружать оператор *[]*. Вот пример типа *BitArray*, который позволяет индексировать набор битов, поддерживаемый экземпляром типа, используя синтаксис массива:

```
using System;

public sealed class BitArray {
   // Закрытый массив байт, хранящий биты.
   private Byte[] m_byteArray;
   private Int32  m_numBits;

   // Конструктор, выделяющий память для массива байт
   // и устанавливающий все биты равными 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("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 битов.
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 типа *Int32* — *bitPos*. 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 создают 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 также 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 нее любое свойство — это всего лишь пара методов, определенных внутри типа. Как сказано выше, в различных языках синтаксис создания и использования свойств с параметрами различны. 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-аксессор *set* свойства с параметрами — генерируется, только если у свойства определен аксессор *set*;
- метод-аксессор *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 индексатора типа *BitArray*, показанного выше, происходит так, как если бы он исходно был написан так:

```
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 выбирать имя методов-аксессоров, 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.GenericList* предлагает открытое свойство-экземпляр *Item*, которое является индексатором объекта *List*.

Программируя на C#, вы никогда не увидите имя *Item*, поэтому выбор его компилятором обычно не должен вызывать беспокойства. Но, если вы конструируете 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.CompilerServicesIndexerNameAttribute*. Например:

```
using System;
using System.Runtime.CompilerServices;

public sealed class BitArray {

   [IndexerName("Bit")]
   public Boolean this[Int32 bitPos] {
      // Здесь определен по крайней мере один метод-аксессор.
   }
}
```

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 не создается в метаданных сборки.[1]

Вот фрагмент кода на 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 определять несколько индексаторов с одинаковой сигнатурой, поскольку их имена могут отличаться. Однако C# не допускает этого, так как принятый в нем синтаксис не ссылается на индексатор по имени, а значит, компилятор не будет знать, на какой индексатор ссылаются. Попытка компиляции следующего исходного текста на C# заставляет компилятор генерировать сообщение об ошибке «error CS0111: Class 'SomeType' already defines a member called 'this' with the same parameter types» («ошибка CS0111: в классе 'SomeType' уже определен член 'this' с таким же типом параметра»).

```
using System;
using System.Runtime.CompilerServices;

public sealed class SomeType {

   // Определяем метод-аксессор get_Item.
   public Int32 this[Boolean b] {
      get { return 0; }
   }

   // Определяем метод-аксессор get_Jeff.
   [IndexerName("Jeff")]
   public String this[Boolean b] {
      get { return null; }
   }
}
```

Как видите, C# представляет себе индексаторы как способ перегрузки оператора *[]*, и этот оператор не позволяет различать свойства с одинаковыми наборами параметров, но различающиеся лишь именами аксессоров.

Кстати, примером типа с измененным именем индексатора может быть *System.String*, в котором имя индексатора *String — Chars*, а не *Item*. Это свойство по-

[1] По этой причине класс *IndexerNameAttribute* не входит в описанные в ECMA стандарты CLI и языка C#.

зволяет 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.

Выбор главного свойства с параметрами

При анализе ограничений, которые C# налагает на индексаторы, возникают два вопроса.

- Что, если язык, на котором написан тип, позволяет разработчику определять несколько свойств с параметрами?
- Как задействовать этот тип в C#-программе?

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 C# 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 C#, у которого определено свойство с параметрами, но нет атрибута *IndexerName*, атрибут *DefaultMember*, которым помечен 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, имеющие разные имена.

В программах на языке, поддерживающем несколько свойств с параметрами, нужно выбрать один метод свойств и пометить определяющий его тип атрибутом *DefaultMember*. Это будет единственное свойство с параметрами, доступное C#-программам.

Обнаружив код, пытающийся получить или заказать значение индексатора, компилятор C# генерирует вызов соответствующего метода-аксессора. 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-аксессор. CLR не различает свойства с параметрами и без параметров, поэтому для поиска связи между свойством с параметрами и его методами-аксессорами служит все тот же класс *System.Reflection.PropertyInfo*.

Производительность при вызове аксессоров свойств

В случае простых методов-аксессоров *get* и *set* JIT-компилятор *встраивает* код аксессора внутрь кода вызываемого метода, поэтому характерное для использования свойств вместо полей снижение производительности работы программы не наблюдается. Встраивание (inline) подразумевает компиляцию кода метода (или,

в данном случае, аксессора) непосредственно в составе кода 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 встраивание может приводить к сокращению общего объема машинного кода, а, значит, повышению скорости выполнения.

Заметьте: при отладке JIT-компилятор не встраивает методы свойств, потому что встроенный код отлаживать сложнее. Это означает, что эффективность доступа к свойству в готовой версии программы выше, чем в отладочной. Скорость доступа к полям одинакова в обеих версиях.

Доступность аксессоров свойств

Иногда при проектировании типа желательно задать разный уровень доступа к аксессорам *get* и *set*. Чаще всего требуется открытый аксессор *get* и закрытый аксессор *set*:

```
public class SomeType {
   public String Name {
      get { return null; }
      protected set {}
   }
}
```

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* и доступен для вызова только из кода *SomeType* или кода класса, производного от *SomeType*.

При определении свойства с методами-аксессорами с различным уровнем доступа синтаксис C# требует, чтобы само свойство было объявлено с наименее жестким уровнем доступа, а более жесткое ограничение было наложено только на один из методов-аксессоров. 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. Но 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 предоставить какое-либо поведение объекта — обобщенное или нет, создайте метод, а не свойство.

ГЛАВА 10

События

Предмет этой главы — последний вид членов, определяемых типами, — события. Если в типе определен член-событие, то этот тип (или его экземпляр) может уведомлять другие объекты о некоторых особых событиях. Скажем, класс *Button* («кнопка») определяет событие *Click* («щелчок»). В приложении могут быть объекты, которые должны 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:

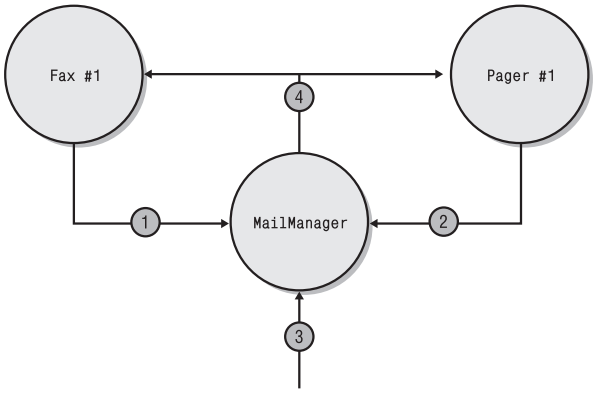
- регистрацию статического метода типа или экземплярного метода объекта, заинтересованных в получении уведомления о событии;
- отмену регистрации статического метода типа или экземплярного метода объекта, получающих уведомления о событии;
- уведомление зарегистрированных методов о том, что событие произошло.

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 (callback method), не нарушая безопасности типов. Метод обратного вызова — 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 15.

Чтобы помочь вам досконально разобраться в работе событий в CLR, я начну с примера ситуации, в которой 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 зарегистрироваться 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. 10-1 показано, как инициализируется приложение и что происходит при получении сообщения.



1. Объект Fax регистрируется для получения уведомлений о событии объекта MailManager.
2. Объект Pager регистрируется для получения уведомлений о событии объекта MailManager.
3. MailManager получает новое почтовое сообщение.
4. MailManager уведомляет все зарегистрированные у него объекты, которые обрабатывают пришедшее сообщение, как им нужно.

Рис. 10-1. *Архитектура приложения, в котором используются события*

Это приложение работает так. При его инициализации создается экземпляр объекта *MailManager*, поддерживающего событие *NewMail*. Во время создания объекты *Fax* и *Pager* регистрируются для получения уведомлений о событии *NewMail* (получение нового сообщения) объекта *MailManager*, так что *MailManager* знает, что эти объекты следует уведомить о прибытии нового сообщения. Если в дальнейшем *MailManager* примет новое сообщение, это вызовет возникновение события *NewMail*, что позволит всем зарегистрировавшимся объектам выполнить нужную обработку нового сообщения.

Проектирование типа, поддерживающего событие

Создание типа, поддерживающего одно или более событий, требует от разработчика выполнения многих этапов. В этом разделе я расскажу о каждом из них. Приложение-пример *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, который будет хранить всю дополнительную информацию, передаваемую получателям уведомления о событии

При возникновении события объекту, в котором оно возникло, нужно передать дополнительную информацию объектам-получателям уведомления о событии. Для предоставления получателям эту информацию нужно инкапсулировать в собственный класс, который содержит набор закрытых полей, а также открытых свойств «только для чтения». В соответствии с соглашением, классы, содержащие инфор-

мацию о событиях, передаваемую обработчику события, должны наследовать *SystemEventArgs*, 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, хранящего информацию,
// 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 так:

```
[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. Так, в случае уведомления объектом *Button* о щелчке кнопки, вызов метода обратного вызова и есть вся нужная информация. Определяя событие, не передающее дополнительные данные, можно не создавать новый объект *EventArgs*, достаточно просто воспользоваться *EventArgs.Empty*.

Этап 2: определение члена-события

В C# член-событие объявляется ключевым словом *event*. Каждому члену-событию определяется 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;
   ...
}
```

NewMail — имя события, а тип члена-события — *EventHandler<NewMailEventArgs>*, это означает, что получатели уведомления о событии должны предоставлять метод обратного вызова, прототип которого совпадает с типом-делегатом *EventHandler<NewMailEventArgs>*. Поскольку обобщенный делегат *System.EventHandler* определен так:

```
public delegate void EventHandler<TEventArgs>
   (Object sender, TEventArgs e) where TEventArgs: EventArgs;
```

метод должен выглядеть так:

```
void MethodName(Object sender, NewMailEventArgs e);
```

Примечание Многих удивляет, почему механизм событий требует, чтобы параметр *sender* был типа *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);
```

Причина необходимости того, чтобы параметр *sender* был типа *Object*, — в наследовании. 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 *SmtpMailManager* просто унаследовал 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 *sender* был типа *Object*.

Еще одна причина того, что *sender* относят к типу *Object* — простая гибкость. Это позволяет использовать делегат нескольким типам, которые поддерживают событие, передающее объект *NewMailEventArgs*. В частности, класс *PopMailManager* мог бы использовать делегат, даже если бы не наследовал классу *MailManager*.

И еще одно: механизм событий требует, чтобы в имени делегата и методе обратного вызова производный от *EventArgs* параметр назывался *e*. Единственная причина — обеспечить дополнительное однообразие, облегчая и упрощая для разработчиков изучение и реализацию событий. Инструменты создания кода (например, такой как Microsoft Visual Studio) так же в курсе, что нужно вызывать параметр *e*.

И последнее, механизм событий требует, чтобы все обработчики возвращали *void*. Это обязательно, потому что при возникновении события могут вызываться несколько методов обратного вызова и невозможно получить у них всех возвращаемое значение. Тип *void* просто запрещает методам возвращать какое бы то ни было значение. К сожалению, в библиотеке FCL есть обработчики событий, в частности *ResolveEventHandler*, 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 уведомляет зарегистрированные методы 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 невиртуальным.
   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);
   }
   ...
}
```

Обратите внимание, что метод *OnNewMail* определяет временную локальную переменную *temp*, которая инициализируется самим членом-событием. Далее *temp* сравнивается с null и в случае неравенства используется для инициирования события. Эта временная переменная необходима для предотвращения возможных проблем с синхронизацией потоков. Видите ли, если вместо использования *temp* просто сослаться на *NewMail*, вполне возможно, что поток «увидит», что *NewMail* не равно null, но позднее, непосредственно перед генерацией события, другой поток может переопределить *NewMail*, присвоив null. В такой ситуации при попытке инициировать событие CLR вернет исключение *NullReferenceException*. Временная переменная *temp* решает проблему и избавляет от возможности генерации исключения *NullReferenceException*.

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 пересылки уведомления о событии.

Этап 4: определение метода, транслирующего входную информацию в желаемое событие

У класса должен быть метод, принимающий некоторую входную информацию и преобразующий его в генерацию события. В примере *MailManager* метод *SimulateNewMail* вызывается для оповещения о прибытии 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 сказано выше, тип, производный от *MailManager*, может переопределять это действие.)

Как реализуются события

Научившись определять класс с членом-событием, пора 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 строчки компилятор превращает ее в следующие три конструкции:

```
// 1. ЗАКРЫТОЕ поле делегата, инициализированное null.
private EventHandler<NewMailEventArgs> NewMail = null;

// 2. ОТКРЫТЫЙ метод add_Xxx (где Xxx - это имя события).
//  Позволяет объектам регистрироваться для получения уведомлений о событии.
[MethodImpl(MethodImplOptions.Synchronized)]
public void add_NewMail(EventHandler<NewMailEventArgs> value) {
   NewMail = (EventHandler<NewMailEventArgs>)
      Delegate.Combine(NewMail, value);
}
```

```
// 3. ОТКРЫТЫЙ метод remove_Xxx (где Xxx - это имя события).
// Позволяет объектам отменять регистрацию для получения уведомлений о событии
[MethodImpl(MethodImplOptions.Synchronized)]
public void remove_NewMail(EventHandler<NewMailEventArgs> value) {
   NewMail = (EventHandler<NewMailEventArgs>)
      Delegate.Remove(NewMail, value);
}
```

Первая конструкция — просто поле соответствующего типа делегата. Оно содержит ссылку на заголовок списка делегатов, которые будут уведомляться при возникновении события. Поле инициализируется значением null, что означает, что нет получателей, ожидающих уведомления о событии. Когда метод регистрирует получателя уведомления, это поле ссылается 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 экземпляр 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 в предотвращении доступа 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 на получение 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.

Заметим также, что оба метода — *add* и *remove* — помечены атрибутом *MethodImplAttribute* (определенным в пространстве имен *System.Runtime.CompilerServices*). Точнее, эти методы помечены как синхронизированные, поэтому они не нарушают безопасность потоков: множество объектов, следящих за событием, может регистрироваться или отменять свою регистрацию одновременно, не нарушая целостность связного списка.

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 *add* и *remove*, сгенерированные компилятором, тоже были бы объявлены как закрытые. Так что, когда в типе определяется событие, модификатор доступа события определяет, 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 также 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-аксессоры *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 CLR не использует эти метаданные и во время выполнения требует лишь методы-аксессоры.

Создание типа, отслеживающего событие

Ну, самое трудное позади. В этом разделе я покажу, 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 в 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
   // на получение 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 *EventHandler<NewMailEventArgs>* делегат, инкапсулирующий 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 таким инструментом, как 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 при этом получается не столь изящным. Именно метод *add*, регистрирующий делегат для уведомления о событии, добавляет делегат в связный 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, а в этом 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 для уведомления о событии *NewMail*, если пользователю больше не нужно пересылать сообщения электронной почты по факсу. Пока объект зарегистрирован для уведомления о событии другого объекта, он не может стать добычей сборщика мусора. Если в вашем типе реализован метод *Dispose* объекта *IDisposable*, уничтожение объекта должно вызвать отмену его регистрации для уведомления обо всех событиях (об объекте *IDisposable* см. также главу 20).

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 связного списка в поисках делегата-оболочки метода, соответствующего переданному методу обратного вызова. Если обнаружено совпадение, делегат удаляется из связного списка делегатов события. Если нет, ошибка не возникает, и список делегатов события остается неизменным.

Кстати, C# требует, чтобы для добавления и удаления делегатов из связного списка в ваших программах использовались операторы «+=» и «–=». Если попытаться обратиться к методам *add* или *remove* напрямую, компилятор C# сгенерирует сообщение об ошибке «CS0571: cannot explicitly call operator or accessor» («CS0571: оператор или аксессор нельзя вызывать явно»).

События и безопасность потоков

В предыдущем разделе я продемонстрировал, как компилятор C# добавляет атрибут *[MethodImpl(MethodImplOptions.Synchronized)]* в методы *add* или *remove* события. Задача этого атрибута — гарантировать, что только методы *add* или *remove* будут выполняться в типе при обработке статического члена-события. Эта синхронизация потоков необходима для обеспечения целостности списка объектов-делегатов. Однако надо иметь в виду, что есть много проблем, возникающих из-за способа реализации этой синхронизации потоков в CLR.

При применении атрибута MethodImpl к экземплярному (нестатическому) методу среда CLR использует сам объект в качестве блокировки, предназначенной для синхронизации потоков. Это означает, что, если в классе определено много событий, все методы *add* или *remove* используют одну блокировку, а это отрицательно сказывается на масштабируемости, если есть несколько потоков, одновременно регистрирующихся и отказывающихся от регистрации на подписку на различные события. Это очень редкая ситуация, и в большинстве случаев описанная проблема себя не проявляет. Тем не менее правила синхронизации потоков требуют, чтобы методы не использовали для блокировки собственный объект, потому что блокировка открыта и доступна любому внешнему коду. Это означает, что любой может написать код, блокирующий объект и, в принципе, способный вызвать взаимную блокировку. Если требуется обеспечить защиту объекта от таких проблем и сделать его работу надежнее, для блокировки следует использовать другой объект. Как это сделать, я расскажу в следующем разделе.

При применении атрибута *[MethodImpl(MethodImplOptions.Synchronized)]* к статическому методу среда CLR использует объект-тип в качестве блокировки, предназначенной для синхронизации потоков. И это означает, что, если в классе определено много событий, все методы *add* или *remove* используют одну блокиров-

ку и это отрицательно сказывается на масштабируемости, если есть несколько потоков одновременно, регистрирующихся и отказывающихся от регистрации на подписку на различные события. И снова, такая ситуация случается нечасто.

Однако есть значительно более серьезная проблема: правила синхронизации потоков требуют, чтобы методы не использовали для блокировки собственный объект, потому что блокировка открыта и доступна любому внешнему коду. В добавок, в CLR есть ошибка, связанная с загрузкой без привязки к конкретному домену приложения. В такой ситуации блокировка совместно применяется всеми доменами AppDomain, использующими тип, что позволяет коду одного домена нарушить работу кода другого домена. В действительности, компилятор C# должен был бы делать совсем другое для обеспечения безопасности потоков при работе методов *add* и *remove*. В следующем разделе я расскажу о механизме, который можно использовать для устранения недостатка компилятора C#.

C# и CLR позволяют определить значимый тип (структуру) с одним или несколькими (нестатическими) членами-событиями, но нужно иметь в виду, что при этом компилятор C# не обеспечивает никакой безопасности потоков. Причина в том, что у неупакованных значимых типов нет связанного с ними объекта блокировки. В сущности компилятор C# не обозначает атрибутом *[MethodImpl(MethodImplOptions.Synchronized)]* методы *add* и *remove*, потому что этот атрибут никак не повлияет на экземплярные методы значимого типа. К сожалению, нет универсального способа обеспечить безопасность потоков событий экземпляров, если они определены как члены значимого типа, поэтому рекомендуется избегать этого. Стоит заметить, что вполне допустимо (с описанными выше ограничениями) определять статические события значимого типа, так как для блокировки они используют объект-тип (а это ссылочный тип). Однако, если вас интересует по-настоящему надежное решение, рекомендую использовать механизм, описанный в следующем разделе.

Явное управление регистрацией событий

Порой методы *add* и *remove*, сгенерированные компилятором, далеки от идеала. Ранее я уже говорил о всех проблемах, связанных с безопасностью потоков, существующих в компиляторе C# компании Microsoft. Вообще-то компилятор C# компании Microsoft никогда не обеспечивает максимальную безопасность, если речь идет о программировании безопасного и надежного кода. Чтобы создавать исключительно надежные компоненты, рекомендую всегда использовать прием, описанный в этом разделе — он позволит решить все проблемы безопасности потоков. Впрочем, его можно использовать и для решения других задач. В частности, очень распространенная причина, побуждающая программистов к самостоятельной реализации методов *add* и *remove*, — определение в типе множества событий, когда одновременно требуется эффективнее использовать память. В следующем разделе мы разберем такой сценарий подробнее.

К счастью, компилятор C#, как и многие другие, позволяет разработчикам явно реализовывать методы-аксессоры *add* и *remove*. Вот как можно модифицировать объект *MailManager* для обеспечения безопасной для потоков регистрации и отмены регистрации на уведомление о событии:

```
internal class MailManager {

   // Создаем закрытое экземплярное поле
   // для блокировки синхронизации потоков.
   private readonly Object m_eventLock = new Object();

   // Создаем закрытое поле, ссылающееся на заголовок списка делегатов.
   private EventHandler<NewMailEventArgs> m_NewMail;

   // Создаем в классе член-событие.
   public event EventHandler<NewMailEventArgs> NewMail {
      // Явно реализуем метод add.
      add {
         // Берем закрытую блокировку и добавляем обработчик
         // (передаваемый по значению) в список делегатов.
         lock (m_eventLock) { m_NewMail += value; }
      }

      // Явно реализуем метод remove.
      remove {
         // Берем закрытую блокировку и удаляем обработчик
         // (передаваемый по значению) из списка делегатов.
         lock (m_eventLock) { m_NewMail -= value; }
      }
   }

   // Определяем метод, отвечающий за инициирование события
   // и информирование об этом зарегистрированных объектов.
   // Если класс изолирован, определяем метод как закрытый и невиртуальный.
   protected virtual void OnNewMail(NewMailEventArgs e) {

      // Сохраняем поле делегата во временном поле для обеспечения безопасности потоков.
      EventHandler<NewMailEventArgs> temp = m_NewMail;

      // Если есть зарегистрировавшиеся объекты,
      // уведомляем их.
      if (temp != null) temp(this, e);
   }

   // Создаем метод, преобразующий входную информацию
   // в требуемое событие.
   public void SimulateNewMail(String from, String to, String subject) {

      // Создаем объект, хранящий информацию, которую нужно передать
      // объектам, получающим уведомление о событии.
      NewMailEventArgs e = new NewMailEventArgs(from, to, subject);

      // Вызываем виртуальный метод, уведомляющий наш объект о возникновении события.
      // Если нет типа, переопределяющего этот метод,
      // наш объект уведомит все объекты, подписавшиеся
```

```
        // на уведомление о событии.
        OnNewMail(e);
    }
}
```

В новой версии объекта *MailManager* закрытое поле *m_NewMail*, ссылающееся на список делегатов, должно определяться явно. В оригинальном синтаксисе события компилятор C# автоматически определял закрытое поле. В новом синтаксисе, в котором разработчик явно предоставляет реализацию методов-аксессоров *add* и *remove*, поля также должны объявляться явно.

Это поле просто представляет собой ссылку на делегат *EventHandler<NewMailEventArgs>*. Нет ничего, что превращает это поле в событие. В новом расширенном синтаксисе после ключевого слова *event* следует то, что собственно определяет событие в типе. В блоках *add* и *remove* содержится реализация методов-аксессоров. Заметьте: каждый метод принимает скрытый параметр по имени *value* типа *EventHandler<NewMailEventArgs>*. Код внутри методов выполняет все операции по добавлению и удаление делегатов из списка. В отличие от свойств, у которых есть оба или хотя бы один метод-аксессор *get* или *set*, у событий всегда должны быть оба метода-аксессора *add* и *remove*.

Явная реализация методов-аксессоров, показанная в предыдущем примере, работает как методы-аксессоры, сгенерированные компилятором C#, если не считать отсутствия атрибута *[MethodImpl(MethodImplOptions.Synchronized)]*, а вместо этого используется оператор *lock* языка C# со ссылкой на закрытый определенный объект *m_eventLock* типа *Object*. Именно так я решаю проблему небезопасности потоков, о которой говорил в предыдущем разделе. Поскольку поле *m_eventLock* объявляется как закрытое, никакой код, за исключением кода класса *MailManager*, не в состоянии получить доступ к нему; за счет этого повышается надежность класса *MailManager*.

Событие можно объявлять как статический член, отчего методы-аксессоры также становятся статическими. Естественно, что закрытое поле *m_NewMail* также станет статическим. В таком случае для обеспечения безопасности потоков поле *m_eventLock* также нужно объявить статическим. Это применимо как к ссылочным, так и значимым типам, в которых требуется предоставить статическое событие безопасно для потоков. К сожалению, как говорилось в предыдущем разделе, не существует хорошего способа сделать события экземпляров значимого типа безопасными с точки зрения потоков из-за того, что нет хорошего способа инициализировать экземплярное поле в значимом типе. За объяснением причин отсылаю вас к главе 5.

Код, регистрирующий и отменяющий регистрацию события, не различает методы *add* и *remove*, созданные автоматически компилятором или явно реализованные разработчиком. На самом деле это не исключает возможности использования в исходном тексте операторов «+=» и «–=», при этом компилятор будет знать, что следует сгенерировать вызовы явно определенных методов.

И последнее замечание относительно метода *OnNewMail*. Семантически этот метод идентичен предыдущей версии. Единственная разница в том, что имя события (*NewMail*) заменено на имя поля делегата, *m_NewMail*.

Конструирование типа с множеством событий

В предыдущем разделе я описал ситуацию, в которой могут понадобиться явные методы-аксессоры *add* и *remove* события. Когда вы создаете явную реализацию аксессоров сами, у вас немного больше свободы творчества. Рассмотрим, как при помощи явной реализации этих методов можно уменьшить использование памяти приложением.

Тип *System.Windows.Forms.Control* определяет около 70 событий. Если бы тип *Control* реализовал эти события, позволяя компилятору неявно генерировать аксессоры *add* и *remove* и поля делегатов, то у каждого объекта типа *Control* пришлось бы создавать 70 полей делегатов лишь для одних событий! Поскольку объекты никогда не регистрируются для уведомления о большинстве из этих событий, то при создании каждого объекта типа, производного от *Control*, огромное количество памяти просто пропадало бы зря. Кстати, тип *System.Web.UI.Control* также использует технологию, о которой пойдет речь ниже, чтобы сэкономить память, которая в противном случае напрасно тратится на обслуживание неиспользуемых событий.

Если подойти к явной реализации методов *add* и *remove* творчески, можно значительно уменьшить объем памяти, бесполезно затраченной каждым объектом. В этом разделе я покажу, как определить тип, способный эффективно поддерживать множество событий.

Идея такова: каждый объект поддерживает набор (обычно это словарь) идентификаторов событий, используемых в качестве ключа, и списка делегатов в качестве значений. При создании нового объекта этот набор пуст. При регистрации объекта для уведомления о событии происходит поиск идентификатора этого события в наборе. Если оно там обнаружено, новый делегат объединяется со списком делегатов этого события. Если нет, событие добавляется вместе с соответствующим делегатом.

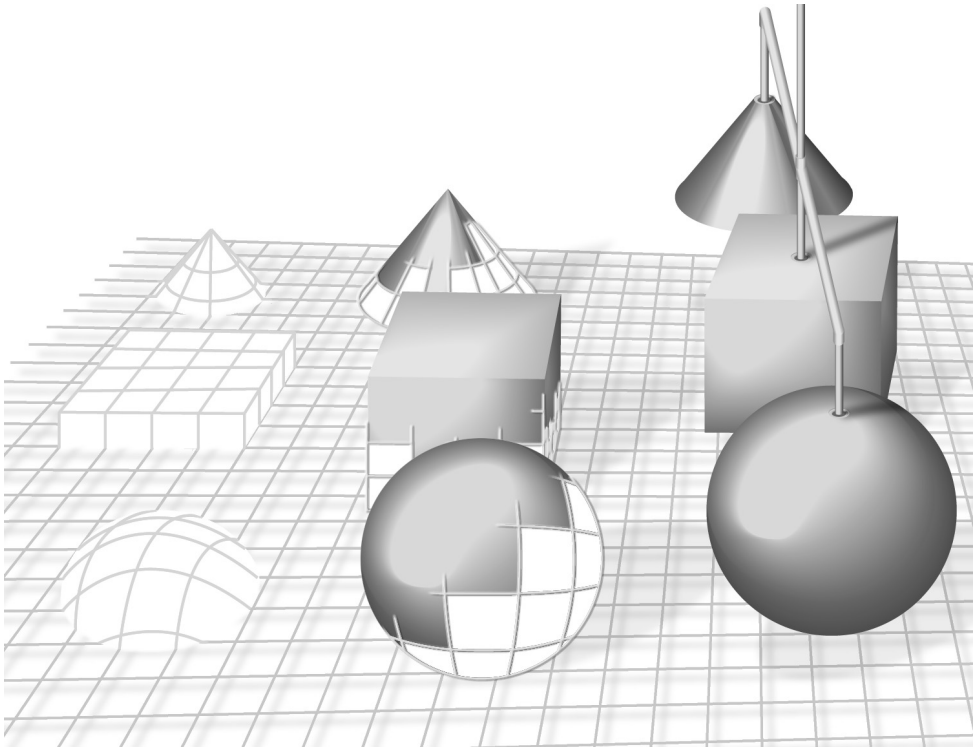
Когда объект должен инициировать событие, происходит его поиск в наборе. Если там нет элемента для этого идентификатора события, следовательно, ни один объект не требует уведомления о нем, и никакого обратного вызова какого-либо делегата не требуется. Если событие в наборе есть, вызывается список делегатов, ассоциированный с событием. Реализация этой архитектуры остается на совести программиста, разрабатывающего тип, в котором определены соответствующие события; разработчик, использующий тип, ничего не знает о внутренней реализации событий.

Как реализовать все описанное выше, демонстрирует приложение-пример *TypeWithLotsOfEvents,* которое можно скачать с сайта *http://wintellect.com*. Код оснащен подробными комментариями, и, если вы поняли все сказанное ранее в этой главе, у вас не будет проблем с его пониманием и адаптацией для решения собственных задач.

Примечание В моем коде используется вспомогательный класс *EventSet*, который поддерживает словарь идентификаторов-ключей событий и значений делегатов. В FCL определен тип *System.ComponentModel.EventHandlerList*, который делает в сущности то же, что и мой тип *EventSet*. Типы *System.Windows.Forms.Control* и *System.Web.UI.Control* внутренне используют тип *EventHandlerList* для обслуживания своих разреженных наборов событий. Вы, несомненно, можете использовать определенный в FCL тип *EventHandlerList*. Разница между *EventHandlerList* и моим типом *EventSet* в том, что *EventHandlerList* использует связный список вместо хеш-таблицы. Это означает, что тип *EventHandlerList* не поддерживает доступ к набору, обеспечивающий безопасность потоков. Если требуется безопасность потоков, вам придется самим создавать оболочку для набора *EventHandlerList*. Мой класс *EventSet* обеспечивает полную безопасность потоков.

ЧАСТЬ IV

ВАЖНЕЙШИЕ ТИПЫ



ГЛАВА 11

Символы, строки и обработка текста

Эта глава посвящена приемам работы с отдельными символами, а также строками в Microsoft .NET Framework. Вначале мы рассмотрим структуру *System.Char* и способы работы с символами. Потом перейдем к весьма полезному классу *System.String*, предназначенному для работы с неизменяемыми строками. (Такую строку можно создать, а изменить — нет.) Затем я расскажу о динамическом создании строк с помощью класса *System.Text.StringBuilder*. Выходя за рамки основной темы главы, мы обсудим форматирование объектов в строки и эффективное сохранение и передачу строк различными способами. В конце я расскажу 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 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 *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*. Заметьте: в качестве параметров они принимают либо одиночный символ, либо *String* с указанием индекса символа в *String*.

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 региональ-

ных 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 *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/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;
      // '\u0033' - это "цифра 3".
      d = Char.GetNumericValue('\u0033'); // Параметр '3' даст тот же результат.
      Console.WriteLine(d.ToString());    // Отображается "3".

      // '\u00bc' - это "простая дробь одна четвертая ('1/4')".
      d = Char.GetNumericValue('\u00bc');
      Console.WriteLine(d.ToString());    // Отображает "0.25".

      // 'A' - это "Латинская прописная буква A".
      d = Char.GetNumericValue('A');
      Console.WriteLine(d.ToString());    // Отображает "-1".
   }
}
```

А теперь представлю в порядке предпочтения три способа преобразования различных числовых типов в экземпляры *Char* и наоборот.

- **Приведение типа** Самый эффективный способ, так как компилятор генерирует IL-команды преобразования без вызовов каких-либо методов. Для преобразования *Char* в числовое значение, такое как *Int32*, он 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, что-

бы в случае потери данных при преобразовании возникало 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* генерируют исключение *SystemInvalidCastException*, если преобразование невозможно (скажем, преобразование типа *Char* в *Boolean*) или грозит потерей данных. Во многих типах (в том числе *Char* и числовых типах FCL) методы *IConvertible* являются явно реализованными методами интерфейса (см. главу 14), 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);                  // Отображает "A".

      n = (Int32) c;
      Console.WriteLine(n);                  // Отображает "65".

      c = unchecked((Char) (65536 + 65));
      Console.WriteLine(c);                  // Отображает "A".


      // Преобразование "число - символ" с помощью Convert.
      c = Convert.ToChar(65);
      Console.WriteLine(c);                  // Отображает "A".

      n = Convert.ToInt32(c);
      Console.WriteLine(n);                  // Отображает "65".

      // Этот код демонстрирует проверку допустимых значений для Convert.
      try {
         c = Convert.ToChar(70000);    // Слишком большое 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 IConvertible.
      c = ((IConvertible) 65).ToChar(null);
      Console.WriteLine(c);                  // Отображает "A".

      n = ((IConvertible) c).ToInt32(null);
      Console.WriteLine(n);                  // Отображает "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 прямо в исходный код. Компилятор помещает эти литеральные строки в метаданные модуля, и они часто загружаются и на них ссылаются во время выполнения.

В C# оператором *new* нельзя создавать объекты *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 него 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, используя 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 C#-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 байт 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 примере выше, 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 строку, необходимую для создания 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 так:

```
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, и в результате в метаданных модуля окажется лишь строка «Hi there.». Конкатенация нелитеральных строк оператором «+» выполняется на этапе выполнения. Для конкатенации нескольких строк в период выполнения оператор «+» применять нежелательно, так как он создает в куче несколько строковых объектов. Вместо него используйте тип *System.Text.StringBuilder* (о нем я расскажу ниже).

И, наконец, в C# есть особый способ 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 примера дают одинаковый результат. Однако символ @ перед строкой во втором случае сообщает компилятору, что перед ним буквальная строка и он должен рассматривать символ «обратный слэш» (\) как таковой, а не как признак управляющей последовательности, благодаря чему путь выглядит привычнее.

Теперь, познакомившись с формированием строки, рассмотрим операции, выполняемые над объектами *String*.

Строки не изменяются

Самое важное, что нужно помнить об объекте *String*, — это то, что он неизменяем; то есть созданную однажды строку нельзя сделать длиннее или короче, в ней нельзя изменить ни одного символа. Наличие постоянных строк имеет определенные преимущества. Во-первых, 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 коде приложения нет ссылок на две временные строки, созданные *ToUpperInvariant* и *SubString*, и их память освободится при очередной сборке мусора. Если выполняется много операций со строками, в куче создается много объектов *String* — это вызывает более частую очистку сборщиком мусора, что отрицательно сказывается на производительности приложения. Если требуется эффективно выполнять много операций со строками, следует использовать класс *StringBuilder*.

Благодаря неизменяемости строк отпадает проблема синхронизации потоков при работе со строками. Кроме того, в CLR возможно нескольким ссылкам *String* 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 ниже.

По соображениям производительности *String* тесно интегрирован с CLR. В частности, CLR «знает» размещение 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 добавить 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 неизменяемости.

Сравнение строк

Сравнение — пожалуй, наиболее часто выполняемая со строками операция. Есть две причины, по которым приходится сравнивать две строки. Мы сравниваем две строки для выяснения, равны ли они, и для сортировки (прежде всего, для представления их пользователю программы).

Для выяснения равенства и сравнения строк при сортировке я настоятельно рекомендую использовать один из перечисленных ниже 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 по-разному упорядочиваться при каждой сортировке, 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 так:

```
public enum StringComparison {
   CurrentCulture = 0,
   CurrentCultureIgnoreCase = 1,
   InvariantCulture = 2,
   InvariantCultureIgnoreCase = 3,
   Ordinal = 4,
   OrdinalIgnoreCase = 5
}
```

Во многих программах строки используют 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 *StringComparison.Ordinal* или *StringComparison.OrdinalIgnoreCase*. Это самый быстрый способ сравнения, так как он игнорирует лингвистические особенности и региональные стандарты.

С другой стороны, если требуется корректно сравнить строки с точки зрения лингвистических особенностей (обычно перед отображением их на экране для пользователя), следует использовать *StringComparison.CurrentCulture* или *StringComparison.CurrentCultureIgnoreCase*.

Внимание! Обычно следует избегать использования *StringComparison.InvariantCulture* и *StringComparisonInvariantCultureIgnoreCase*. Хотя эти значения и позволяют выполнить лингвистически корректное сравнение, использование их для сравнения строк в программе занимает больше времени, чем использование *StringComparison.Ordinal* или *StringComparison.OrdinalIgnoreCase*. Кроме того, игнорирование региональных стандартов — совсем неудачный выбор для сортировки строк, которые планируется показать пользователю.

Внимание! Если вы хотите изменить регистр символов строки перед выполнением простого сравнения, следует использовать предоставленный *String* метод *ToUpperInvariant* или *ToLowerInvariant*. При нормализации строк настоятельно рекомендуется использовать *ToUpperInvariant*, а не *ToLowerInvariant* из-за того, что в Microsoft оптимизировали выполнение сравнений строк в верхнем регистре. На самом деле, в FCL перед не зависящим от регистра сравнением строки нормализуют путем приведения к верхнему регистру.

Иногда для лингвистически корректного сравнения строк используют региональные стандарты, отличные от таковых 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 приведенных 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 определить, 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 определять, 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 в приложениях Windows Forms или Web Forms, так как указывает на язык, который следует использовать для отображения элементов пользовательского интерфейса, таких как метки и кнопки. При извлечении ресурсов используется 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 *CurrentUICulture*, в том числе для форматирования чисел и дат, приведения и сравнения строк. При форматировании используются обе части объекта *CultureInfo* — информация о языке и стране. По умолчанию при создании потока это свойство потока задается Win32-функцией *GetUserDefaultLCID* на основании объекта *CultureInfo*. Это свойство 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 берут из одного *CultureInfo*, то есть в них содержится одинаковая информация о языке и стране. Однако она может отличаться. Например, в приложении, работающем в США, все элементы интерфейса могут отображаться на испанском, а валюта и форматирование даты отображаться 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 со стандартом Unicode. Эти таблицы являются частью самой инфраструктуры .NET Framework, поэтому все версии .NET Framework (независимо от ОС) так же будут сравнивать и сортировать строки.

Значение региональных стандартов при сортировке строк демонстрирует пример:

```
using System;
using System.Globalization;

public static class Program {
   public static void Main() {
      String s1 = "Strasse";
      String s2 = "Straβe";
      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' != 'Straβe'
Cultural comparison: 'Strasse' == 'Straβe'
```

Примечание Если метод *Compare* не выполняет простое сравнение, он выполняет *расширение символов* (character expansions), то есть разбивает сложные символы на несколько символов, игнорируя региональные стандарты. В предыдущем случае немецкий символ эсцет «ß» всегда расширяется до «ss». Аналогично лигатурный символ «Æ» всегда расширяется до «AE». Поэтому в приведенном выше 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 говорилось ранее, объект *CultureInfo* инкапсулирует таблицы сравнения символов для различных региональ-

ных стандартов, причем для каждого регионального стандарта есть только один объект *CompareInfo*.

При вызове метода *Compare* класса *String* используются указанные вызывающим потоком региональные стандарты. Если региональные стандарты не указаны, используется 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, игнорирование 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 битовые флаги, которые можно объединять посредством оператора «или» для получения большего контроля над сравнением срок. Полное описание этих символов см. в документации по .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 = "cô te";

      // 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) на разных вариантах письма: хирагане и катакане.

        s1 = "しんかんせん";  // ("\u3057\u3093\u304b\u3093\u305b\u3093")
        s2 = "シンカンセン";  // ("\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");
    }
}
```

После компоновки и выполнения кода получим результат, показанный на рис. 11-1.

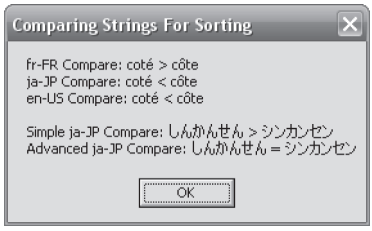


Рис. 11-1. *Результаты сортировки строк*

Японские символы

Чтобы увидеть японские символы в исходном коде и в информационном окне, нужно установить в Windows файлы поддержки восточно-азиатских языков (они занимают на диске около 230 Мб). Для этого откройте на панели управления диалоговое окно Regional And Language Options (Язык и региональные стандарты), выберите вкладку Languages (Языки), пометьте флажок Install Files For East Asian Languages (Установить поддержку языков с письмом иероглифами) и щелкните OK (рис. 11-2) — будут установлены восточно-азиатские шрифты и редакторы способов ввода (Input Method Editor, IME).

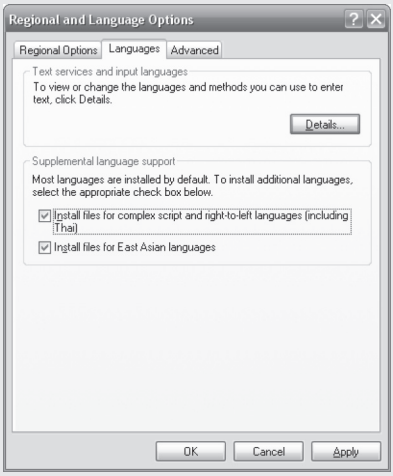


Рис. 11-2. *Установка файлов поддержки восточно-азиатских языков в диалоговом окне панели управления Regional And Language Options (Язык и региональные стандарты)*

Кроме того, подобные файлы с исходным кодом нельзя сохранить в ANSI; я использовал кодировку UTF-8, с которой прекрасно справляются и редактор Microsoft Visual Studio, и компилятор Microsoft C#.

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*. Также следует иметь в виду, что в FCL есть класс *System.StringComparer*, который также можно использовать для сравнения строк. Он оказывается кстати, когда нужно многократно выполнять однотипные сравнения множества строк.

Интернирование строк

Как я уже говорил, сравнение строк встречается во многих приложениях. Но эта операция может ощутимо сказаться на производительности. При выполнении *порядкового сравнения* (ordinal comparison) CLR быстро проверяет, равно ли количе-

ство символов в строках. При отрицательном результате строки точно не равны, но, если длина одинакова, придется кропотливо сравнивать их символ за символом. При сравнении с учетом региональных стандартов CLR обязана посимвольно сравнить строки, потому что две строки разной длины могут оказаться равными.

А 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 все соответствующие ссылки.

Если в приложении строки сравниваются часто с применением порядкового сравнения с учетом регистра или если в нем ожидается появление множества одинаковых строковых объектов, то для повышения производительности надо применить имеющийся в CLR механизм *интернирования строк* (string interning). При инициализации CLR создает внутреннюю хеш-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 хеш-таблице:

```
public static String Intern(String str);
public static String IsInterned(String str);
```

Первый из них, *Intern*, ищет *String* во внутренней хеш-таблице. Если строка есть, возвращается ссылка на соответствующий объект *String*. Иначе создается копия строки, она добавляется во внутреннюю хеш-таблицу, и возвращается ссылка на копию. Если приложение больше не удерживает ссылку на исходный объект *String*, сборщик мусора вправе освободить память, занимаемую этой строкой. Заметьте: сборщик мусора не вправе освободить строки, на которые ссылается внутренняя хеш-таблица, поскольку в ней самой есть ссылки на эти *String*. Объекты *String*, на которые ссылается внутренняя хеш-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 хеш-таблице. Если поиск удачен, *IsInterned* возвращает ссылку на интернированную строку. В противном случае он возвращает *null*, а саму строку не вставляет в хеш-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 выполнения дополнительного поиска в хеш-таблицах, поэтому теперь можно отключить эту «функцию». Если сборка отмечена атрибутом *System.Runtime.CompilerServices.CompilationRelaxationsAttribute*, определяющим значение флага *System.Runtime.CompilerServices.CompilationRelaxations.NoStringInterning*, в соответствии со спецификацией ECMA CLR *может* не интернировать все строки, определенные в метаданных сборки. Заметьте: в целях повышения производительности 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 не написали код, который явно вызывает 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* ссылается на другую объект-строку «Hello». Поскольку ссылки разные, должно отображаться 'False'. Но, 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 CLR интернирует литеральную строку «Hello» при загрузке сборки в домен AppDomain. Это означает, что *s1* и *s2* ссылаются на одну строку в куче. Однако, как уже говорилось, никогда не стоит писать код с расчетом на такое поведение, потому что в последующих версиях этот атрибут/флаг может приниматься во внимание, а строка «Hello» — не интернироваться. В действительности, CLR версии 2.0 учитывает этот атрибут/флаг, но только если код сборки создан с использованием утилиты NGen.exe.

Перед вторым вызовом метода *ReferenceEquals* строка «Hello» явно интернируется, и *s1* теперь ссылается на интернированную строку «Hello». Затем при повторном вызове *Intern 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 можно использовать интернирование строки для повышения производительности и снижения нагрузки на память. Показанный ниже 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;
   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 много идентичных строк может существовать в куче и не подлежать сборке в качестве «мусора».

А теперь посмотрим на версию этого метода, который написан с использованием интернирования 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* будет содержать множественные 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 *NumTimesWordAppearsIntern* из-за времени, которое потребуется на интернирование всех строк по мере добавления их в массив *wordlist* (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 полезно, но использовать его нужно с очень большой 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 росту результирующего файла.

Чтобы не допустить роста объема кода, многие компиляторы (в том числе компилятор C#) хранят литеральную строку в метаданных модуля только в одном экземпляре. Все упоминания этой строки в исходном коде компилятор заменяет ссылками на ее экземпляр в метаданных. Благодаря этому заметно уменьшается размер модуля. Способ не нов — в компиляторах C/C++ этот механизм существует уже давно. В компиляторе Microsoft C/C++ это называется *созданием пула строк* (string pooling). Это еще одно средство, позволяющее ускорить обработку строк. Полагаю, знание о нем может пригодится.

Работа с символами и текстовыми элементами в строке

Сравнение строк полезно при сортировке и поиске одинаковых строк, однако иногда требуется посмотреть отдельные символы в пределах какой-то строки. С подобными задачами должны справляться несколько методов и свойств *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, а 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 далее 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.

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. 11-3, 11-4 и 11-5).

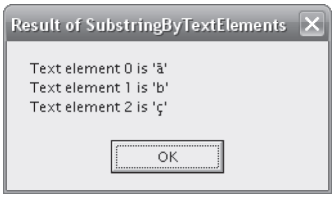


Рис. 11-3. *Результат работы SubstringByTextElements*

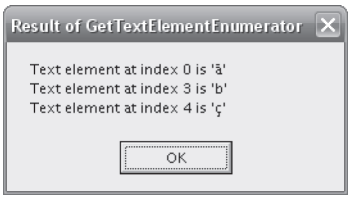


Рис. 11-4. *Результат работы GetTextElementEnumerator*

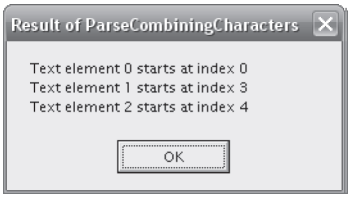


Рис. 11-5. *Результат работы ParseCombiningCharacters*

Примечание Чтобы информационное окно выглядело так, как на рис. 11-3 и 11-4, мне пришлось в диалоговом окне Свойства: Экран (Windows Display Properties) на вкладке Оформление (Appearance) щелкнуть кнопку Дополнительно (Advanced) и выбрать шрифт Lucida Sans Unicode в качестве шрифта для окон сообщений, так как в нем есть глифы для данных комбинированных символов. По той же причине мне не удалось вывести результаты работы кода на консоль.

Прочие операции со строками

Следующие методы типа *String* предназначены для полного или частичного копирования строк (табл. 11-1).

Табл. 11-1. Методы для копирования строк

Член	Тип метода	Описание
Clone	Экземплярный	Возвращает ссылку на тот же самый объект (*this*). Это правильно, так как объекты *String* неизменяемы. Этот метод реализует интерфейс *Icloneable* класса *String*
Copy	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
CopyTo	Экземплярный	Копирует группу символов строки в массив символов
SubString	Экземплярный	Возвращает новую строку, представляющую часть исходной строки
ToString	Экземплярный	Возвращает ссылку на тот же объект (*this*)

Помимо этих методов, у *String* много статических и экземплярных методов для манипуляций со строкой, таких как *Insert*, *Remove*, *PadLeft*, *Replace*, *Split*, *Join*, *ToLower*, *ToUpper*, *Trim*, *Concat*, *Format* и пр. Еще раз: все эти методы возвращают новые строковые объекты; создать строку можно, но изменить ее нельзя (при условии использования безопасного кода).

Эффективное создание строки динамически

Тип *String* представляет постоянную строку, однако 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 с 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. Прежний массив станет добычей 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 *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 2 млрд). 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, и 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*. *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 *StringBuilder*

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, превышающего установленную емкость;
- при попытке изменить массив, после того как был вызван метод *ToString* типа *StringBuilder*.

В интересах повышения производительности методы *StringBuilder* не обеспечивают безопасной работы с несколькими потоками. Пока потоки не обращаются к одному объекту *StringBuilder*, проблем обычно не возникает. Если же приложению, работающему с несколькими потоками, нужны безопасные манипуляции с объектом *StringBuilder*, добавьте свой код синхронизации потоков.

В табл. 11-2 представлены методы класса *StringBuilder*.

Табл. 11-2. Члены *StringBuilder*

Член	Тип члена	Описание
MaxCapacity	Неизменяемое свойство	Возвращает наибольшее количество символов, которое может быть размещено в строке
Capacity	Изменяемое свойство	Получает/устанавливает размер массива символов. При попытке установить емкость меньшую, чем длина строки, или большее, чем *MaxCapacity*, генерируется исключение *ArgumentOutOfRangeException*

Табл. 11-2. *(окончание)*

Член	Тип члена	Описание
EnsureCapacity	Метод	Гарантирует, что размер массива символов будет не меньше, чем значение параметра, передаваемого этому методу. Если значение превышает текущую емкость объекта *StringBuilder*, размер массива увеличивается. Если текущая емкость больше, чем значение, передаваемое этому свойству, размер массива не изменяется
Length	Изменяемое свойство	Возвращает количество символов в «строке». Эта величина может быть меньше текущей емкости массива символов. Присвоение этому свойству значения 0 сбрасывает содержимое и очищает строку *StringBuilder*
ToString	Метод	Версия без параметров возвращает объект *String*, представляющий поле с массивом символов объекта *StringBuilder*. Поскольку новый объект String не создается, метод работает быстро. Любая попытка модифицировать массив *StringBuilder* приведет к выделению памяти под новый массив (заполняемый значениями из старого массива). Метод *ToString* с параметрами *startIndex* и *length* создает новый объект *String*, в котором представлена указанная параметрами часть *StringBuilder*
Chars	Изменяемое свойство-индексатор	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 [])
Append и *Insert*	Методы	Добавляют/вставляют 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
AppendFormat	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* возвращает 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*, *EndsWiths*, *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 мы должны прибегать иногда к преобразованиям между *String* и *StringBuilder*. Так, сформировать строку, сделать все буквы прописными, а затем вставить в нее другую строку позволяет примерно такой 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 назад 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, которое требуется, как правило, для отображения числового типа (такого как *Byte*, *Int32*, *Single* и т. д.) и объекта *DateTime*. Поскольку .NET Framework является объектно-ориентирован-

ной платформой, то каждый тип должен сам обеспечить код, преобразующий «значение» экземпляра в некий строковый эквивалент. Выбирая способы решения этой задачи, разработчики FCL придумали шаблон, предназначенный для повсеместного использования. В этом разделе я опишу этот шаблон.

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 вызвать 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 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.

Форматы и региональные стандарты

Метод *ToString* без параметров создает две проблемы. 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 приложений на стороне сервера и менее актуальна для кода на стороне клиента. Изредка приложению требуется форматировать строку с учетом региональных стандартов, отличных от аналогичных параметров 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*. Кроме того, 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, *formatProvi-*

der, — 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 сгенерировать исключение *System.FormatException*.

Многие типы FCL различают несколько строк форматирования. Так, тип *DateTime* поддерживает: строку «d» — для дат в кратком формате, «D» — для дат в полном формате, «g» — для дат в общем формате, «M» — в формате «месяц/день», «s» — для сортируемых дат, «T» — для времени, «u» — для универсального времени в стандарте ISO 8601, «U» — для универсального времени в полном формате, «Y» — для формата «год/месяц» и т. д. Все перечислимые типы поддерживают: «G» как общий формат, «F» — для флагов, «D» — для десятичных и «X» — для шестнадцатеричных чисел. Подробнее о форматировании перечислимых типов см. главу 12.

Кроме того, все встроенные числовые типы поддерживают: «C» — для валют, «D» — для десятичных, «E» — для научных (экспоненциальных), «F» — для чисел с фиксированной точкой, «G» — общий формат, «N» — для чисел, «P» — для процентов, «R» — для обратимого (round-trip) формата и «X» — для шестнадцатеричных чисел. 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* с *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*, *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*. *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, где 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. 11-6).

Рис. 11-6. *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 сведения только о форматировании чисел. Однако другие типы (вроде *DateTime*) могут вызвать *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 мо-

гут запросить 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. 11-7). Обратите внимание на первый символ в выходной строке: ¤. Он представляет международное обозначение денежного знака (U+00A4).

Рис. 11-7. *Числовое значение в формате, представляющем абстрактную денежную единицу*

Обычно нет необходимости отображать 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: уже упоминавшегося *CultureInfo*, а также *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.

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 добавлены перегруженные версии метода *ToString*. Так, 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.
// В этой версии 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).
// Эта версия реализует 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 отдельным типом своих объектов. Однако иногда нужно формировать строки из множества форматированных объектов. В следующем примере в строку включается дата, имя человека и его возраст:

```
String s = String.Format("On {0}, {1} is {2} years old.",
   new DateTime(2006, 4, 22, 14, 35, 5), "Aidan", 3);
Console.WriteLine(s);
```

Если собрать и запустить этот код в потоке с региональным стандартом «en-US», на выходе получится строка:

```
On 4/22/2006 2:35:05 PM, Aidan is 3 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 (текущую дату и время), вместо «{1}» — следующий параметр («Aidan») и вместо «{2}» — третий, последний параметр (3).

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 формата. В частности, следующий код отличается от предыдущего только наличием строк формата для подставляемых параметров 0 и 2:

```
String s = String.Format("On {0:D}, {1} is {2:E} years old.",
   new DateTime(2006, 4, 22, 14, 35, 5), "Aidan", 3);
Console.WriteLine(s);
```

Если собрать и запустить этот код в потоке с региональным стандартом «en-US», на выходе вы увидите строку:

```
On Saturday, April 22, 2006, Aidan is 3.000000E+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 «D» 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 усложняет 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 .NET Framework предлагает весьма гибкие средства форматирования. Но это не все. Вы также 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 AppendFormat* вызовет вашу функцию, которая будет форматировать один или несколько объектов так, как вам нужно. Описанное ниже также применимо к методу *Format* типа *String*.

Попробую пояснить описанный механизм на примере. Допустим, вам нужен форматированный HTML-текст, который пользователь будет просматривать в Интернет-браузере, и надо отображать все значения *Int32* полужирным шрифтом. Для этого всякий раз, когда значение типа *Int32* форматируется в *String*, нужно обрамлять строку тегами полужирного шрифта: *<B>* и *</B>*. Вот как это сделать:

```
using System;
using System.Text;
using System.Threading;

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 выше 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. Взглянем 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.
            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 *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 форматируемый 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 поставщик формата.

Теперь, имея форматированную строку, я проверяю, является ли объект типом *Int32*, и, если да, обрамляю строку HTML-тегами *<B>* и *</B>* и возвращаю полученную строку. Если объект не является *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. Получение объекта из строки встречается нечасто, однако иногда оно может оказаться полезным. В Microsoft осознали важность формализации механизма, посредством которого строки можно разобрать на объекты.

Любой тип, способный анализировать строку, 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*. Здесь я покажу вам только метод *Parse*, описанный 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 вы хотите получить путем анализа объекта *Int32*. Параметр *style* типа *System.Globalization.NumberStyles* — это набор двоичных флагов для идентификации символов, которые *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 выше.

Ниже при обращении к *Parse* генерируется 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 к .NET Framework SDK.

Вот пример 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);

// 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 рассказал 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 перечислимым типом *System.Globalization.DateTimeStyles*, а не типом *NumberStyles*. Подробнее о двоичных символах и стандартных комбинациях, определенных в типе *DateTimeStyles*, см. документацию к .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 разбору. Многие разработчики столкнулись с тем, что метод *Parse* типа *DateTime* позволяет 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 анализ.

О шаблонах форматирования см. раздел, посвященный *DateTimeFormatInfo*, в документации .NET Framework SDK.

Примечание Некоторые разработчики сообщили в Microsoft о следующем факте: при частом вызове *Parse* этот метод часто генерирует исключения (из-за неверных данных, вводимых пользователями), а это отрицательно сказывается на производительности приложения. Для таких, требующих высокой производительности, случаев использования *Parse* в Microsoft создали методы *TryParse* для всех числовых типов данных, *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 как *Int32*. Если метод возвращает true, переменная, переданная по ссылке в результирующем параметре, будет содержать полученное в результате анализа числовое значение. Шаблон *Try*Xxx обсуждается в главе 19.

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 чревато ошибками. В CLR все символы представлены 16-разрядными Unicode-значениями, а строки состоят только из 16-разрядных Unicode-символов. Это намного упростило работу с символами и строками в период выполнения.

Однако порой какой-то 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 байт, 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 (кодовая страница 932). Аналогично с помощью кода Shift-JIS можно прочитать в CLR текстовый файл, созданный в японской версии Windows.

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*. При отсутствии явного указания кодировки 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, а 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 байт (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 1 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 байт. 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 редко используется для сохранения или передачи строк в файл или по сети. Эта кодировка обычно используется внутри программ. Стоит также заметить, что UTF-32 можно использовать для преобразования прямого порядка байт 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 не рекомендует использовать UTF-7.
- **ASCII** кодирует 16-разрядные символы в символы ASCII; 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 и 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 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 байт.
      Byte[] encodedBytes = encodingUTF8.GetBytes(s);

      // Показываем значение закодированных байт.
      Console.WriteLine("Encoded bytes: " +
         BitConverter.ToString(encodedBytes));

      // Выполняем декодирование массива байт 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 Панели управления Regional and Language Options (Язык и региональные стандарты). (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 по умолчанию или выполнении приложения на другой машине приложение будет вести себя иначе.

Наряду с перечисленными свойствами у *Encoding* есть статический метод *GetEncoding*, позволяющий указать кодовую страницу (в виде числа или строки). *GetEncoding* возвращает объект, выполняющий кодирование/декодирование, ис-

пользуя 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, а 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 дополнительное управление процессом кодирования и маркерами последовательности байт (byte order mark, BOM). Первые три класса также имеют конструкторы, которые позволяют заставить класс генерировать исключение при декодировании некорректной последовательности байт; эти конструкторы нужно использовать для обеспечения безопасности приложения и устойчивости к приему 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 через запрос 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 байт. (У этого метода есть несколько перегруженных версий.) Для обратного преобразования вызовите метод *GetChars* или более удобный *GetString*. (Эти методы также имеют несколько перегруженных версий.) В приведенном выше примере продемонстрирована работа методов *GetBytes* и *GetString*.

Кстати, у всех типов, производных от *Encoding*, есть метод *GetByteCount*, который, не выполняя реального кодирования, подсчитывает количество байт, необходимых для кодирования данного набора символов. Он может пригодиться, когда нужно выделить память под массив байт. Имеется также аналогичный метод *GetCharCount*, который возвращает число подлежащих декодированию символов, не выполняя реального декодирования. Эти методы полезны, когда требуется сэкономить память и повторно использовать массив.

Методы *GetByteCount* и *GetCharCount* работают не так быстро, поскольку для получения точного результата они должны анализировать массив символов/байт. Если скорость важнее точности, вызовите *GetMaxByteCount* или *GetMaxCharCount* — оба метода принимают целое число, в котором задается число символов или байт 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 .NET Framework SDK.

Чтобы проиллюстрировать свойства и их назначение, я написал программу, в которой эти свойства вызываются для нескольких способов 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
        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. 11-3.

Табл. 11-3. Методы классов, производных от *Encoding*

Метод	Описание
GetPreamble	Возвращает массив байт, показывающих, что нужно записать в поток перед записью кодированных байт. Часто такие байты называют *BOM-байтами* (byte order mark) или *заголовком* (preamble). Когда вы приступаете к чтению из потока, BOM 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 заголовка. 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 байт (big endian) или 0xFF, 0xFE — для обратного (little endian)
Convert	Преобразует массив байт 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 байт возвращается вызывающей программе
Equals	Возвращает true, если два производных от *Encoding* объекта представляют одну кодовую страницу и одинаковый заголовок
GetHashCode	Возвращает кодовую страницу объекта кодирования

Кодирование и декодирование потоков символов и байт

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 2 байт. Поэтому в результате вызова метода *GetString* класса *Encoding* с передачей первого массива из 5 байт будет возвращена строка, содержащая только 2 символа. При следующем вызове *GetString* из потока поступят следующие 7 байт, и *GetString* вернет строку, содержащую 3 символа, причем все неверные!

Причина искажения данных 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 байт, поступающих порциями, то вам придется приложить дополнительные усилия на отслеживание 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*. *Decoder*, подобно классу *Encoding*, является абстрактным базовым классом. В документации .NET Framework SDK вы не увидите, что некоторые классы представляют собой конкретные реализации класса *Decoder*. Однако 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 есть два метода: *GetChars* и *GetCharCount*. Они, конечно, применяются для декодирования и работают аналогично рассмотренным выше методам *GetChars* и *GetCharCount* класса *Encoding*. Когда вы вызываете один из этих методов, он декодирует байтовый массив, насколько это возможно. Если в массиве не хватает байт для формирования символа, оставшиеся байты сохраняются внутри объекта декодирования. При следующем вызове одного из этих методов объект декодирования берет оставшиеся байты и складывает их с вновь полученным байтовым массивом — благодаря этому декодирование данных, поступающих порциями, выполняется корректно. Объекты *Decoder* весьма удобны для чтения байт из потока.

Тип, производный от *Encoding*, может быть использован 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* используйте метод *GetEncoder*. Он возвращает вновь созданный объект, производный от класса *System.Text.Encoder*, который также является абстрактным базовым классом. И опять, в документации .NET Framework SDK нет описания классов, представляющих собой конкретную реализацию класса *Encoder*. Однако 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. Это связано также с возможностью кодировать последовательности байт 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 байтовый массив, вызовите статический метод *FromBase64String* или *FromBase64CharArray* класса *Convert*. Аналогично для декодирования байтового массива в строку base-64 используется 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
O7knQFk1hlRf8Q==
3B-B9-27-40-59-35-86-54-5F-F1
```

Защищенные строки

Часто объекты *String* используются 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 не-

благовидных 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 с ними старые версии «висят» в памяти, в результате разные версии строки остаются в различных областях памяти.

В некоторых государственных учреждениях действуют строгие требования по безопасности, гарантирующие определенный уровень защиты. Для решения таких задач Microsoft добавила 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*. Всякий раз 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 все-таки желательно пореже.

Класс *SecureString* реализует интерфейс *IDisposable*, служащий для надежного уничтожения конфиденциальной информации, хранимой в строке. Когда приложению больше не нужно хранить конфиденциальную строковую информации, 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 классу *CriticalFinalizerObject* (см. главу 20), 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 не остаются в памяти.

Узнав, как создавать и изменять объект *SecureString*, пора поговорить о его использовании. К сожалению, в последней версии FCL поддержка класса *SecureString* ограничена. Иначе говоря, 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*.

В будущем в .NET Framework планируется обеспечить более качественную поддержку *SecureString*. В частности, ожидается поддержка *SecureString* текстовых полей в Windows Forms, Windows Presentation Foundation и приложениях Web Form, что позволит конечным пользователям вводить безопасную строковую информацию в пользовательском интерфейсе приложения. Также ожидается, что *SecureString* будет передаваться как строка для подключения к базам данных или создания сетевого подключения. Есть много других ситуаций в FCL, в которых *SecureString* была бы очень кстати.

Наконец, вы вправе создавать собственные методы, принимающие в качестве аргумента объект *SecureString*. В методе надо использовать объект *SecureString* для создания буфера неуправляемой памяти, хранящего расшифрованные символы, до использования этого буфера в методе. Чтобы сократить до минимума временное «окно» возможности 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 в *SecureString* в *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* преднамеренно — 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 будет определить 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.Write("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.Write("*");
         }
         Console.WriteLine();

         // Пароль введен, отобразим его для целей демонстрации.
         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 обязательно вызывать для обнуления и освобождения внутреннего буфера. В табл. 11-4 приведены методы класса *System.Runtime.InteropServices.Marshal*, используемые для расшифровки *SecureString* в буфер неуправляемой памяти, а также связанные методы для обнуления и освобождения буфера.

Табл. 11-4. Методы класса *Marshal* для работы с защищенными строками

Метод для расшифровки SecureString в буфер	Метод для обнуления и освобождения буфера
SecureStringToBSTR	*ZeroFreeBSTR*
SecureStringToCoTaskMemAnsi	*ZeroFreeCoTaskMemAnsi*
SecureStringToCoTaskMemUnicode	*ZeroFreeCoTaskMemUnicode*
SecureStringToGlobalAllocAnsi	*ZeroFreeGlobalAllocAnsi*
SecureStringToGlobalAllocUnicode	*ZeroFreeGlobalAllocUnicode*

ГЛАВА 12

Перечислимые типы и битовые флаги

Перечислимые типы и битовые флаги существуют в Windows долгие годы, поэтому я уверен, что многие из вас уже знакомы с их применением. Но по-настоящему объектно-ориентированными перечислимые типы и битовые флаги становятся в общеязыковой 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 пойдет разговор в этой главе, можно облегчить разработку приложений.

Перечислимые типы

Перечислимым (enumerated type) называют тип, в котором описан набор пар, состоящих из символьного имени и значения. Ниже 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 задать 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 «зашитого» в коде значения (что White равен 0, а 0 означает White). Если же числовое значение символа почему-либо изменилось, то нужно только перекомпи-

лировать исходный код, не изменив 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), а 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 возлагаются очень серьезные задачи, просто немыслимые для перечислимых типов в других средах (например, в неуправляемом C++).

Каждый перечислимый тип прямо наследует *System.Enum*, производному от *System.ValueType*, а тот в свою очередь — *System.Object*. Из чего 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 не может быть методов, свойств и событий.

При компиляции перечислимого типа компилятор C# превращает каждый идентификатор в константное поле типа. Например, описанное выше перечисление *Color* компилятор рассматривает примерно так, как если бы это был такой код:

```
internal struct Color : System.Enum {
   // Ниже 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;

   // Ниже приведено открытое экземплярное поле,
   // содержащее значение переменной Color.
   // Невозможно написать код, ссылающийся на этот экземпляр напрямую.
   public Int32 value__;
}
```

В действительности, компилятор C# не будет компилировать такой код, потому что не разрешает определять типы, производные от специального типа *System.Enum*. Зато на этом примере видно, что там внутри. В общем-то, перечислимый тип — это обычная структура, внутри которой описан набор константных полей и одно экземплярное поле. Константные поля попадают в метаданные модуля, откуда их можно извлечь с помощью механизма отражения. Это означает, что можно получить все идентификаторы и их значения, связанные перечислимым типом, в период выполнения, 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 возиться с отражением. Вскоре мы посмотрим, что это за операции.

Внимание! Идентификаторы, описанные перечислимым типом, являются константами, то есть компилятор уже на этапе компиляции преобразует ссылку на идентификатор перечислимого типа в числовое значение. А раз так, то метаданные не содержат ссылку на такой перечислимый тип, и сборка, описывающая перечислимый тип, становится не нужна в период выполнения. Если в коде есть ссылки на перечислимый тип, — а не просто ссылки на идентификаторы, описанные в этом типе, — сборка, где описан этот тип, будет затребована в период выполнения. Здесь возникают проблемы, связанные с управлением версиями, поскольку идентификаторы перечислимого типа — это не значения «только для чтения», а константы. Эти вопросы подробно освещены в главе 8.

Например, у типа *Enum* есть статический метод *GetUnderlyingType*:

```
public static Type GetUnderlyingType(Type enumType);
```

Он возвращает базовый тип, используемый для хранения значения перечислимого типа. В основе каждого перечислимого типа лежит некий исходный тип, в качестве которого могут быть *byte*, *sbyte*, *short*, *ushort*, *int* (наиболее популярный и поэтому используемый в C# по умолчанию), *uint*, *long* и *ulong*. Конечно, у этих элементарных типов C# есть аналоги в FCL. Однако компилятор C# пропустит только элементарный тип; задание базового класса FCL (такого как *Int32*) вызовет ошибку: «error CS1008: Type byte, sbyte, short, ushort, int, uint, long, or ulong expected» («ошибка CS1008: Ожидается тип byte, sbyte, short, ushort, int, uint, long или ulong»). Вот правильное объявление в C# перечислимого типа, основанного на типе *byte* (*System.Byte*):

```
internal enum Color : byte {
   White,
   Red,
   Green,
   Blue,
   Orange
}
```

Если перечислимый тип *Color* описан подобным образом, то с помощью следующего кода мы увидим, что возвращает *GetUnderlyingType*:

```
// Следующая строка выводит "System.Byte".
Console.WriteLine(Enum.GetUnderlyingType(typeof(Color)));
```

Компилятор C# считает перечислимые типы элементарными. Поэтому для операций с экземплярами перечислимых типов можно использовать большинство знакомых операторов («==», «!=», «<», «>», «<=», «>=», «+», «–», «^», «&», «|», «~», «++» и «––»). Вообще-то, все эти операторы действуют на экземплярное поле *value__* экземпляра перечислимого типа. Более того, компилятор C# позволяет явно приводить экземпляры одного перечислимого типа в другой. Также поддерживается неявное приведение перечислимых типов в числовой тип.

Если есть экземпляр перечислимого типа, можно получить его значение в виде одной или более строк, вызвав 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 использовании шестнадцатеричного формата *ToString* всегда возвращает прописные буквы, а число возвращенных цифр зависит от типа, лежащего в основе перечислимого типа: *byte/sbyte* — 2 цифры, *short/ushort* — 4 цифры, *int/uint* — 8 цифр и *long/ulong* — 2 цифры. 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);
```

Вообще я предпочитаю вызывать метод *ToString* — он требует меньше кода и с ним проще работать. Но у *Format* есть одно преимущество перед *ToString*: ему можно передавать в качестве параметра *value* числовое значение, а иметь экземпляр перечислимого типа при этом не обязательно. Например, этот код выведет строку «Blue»:

```
// Следующая строка выводит "Blue".
Console.WriteLine(Enum.Format(typeof(Color), 2, "G"))
```

Примечание В объявлении перечислимого типа несколько идентификаторов могут иметь одинаковое числовое значение. При преобразовании числового значения в символьный идентификатор методы типа *Enum* возвращают один из этих символьных идентификаторов, при этом заранее неизвестно, какой идентификатор вернет метод. Кроме того, если для указанного числового значения идентификатора нет, мы получим строку, содержащую только числовое значение.

Можно также вызовом *GetValues* типа *System.Enum* создать массив, содержащий по одному элементу на каждое символическое имя перечислимого типа; при этом каждый элемент будет содержать числовое значение, соответствующее символическому имени:

```
public static Array GetValues(Type enumType);
```

Методы *GetValues* и *ToString* позволяют показать все символические имена и числовые значения перечислимого типа:

```
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 формате.
   Console.WriteLine("{0,5:D}\t{0:G}", c);
}
```

Результат выполнения этого кода выглядит так:

```
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 отображать 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 (увы, перечислимые типы не поддерживают локализацию). Помимо метода *GetValues*, у типа *Enum* есть еще два статических метода, возвращающих идентификаторы перечислимого типа:

```
// Возвращает строковое представление числового значения.
public static String   GetName(Type enumType, Object value);

// Возвращает массив строк: по одной строке на каждое символьное имя,
// описанное в перечислении.
public static String[] GetNames(Type enumType);
```

Я показал несколько методов, позволяющих найти символьное имя, или идентификатор перечислимого типа. Но еще необходим метод для определения значения, соответствующего идентификатору, например, когда пользователь вводит его в текстовое поле. Преобразование идентификатора в экземпляр перечислимого типа легко реализуется статическим методом *Parse* типа *Enum*:

```
public static Object Parse(Type enumType, String value);
public static Object Parse(Type enumType,
   String value, Boolean ignoreCase);
```

Вот пример использования этого метода:

```
// Поскольку Orange определен как 4, 'c' инициализируется значением 4.
Color c = (Color) Enum.Parse(typeof(Color), "orange", true);

// Поскольку Brown не определен, генерируется исключение ArgumentException.
Color c = (Color) Enum.Parse(typeof(Color), "Brown", false);

// Создает экземпляр перечисления Color со значением 1.
Color c = (Color) Enum.Parse(typeof(Color), "1", false);

// Создает экземпляр перечисления Color со значением 23.
Color c = (Color) Enum.Parse(typeof(Color), "23", false);
```

Наконец, статический метод *IsDefined* типа *Enum*:

```
public static Boolean IsDefined(Type enumType, Object value);
```

позволяет определить допустимость числового значения для данного перечислимого типа:

```
// Отображает "True", поскольку Red определен в Color как 1.
Console.WriteLine(Enum.IsDefined(typeof(Color),  1));

// Отображает "True", поскольку White определен в Color как 0.
Console.WriteLine(Enum.IsDefined(typeof(Color), "White"));

// Отображает "False", поскольку выполняется проверка с учетом регистра.
Console.WriteLine(Enum.IsDefined(typeof(Color), "white"));

// Отображает "False" поскольку в Color нет идентификатора со значением 10.
Console.WriteLine(Enum.IsDefined(typeof(Color),  10));
```

Часто с помощью метода *IsDefined* выполняется проверка параметра. Например:

```
public void SetColor(Color c) {
   if (!Enum.IsDefined(typeof(Color), c)) {
      throw(new ArgumentOutOfRangeException("c", c, "Invalid Color value."));
   }
   // Установить цвет в White, Red, Green, Blue или Orange.
   ...
}
```

Проверять параметр нелишне — вдруг кто-то обратится к *SetColor* так:

```
SetColor((Color) 547);
```

Так как идентификатора, соответствующего величине 547, в описании нет, метод *SetColor* вызовет исключение *ArgumentOutOfRangeException*, и мы увидим, какой параметр некорректен и почему.

Внимание! Метод *IsDefined* очень удобен, но применять его надо осторожно. По-первых, *IsDefined* всегда выполняет поиск с учетом регистра, и никак нельзя заставить его игнорировать регистр. Во-вторых, *IsDefined* работает довольно медленно, поскольку в нем используется отражение; если самостоятельно написать код, проверяющий все возможные значения, производительность приложения наверняка возрастет. В-третьих, *IsDefined* нужно использовать только в перечислимых типах, определенных в той же сборке, что вызывает этот метод. И вот почему. Допустим, перечисление *Color* определено в одной сборке, а метод *SetColor* — в другой. *SetColor* вызывает *IsDefined* и выполняет свою работу, только если цвет равен *White*, *Red*, *Green*, *Blue* или *Orange*. Однако, если в будущем в перечисление *Color* добавится цвет *Purple*, *SetColor* будет использовать неизвестный ему цвет *Purple*, но при этом результат работы метода предсказать невозможно.

Наконец, у типа *System.Enum* имеется набор статических методов *ToObject*, преобразующих экземпляр типа *Byte*, *SByte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Int64* или *UInt64* в экземпляр 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

Нам часто приходится иметь дело с наборами битовых флагов. При вызове метода *GetAttributes* типа *SystemIO.File* он возвращает экземпляр типа *FileAttributes*. Тип *FileAttributes* является экземпляром перечислимого типа, основанного на *Int32*, где каждый разряд соответствует какому-то атрибуту файла. В FCL тип *FileAttributes* описан так:

```
[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 = @"C:\Boot.ini";
FileAttributes attributes = File.GetAttributes(file);
Console.WriteLine("Is {0} hidden? {1}", file,
   (attributes & FileAttributes.Hidden) == FileAttributes.Hidden);
```

А вот код для установки атрибутов файла «только для чтения» и «скрытый»:

```
File.SetAttributes(@"C:\Boot.ini",
   FileAttributes.ReadOnly | FileAttributes.Hidden);
```

Из описания типа *FileAttributes* видно, что, как правило, при создании набора битовых флагов, которые могут комбинироваться друг с другом, используют перечислимые типы. Но, несмотря на схожесть перечислимых типов и битовых флагов, семантически они различаются. Например, перечислимые типы представ-

ляют отдельные числовые значения, а битовые флаги представляют набор флагов, одни из которых установлены, а другие нет.

При описании перечислимого типа, предназначенного для идентификации битовых флагов, каждому идентификатору следует явно присвоить числовое значение, отображаемое в определенные разряды. Обычно в соответствующем идентификатору значении установлен лишь один бит. Также часто приходится видеть идентификатор *None*, значение которого определено как 0, а еще можно определить идентификаторы, представляющие часто используемые комбинации (см. приведенный ниже символ *ReadWrite*). Настоятельно рекомендуется применять к перечислимому типу специализированный атрибут типа *System.FlagsAttribute*:

```
[Flags]    // Компилятор C# принимает либо "Flags", либо "FlagsAttribute".
internal enum Actions {
   None      = 0
   Read      = 0x0001,
   Write     = 0x0002,
   ReadWrite = Actions.Read | Actions.Write,
   Delete    = 0x0004,
   Query     = 0x0008,
   Sync      = 0x0010
}
```

Actions является перечислимым типом, поэтому при работе с ним можно воспользоваться всеми описанными в предыдущем разделе методами. Но желательно, чтобы некоторые из этих функций вели себя несколько иначе. Допустим, есть такой код:

```
Actions actions = Actions.Read | Actions.Delete;     // 0x0005.
Console.WriteLine(actions.ToString());               // "Read, Delete".
```

Здесь *ToString* пытается перевести числовое значение в его символьный эквивалент. Но у числового значения 0x0005 нет символьного эквивалента. Однако, обнаружив у типа *Actions* наличие атрибута *[Flags]*, метод *ToString* рассматривает числовое значение уже не как единичное значение, а как набор битовых флагов. И раз установлены биты 0x0001 и 0x0005, то *ToString* формирует строку «Read, Delete». Если в описании типа *Actions* убрать атрибут *[Flags]*, *ToString* вернет строку «5».

В предыдущем разделе мы рассмотрели метод *ToString* и привели три способа форматирования выходной строки: «G» (общий), «D» (десятичный) и «X» (шестнадцатеричный). Выполняя форматирование экземпляра перечислимого типа при помощи общего формата, метод сначала определяет наличие у типа атрибута *[Flags]*. Если атрибут не указан, отыскивается и возвращается идентификатор, соответствующий данному числовому значению. Если атрибут *[Flags]* есть, *ToString* действует по следующему алгоритму.

1. Получает набор числовых значений, определенных в перечислимом типе, и сортирует их в нисходящем порядке.
2. Каждое численное значение умножается с применением оператора «и» (AND) на экземпляр перечисления, и, если результат равен численному значению, строка, связанная с численным значением, добавляется в конец выходной строки, а соответствующие биты считаются учтенными и сбрасываются. Эта опе-

рация повторяется, пока не проверены все численные значения или не сброшены все биты экземпляра перечисления.

3. Если после проверки всех численных значений экземпляр перечисления не равен нулю, значит нет идентификаторов, сопоставленных оставшимся несброшенным битам. В этом случае *ToString* 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 исходное значение экземпляра перечисления было равно 0 и в перечислимом типе есть идентификатор со значением 0, возвращается этот идентификатор.
6. При достижении этого этапа возвращается «0».

Тип *Actions* можно определить и без атрибута *[Flags]*, получив при этом правильную результирующую строку. Но для этого придется 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 ни один 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 по имени *All* со значением 0x001F. При форматировании экземпляра типа *Actions* со значением 0x001F будет получена строка «All». Других идентификаторов в строке не будет.

Пока мы говорили лишь о преобразовании числовых значений в строку флагов. Но из строки, содержащей разделенные запятой идентификаторы, можно получить ее числовое значение, используя статический метод *Parse* типа *Enum*. Вот пример использования этого метода:

```
// Поскольку Query описан как 8, 'a' инициализируется значением 8.
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query", true);
Console.WriteLine(a.ToString());  // "Query".

// Поскольку описаны и Query, и Read, 'a' инициализируется значением 9.
a = (Actions) Enum.Parse(typeof(Actions), "Query, Read", false);
Console.WriteLine(a.ToString());  // "Read, Query".
```

```
// Создаем экземпляр перечисления Actions со значением 28.
a = (Actions) Enum.Parse(typeof(Actions), "28", false);
Console.WriteLine(a.ToString());  // "Delete, Query, Sync".
```

Внутренний алгоритм работы метода *Parse* таков:

1. Удаляются все пробелы в начале и конце строки.
2. Если первый символ строки — цифра и знак «плюс» (+) или «минус» (–), строка считается числом и *Parse* возвращает экземпляр перечисления, числовое значение которого равно числовому эквиваленту, полученному в результате преобразования строки.
3. Переданная строка разбивается на лексемы (отделенные запятыми), и у каждой удаляются все пробелы в начале и конце.
4. Выполняется поиск каждой строки лексемы среди определенных в перечислимом типе символов. Если символ найти не удается, генерируется исключение *SystemArgumentException*. Если символ находится, его числовое значение присоединяется к результирующему значению оператором «или» (OR), после чего анализируется следующая лексема.
5. После того как проанализированы и найдены все лексемы, результат возвращается вызывающей программе.

Никогда не следует применять метод *IsDefined* по отношению к перечислимым типам битовых флагов. Это не будет работать по двум причинам:

- получив строку, *IsDefined* не станет разбивать ее на отдельные лексемы и анализировать их, а выполнит поиск строки целиком, вместе с запятыми. Поскольку в перечислимом типе нельзя определить идентификатор, содержащий запятые, *IsDefined* никогда ничего не найдет;
- получив числовое значение, *IsDefined* попытается найти лишь один символ перечислимого типа, значение которого равно переданному числу. Для битовых флагов вероятность получения положительного результата при таком сравнении ничтожно мала, *IsDefined* обычно возвращает false.

ГЛАВА 13

Массивы

Массивы — это механизм объединения отдельных элементов в набор, рассматриваемый как единое целое. Общеязыковая исполняющая среда Microsoft .NET (CLR) поддерживает массивы *одномерные* (single-dimension), *многомерные* (multidimension) и *вложенные* (jagged). 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. Поясню это на примере:

```
Int32[] myIntegers;           // Объявляем ссылку на массив.
myIntegers = new Int32[100];  // Создаем массив из 100 элементов типа Int32.
```

В первой строке объявляется переменная *myIntegers* для хранения ссылки на одномерный массив элементов типа *Int32*. Вначале *myIntegers* присваивается *null*, так как память под массив пока не выделена. Во второй строке выделяется память под 100 значений типа *Int32*; все они инициализируются 0. Поскольку массивы относятся к ссылочным типам, блок памяти для хранения 100 неупакованных экземпляров *Int32* выделяется в управляемой куче. Вообще говоря, помимо элементов массива в блоке памяти также размещается указатель на объект-тип и *SyncBlockIndex*, а также некоторые дополнительные члены. Возвращенный адрес этого блока памяти заносится в переменную *myIntegers*.

Аналогично создается и массив элементов ссылочного типа:

```
Control[] myControls;         // Объявляем ссылку на массив.
myControls = new Control[50]; // Создаем массив из 50 ссылок на Control.
```

В первой строке объявлена переменная *myControls* для размещения ссылки на одномерный массив ссылок на элементы *Control*. Вначале *myControls* присваивается *null*, ведь память под массив не выделена. Во второй строке выделяется память под 50 ссылок на *Control*; все эти ссылки инициализируются значением *null*. Поскольку *Control* относится к ссылочным типам, то создание массива сводится только к созданию ссылок; реальные объекты в этот момент не создаются. Возвращенный адрес блока памяти заносится в переменную *myControls*.

Вот как выглядят массивы значимого и ссылочного типов в управляемой куче (рис. 13-1).

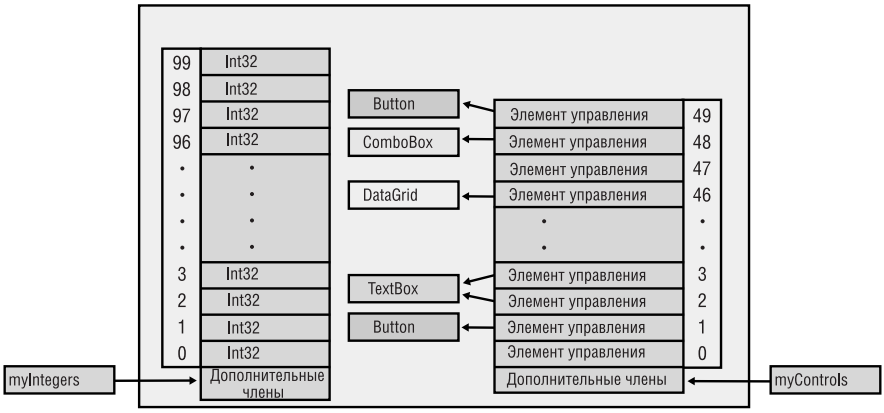


Рис. 13-1. *Массивы значимого и ссылочного типов в управляемой куче*

На этом рисунке показан массив *Controls* после выполнения таких операторов:

```
myControls[1] = new Button();
myControls[2] = new TextBox();
myControls[3] = myControls[2];  // Два элемента ссылаются на один и тот же объект.
myControls[46] = new DataGrid();
myControls[48] = new ComboBox();
myControls[49] = new Button();
```

Согласно общеязыковой спецификации (CLS), нумерация элементов в массиве должна начинаться с нуля. Тогда методы, написанные на C#, 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, Microsoft постаралась оптимизировать их работу. Тем не менее в CLR допускаются и иные варианты индексации массивов, хотя это не приветствуется. Те, для кого вопросы производительности и межъязыковой совместимости программ не имеют большого значения, могут узнать о создании и использовании массивов, начальный индекс которых отличен от 0, далее в этой главе.

Из рис. 13-1 видно, что в каждом массиве есть некоторая *дополнительная информация* (overhead). Это сведения о размерности массива (числе измерений), его нижней границе (чаще всего 0) и числе элементов в каждом измерении. Здесь же указан тип элементов массива. Ниже мы рассмотрим методы, позволяющие запрашивать эти данные.

Пока что нам встречались только одномерные массивы. Желательно ограничиться одномерными массивами с нулевым начальным индексом, которые называют иногда *SZ-массивами* или *векторами*. Векторы обеспечивают лучшую производительность, поскольку для операций с ними используются специальные IL-команды, такие как *newarr*, *ldelem*, *ldelema*, *ldlen* и *stelem*. Но если вам больше подходят многомерные массивы — пожалуйста. Вот некоторые примеры использования многомерных массивов:

```
// Создаем двумерный массив Doubles.
Double[,] myDoubles = new Double[10, 20];
```

```
// Создаем трехмерный массив ссылок на String.
String[,,] myStrings = new String[5, 3, 10];
```

CLR поддерживает также *вложенные* (jagged) массивы. Производительность у 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 *Point*:

```
// Создаем одномерный массив массивов Point.
Point[][] myPolygons = new Point[3][];

// myPolygons[0] ссылается на массив из 10 экземпляров Point.
myPolygons[0] = new Point[10];

// myPolygons[1] ссылается на массив из 20 экземпляров Point.
myPolygons[1] = new Point[20];

// myPolygons[2] ссылается на массив из 30 экземпляров Point.
myPolygons[2] = new Point[30];

// Отображаем точки первого многоугольника.
for (Int32 x = 0; x < myPolygons[0].Length; x++)
   Console.WriteLine(myPolygons[0][x]);
```

Примечание CLR проверяет корректность индекса массива. Иначе говоря, при обращении к элементу массива, состоящему, скажем, из 100 элементов (пронумерованных от 0 до 99), с индексом 100 или -5 генерируется исключение *System.IndexOutOfRangeException*. Доступ к памяти за пределами массива нарушит безопасность типов и откроет потенциальную брешь в защите, недопустимую для верифицируемого кода в CLR. Обычно проверка выхода индекса за границы не имеет существенного значения для производительности, так как JIT-компилятор проверяет границы массивов обычно один раз перед выполнением цикла, а не при каждой итерации. Если же вас все-таки беспокоит потеря скорости, связанная с проверкой индексов, используйте для доступа к массиву небезопасный код C# (см. ниже раздел «Производительность доступа к массиву»).

Приведение типов в массивах

В CLR можно выполнить неявное приведение типа элементов исходного массива, если элементы относятся к ссылочному типу. Условия успешного приведения: оба типа массивов должны быть одной размерности, а также должно иметь место неявное или явное преобразование из типа элементов исходного массива в целевой тип. CLR не допускает приведение массивов с элементами значимых типов к другому типу. (Однако нужного результата позволяет достичь метод *Array.Copy*, создающий новый массив.) Вот пример приведения типа в массиве:

```
// Создаем двумерный массив FileStream.
FileStream[,] fs2dim = new FileStream[5, 10];

// Неявное приведение к двумерному массиву типа Object.
Object[,] o2dim = fs2dim;

// Нельзя привести двумерный массив к одномерному.
// Ошибка компиляции CS0030: Невозможно привести 'object[*,*]' к
// 'System.IO.Stream[]'.
Stream[] s1dim = (Stream[]) o2dim;

// Явное приведение к двумерному массиву Stream.
Stream[,] s2dim = (Stream[,]) o2dim;

// Явное приведение к двумерному массиву String.
// Компиляция проходит, но на этапе выполнения генерируется InvalidCastException.
String[,] st2dim = (String[,]) o2dim;

// Создаем одномерный массив Int32 (значимый тип).
Int32[] i1dim = new Int32[5];

// Нельзя привести к другому типу массив значимого типа.
// Ошибка компиляции CS0030: Невозможно привести 'int[]' к 'object[]'.
Object[] o1dim = (Object[]) i1dim;

// Создаем новый массив, а затем применяем Array.Copy для приведения
// каждого элемента в целевом массиве к желаемому типу.
// Следующий код создает массив ссылок на упакованные элементы типа Int32.
Object[] ob1dim = new Object[i1dim.Length];
Array.Copy(i1dim, ob1dim, i1dim.Length);
```

Метод *Array.Copy* не только копирует элементы одного массива в другой. Действуя, как функция *memmove* языка C, он, в отличие от *memmove* правильно обрабатывает перекрывающиеся области памяти. Он также способен, если требуется, преобразовывать элементы массива при их копировании. Метод *Copy* может выполнять:

- упаковку элементов значимого типа в элементы ссылочного типа, например при копировании *Int32[]* в *Object[]*;
- распаковку элементов ссылочного типа в элементы значимого типа, например при копировании *Object[]* в *Int32[]*;
- *расширение* (widening) элементарных значимых CLR-типов, например при копировании *Int32[]* в *Double[]*;
- приведение элементов с потерями при копировании между массивами разного типа, для которых нельзя определить совместимость по типу массива, например при приведении *Object[]* к *IFormattable[]*. Если все объекты *Object[]* реализует *IFormattable[]*, приведение пройдет успешно.

Вот один из примеров применения *Copy*:

```
// Определяем значимый тип, в котором реализован интерфейс.
internal struct MyValueType : IComparable {
   public Int32 CompareTo(Object obj) {
      ...
   }
}

public static class Program {
   public static void Main() {
      // Создаем массив из 100 элементов значимого типа.
      MyValueType[] src = new MyValueType[100];

      // Создаем массив ссылок IComparable.
      IComparable[] dest = new IComparable[src.Length];

      // Инициализируем массив элементов IComparable ссылками
      // на упакованные версии элементов в исходном массиве.
      Array.Copy(src, dest, src.Length);
   }
}
```

Нетрудно догадаться, что FCL достаточно часто использует достоинства метода *Array.Copy*.

В некоторых ситуациях удобно приводить массив из одного типа в другой, что называют *ковариацией массива* (array covariance). Надо иметь в виду, что ковариация сильно снижает производительность. Допустим, вы написали такой код:

```
String[] sa = new String[100];
Object[] oa = sa;       // oa ссылается на массив элементов String.
oa[5] = "Jeff";         // Удар по производительности: CLR проверяет,
                        // является ли oa типом String; проверка проходит успешно.
oa[3] = 5;              // Удар по производительности: CLR проверяет,
                        // является ли oa типом Int32;
                        // генерируется исключение ArrayTypeMismatchException.
```

Здесь тип переменной *oa* определен как *Object[]*, однако в реальности она ссылается на *String[]*. Компилятор разрешит написать код, пытающийся разместить 5 в элементе массива, потому что 5 относится к типу *Int32*, производному от *Object*. Естественно, CLR стоит на страже безопасности типов, поэтому при присвоении значения элементу массива проверяет корректность операции. Поэтому в процессе выполнения проверяет, содержит ли массив элементы *Int32*. В данном случае это не так, поэтому присвоение признается некорректным, и CLR генерирует исключение *ArrayTypeMismatchException*.

Примечание Если просто нужно скопировать часть элементов из одного массива в другой, метод *BlockCopy* типа *System.Buffer* выполнит это быстрее, чем *Array.Copy*. Однако *BlockCopy* поддерживает только элементарные типы и не предоставляет такие же широкие возможности по приведению типов, как *Copy*. Параметры типа *Int32* выражаются в виде смещений байт, а не индексов элементов. *BlockCopy* разработан для копирования поразрядно совместимых данных из массива одного типа в другой массив битовых таблиц, например копирования *Byte[]*, содержащего Unicode-символы (с соответствующим порядком байт), в *Char[]*. Этот метод частично восполняет невозможность считать массивы просто блоком памяти любого типа.

Если нужно корректно скопировать подмножество элементов из одного массива в другой, следует использовать метод *ConstrainedCopy* типа *SystemArray*. Он гарантирует, что операция копирования завершится верно или будет сгенерировано исключение, уничтожающее все данные в целевом массиве. Это позволяет использовать *ConstrainedCopy* в *области выполнения с ограничениями* (constrained execution region, CER). Такое гарантированное поведение обусловлено тем, что *ConstrainedCopy* требует, чтобы тип элементов исходного массива совпадал или был производным от типа элементов целевого массива. Кроме того, метод не выполняет никакой упаковки, распаковки или приведения с потерями.

Все массивы неявно наследуют классу *System.Array*

Если объявить переменную-массив следующим образом, CLR автоматически создаст тип *FileStream[]* для домена AppDomain.

```
FileStream[] fsArray;
```

Тип *FileStream[]* неявно наследует типу *SystemArray*, поэтому получает в наследство все экземплярные методы и свойства *SystemArray*. Их можно вызывать, используя переменную *fsArray*. Это сильно упрощает работу с массивами, потому что в *SystemArray* определена масса полезных методов и свойств, в том числе *Clone*, *CopyTo*, *GetLength*, *GetLongLength*, *GetLowerBound*, *GetUpperBound*, *Length* и *Rank*.

Тип *SystemArray* также предоставляет много исключительно полезных статических методов 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 эти методы принимают ссылку на массив. У этих методов есть также много перегруженных версий. Вообще-то многие из этих методов предоставляют обобщенные перегруженные версии, обеспечивающие контроль типов во время компиляции и высокую производительность. Я настоятельно рекомендую вам побольше почитать о них в документации к SDK, чтобы понять, насколько полезными и мощными они могут быть.

Все массивы неявно реализуют *IEnumerable*, *ICollection* и *IList*

Многие методы работают с разными наборами объектов только потому, что эти методы объявлены с такими параметрами, как *IEnumerable*, *ICollection* и *IList*. В эти методы можно передавать массивы, так как *System.Array* также реализует эти три интерфейса. *System.Array* реализует эти три необобщенных интерфейса, потому что они обращаются со всеми элементами как с *System.Object*. Однако было бы неплохо, если бы *System.Array* реализовала обобщенные эквиваленты этих интерфейсов, обеспечивая лучший контроль типов во время компиляции и производительность.

Команда разработчиков CLR не хотела, чтобы *System.Array* реализовывала *IEnumerable<T>*, *ICollection<T>* и *IList<T>*, из-за проблем, возникающих с многомерными массивами и массивами, в которых нумерация не начинается с нуля. Определив эти интерфейсы, пришлось обеспечить поддержку всех типов массивов. Вместо этого разработчики CLR пошли на хитрость: при создании типа одномерного массива с индексацией с нуля CLR автоматически реализует в типе *IEnumerable<T>*, *ICollection<T>* и *IList<T>* (где T — тип элементов массива), а также три интерфейса для всех базовых типов типа массива, при условии, что все они — ссылочные типы. Следующая иерархия иллюстрирует эту ситуацию.

```
Object
   Array (необобщенные 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 ссылочного типа)
         .
```

Итак, при наличии следующей строчки кода:

```
FileStream[] fsArray;
```

при создании типа *FileStream[]* CLR автоматически реализует в нем интерфейсы *IEnumerable<FileStream>*, *ICollection<FileStream>* и *IList<FileStream>*. Кроме 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, переменную *fsArray* можно передавать в методы с таким прототипами:

```
void M1(IList<FileStream> fsList) { ... }
void M2(ICollection<Stream> sCollection) { ... }
void M3(IEnumerable<Object> oEnumerable) { ... }
```

Заметьте: если массив содержит элементы значимого типа, тип массива не будет реализовывать интерфейсы базовых классов элемента. Например, при наличии следующей строчки кода:

```
DateTime[] dtArray; // Массив с элементами значимого типа.
```

тип *DateTime[]* будет реализовывать только интерфейсы *IEnumerable<DateTime>*, *ICollection<DateTime>* и *IList<DateTime>*, но не версии этих интерфейсов, общие для *System.ValueType* или *System.Object*. Это означает, что переменную *dtArray* нельзя передавать в качестве аргумента в показанный ранее метод *M3*, потому что массивы значимых типов размещаются в памяти иначе, чем массивы ссылочных типов — об этом говорилось в этой главе ранее.

Передача и возврат массивов

Массив передается в метод всегда по ссылке, а метод может модифицировать элементы в массиве. Если вас это не устраивает, передайте методу копию массива. Имейте в виду, что метод *Array.Copy* выполняет ограниченное копирование и, если элементы массива относятся к ссылочному типу, в новом массиве окажутся ссылки на существующие объекты.

Аналогично, отдельные методы могут возвращать ссылку на массив. Если метод создает и инициализирует массив, возвращение ссылки на массив 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. Если да, возвращайте ссылку на массив. Как правило, этого делать не стоит — пусть лучше метод создаст массив, вызовет *Array.Copy*, а затем вернет ссылку на новый массив. Не забывайте: *Array.Copy* выполняет ограниченное копирование исходного массива.

Когда определяется метод, возвращающий ссылку на массив, в котором нет элементов, метод возвращает либо *null*, либо ссылку на массив с нулевым числом элементов. В такой ситуации Microsoft настоятельно рекомендует возвращать массив нулевой длины, поскольку подобная реализация упрощает код, вызываемый таким методом. Поясню это на примере. Этот код выполняется правильно даже при отсутствии элементов, подлежащих обработке:

```
// Пример простого и понятного кода.
Appointment[] appointments = GetAppointmentsForToday();
for (Int32 a = 0; a < appointments.Length; a++) {
   ...
}
```

А вот следующий код, почти аналогичный предыдущему, выглядит «тяжеловеснее»:

```
// Пример более сложного для понимания кода.
Appointment[] appointments = GetAppointmentsForToday();
if (appointments != null) {
   for (Int32 a = 0, a < appointments.Length; a++) {
      // Выполняем какие-либо действия с элементом appointments[a].
   }
}
```

Вызывающим программам требуется меньше времени на обслуживание методов, которые вместо *null* возвращают массивы с нулевым числом элементов. Между

прочим, то же относится и к полям. Если у вашего типа есть поле, являющееся ссылкой на массив, то в него надо помещать ссылку на массив, даже если в массиве нет элементов.

Создание массивов с ненулевой нижней границей

Выше говорилось о допустимости массивов с ненулевой нижней границей. Собственные массивы можно создавать динамически путем вызова статического метода *CreateInstance* типа *Array*. Существует несколько перегруженных версий этого метода, и в любой можно задать тип элементов массива, размерность массива, нижнюю границу массива и число элементов в каждом измерении. *CreateInstance* выделяет память под массив, записывает заданные параметры в служебную область блока памяти, отведенного под массив, и возвращает ссылку на массив. Если у массива два или более измерений, можно привести ссылку, возвращенную *CreateInstance*, к переменной *ElementType[]* (где *ElementType* — имя типа), чтобы упростить доступ к элементам массива.

Следующий код иллюстрирует динамическое создание двумерного массива значений *System.Decimal*. Первая размерность представляет годы в диапазоне от 2005 до 2009 включительно, вторая — кварталы в диапазоне с 1 до 4 включительно. Код итеративно обрабатывает все элементы динамического массива. Я мог бы жестко прописать в коде границы массива, что дало бы выигрыш в производительности, но предпочел задействовать методы *GetLowerBound* и *GetUpperBound* типа *System.Array*, чтобы продемонстрировать их возможности.

```
using System;

public sealed class DynamicArrays {
   public static void Main() {
      // Мне нужен двумерный 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 получим:

```
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

Внутренние механизмы CLR поддерживают два типа массивов:

- одномерные массивы с нулевым начальным индексом. Их иногда называют *SZ-масивами* (от английских слов single-dimensional, zero-based) или *векторами*;
- одномерные и многомерные массивы с неизвестным начальным индексом.

Познакомиться с разными видами массивов можно, выполнив следующий код (результат работы приводится в комментариях):

```
using System;

public sealed class Program {
       public static void Main() {
       Array a;

       // Создаем одномерный массив с нулевым начальным индексом и без элементов.
       a = new String[0];
       Console.WriteLine(a.GetType());  // "System.String[]"

       // Создаем одномерный массив с нулевым начальным индексом и без элементов.
       a = Array.CreateInstance(typeof(String),
          new Int32[] { 0 }, new Int32[] { 0 });
       Console.WriteLine(a.GetType());  // "System.String[]"

       // Создаем одномерный массив с нулевым начальным индексом и без элементов.
       a = Array.CreateInstance(typeof(String),
          new Int32[] { 0 }, new Int32[] { 1 });
       Console.WriteLine(a.GetType());  // "System.String[*]"  <- ОБРАТИТЕ ВНИМАНИЕ!

       Console.WriteLine();

       // Создаем двухмерный массив с нулевым начальным индексом и без элементов.
       a = new String[0, 0];
       Console.WriteLine(a.GetType());  // "System.String[,]"

       // Создаем двухмерный массив 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[,]"

       // Создаем двухмерный массив 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 каждым оператором *Console.WriteLine* приводится комментарий с результатом работы. В одномерных массивах с нулевой нижней границей выводится «System.String[]», а для одномерных с нижней границей «1» — «System.String[*]». Звездочка означает, что CLR «в курсе», что это массив с ненулевой границей. Заметьте: C# не позволяет определить переменную типа *String[*]*, поэтому, используя синтаксис C#, нельзя обратиться к одномерным массивам с ненулевой нижней границей. Вы вправе для этого воспользоваться методами *GetValue* и *SetValue* типа *Array*, но это намного снижает эффективность из-за дополнительных затрат на вызов метода.

Независимо от нижней границы — «0» или «1» — для многомерных массивов отображается одинаковый тип «System.String[,]». Во время выполнения CLR трактует многомерные массивы так же, как и массивы с ненулевой нижней границей. Можно было бы подумать, что имя типа должно выглядеть так: System.String[*,*], однако CLR не использует звездочки по отношению к многомерным массивам, потому что постоянное наличие звездочек приводило бы в замешательство большинство разработчиков.

Доступ к элементам одномерных массивов с нулевой нижней границей выполняется намного быстрее, чем массивов с ненулевой нижней границей или многомерных массивов. На то есть несколько причин. Во-первых, есть специальные IL-команды (такие как *newarr*, *ldelem*, *ldelema*, *ldlen* и *stelem*) для работы с одномерными массивам с нулевой нижней границей, которые позволяют JIT-компилятору генерировать оптимизированный машинный код. В частности, JIT-компилятор сгенерирует код, предполагающий, что речь идет о массиве с нулевой нижней границей, а это означает, что при доступе к элементу не нужно отнимать смещение. Во-вторых, в стандартных ситуациях JIT-компилятор «умеет» вынести из цикла лишний код проверки границ, выполняя его лишь раз, до выполнения цикла. Вот пример:

```
using System;

public static class Program {
    public static void Main() {
        Int32[] a = new Int32[5];
        for(Int32 index = 0; index < a.Length; index++) {
            // Выполняем какие-то операции с a[index].
        }
    }
}
```

Первое, на что надо обратить внимание в этом коде, — вызов свойства *Length* массива в проверочном операторе цикла *for*. Так как *Length* — свойство, то получение размера фактически является вызовом метода. Однако JIT-компилятор «знает», что *Length* — свойство класса *Array*, поэтому генерирует код, который вызывает свойство только раз и сохраняет результат во временной переменной, кото-

рая проверяется в каждой итерации цикла. Результат — созданный компилятором код выполняется очень «быстро». Честно говоря, некоторые разработчики недооценивают «сообразительность» JIT-компилятора и в попытке помочь компилятору пишут «хитрый код». Однако любые ухищрения практически наверняка негативно сказываются на производительности, а сам код становится менее читабельным, да и поддерживать его сложнее. Лучше положиться на свойство *Length* и не заниматься самодеятельностью, кешируя его в локальной переменной.

Второе, на что стоит обратить внимание, — JIT-компилятор знает, что цикл обращается к элементам массива с нулевой нижней границей, указывая *Length – 1*, поэтому генерирует код, который во время выполнения при каждом доступе к массиву проверяет границы массива. В частности, JIT-компилятор создает код, проверяющий следующее условие:

```
(( Length - 1) <= a.GetUpperBound(0))
```

Проверка выполняется непосредственно перед циклом. В случае положительного результата JIT-компилятор не генерирует в цикле код, каждый раз при доступе к массиву проверяющий, не вышел ли индекс за пределы разрешенного диапазона. Это сильно ускоряет доступ к массиву в цикле.

К сожалению, как уже говорилось, обращение к элементам массива с ненулевой нижней границей или многомерного массива выполняется намного медленнее. Здесь JIT-компилятор не выносит код проверки границ за пределы цикла, так что при каждом доступе к массиву проверяется правильность указания индекса. Кроме того, JIT-компилятор добавляет код, вычитающий нижнюю границу массива из текущего индекса, что также замедляет работу программы, даже для многомерных массивов с нулевой нижней границей.

Так что, если вас серьезно волнует проблема производительности, можно подумать об использовании вложенных (jagged) массивов. C# и CLR также позволяют обращаться к массиву, используя небезопасный (неверифицируемый) код, в сущности, как метод отключения проверки границ индекса при доступе к массиву. Заметьте: метод работы с массивом без проверки индексов применим только к массивам, где элементами являются *SByte*, *Byte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Int64*, *UInt64*, *Char*, *Single*, *Double*, *Decimal*, *Boolean*, перечислимый тип или структура значимого типа, полями которой являются любые из перечисленных выше типов.

Это очень мощный метод, который следует использовать крайне осторожно, потому что при этом предоставляется прямой доступ к памяти. Если при доступе к памяти выйти за границы массива, исключения не возникнет — вы «всего-навсего» повредите память, нарушите безопасность типов и, вполне вероятно, откроете зияющую брешь в защите программы! Поэтому сборке, содержащей небезопасный код, нужно либо обеспечить полное доверие, либо как минимум предоставить разрешение *SecurityPermission* с установленным свойством *Skip Verification*.

Следующий код C# демонстрирует три метода (безопасный доступ, использование вложенного массива и опасный доступ) доступа к двумерному массиву:

```
using System;
using System.Diagnostics;

public static class Program {
   public static void Main() {
```

```
    const Int32 numElements = 100;
    const Int32 testCount = 10000;
    Stopwatch sw;

    // Объявляем двумерный массив.
    Int32[,] a2Dim = new Int32[numElements, numElements];

    // Объявляем двумерный массив как вложенный (вектор векторов).
    Int32[][] aJagged = new Int32[numElements][];
    for (Int32 x = 0; x < numElements; x++)
       aJagged[x] = new Int32[numElements];

    // 1: обращаемся ко всем элементам массива стандартным, безопасным методом.
    sw = Stopwatch.StartNew();
    for (Int32 test = 0; test < testCount; test++)
       Safe2DimArrayAccess(a2Dim);
    Console.WriteLine("{0}: Safe2DimArrayAccess", sw.Elapsed);

    // 2: обращаемся ко всем элементам с использованием метода
    //    вложенных массивов.
    sw = Stopwatch.StartNew();
    for (Int32 test = 0; test < testCount; test++)
       SafeJaggedArrayAccess(aJagged);
    Console.WriteLine("{0}: SafeJaggedArrayAccess", sw.Elapsed);

    // 3: обращаемся ко всем элементам с использованием небезопасного метода.
    sw = Stopwatch.StartNew();
    for (Int32 test = 0; test < testCount; test++)
       Unsafe2DimArrayAccess(a2Dim);
    Console.WriteLine("{0}: Unsafe2DimArrayAccess", sw.Elapsed);
}

private static void Safe2DimArrayAccess(Int32[,] a) {
    for (Int32 x = 0; x < a.GetLength(0); x++) {
       for (Int32 y = 0; y < a.GetLength(1); y++) {
          Int32 element = a[x, y];
       }
    }
}

private static void SafeJaggedArrayAccess(Int32[][] a) {
    for (Int32 x = 0; x < a.GetLength(0); x++) {
       for (Int32 y = 0; y < a[x].GetLength(0); y++) {
          Int32 element = a[x][y];
       }
    }
}

private static unsafe void Unsafe2DimArrayAccess(Int32[,] a) {
    Int32 dim0LowIndex  = 0;  // a.GetLowerBound(0);
    Int32 dim0HighIndex = a.GetUpperBound(0);
```

```
      Int32 dim1LowIndex  = 0;   // a.GetLowerBound(1);
      Int32 dim1HighIndex = a.GetUpperBound(1);

      Int32 dim0Elements = dim0HighIndex - dim0LowIndex;

      fixed (Int32* pi = &a[0, 0]) {
         for (Int32 x = dim0LowIndex; x <= dim0HighIndex; x++) {
            Int32 baseOfDim = x * dim0Elements;
            for (Int32 y = dim1LowIndex; y <= dim1HighIndex; y++) {
               Int32 element = pi[baseOfDim + y];
            }
         }
      }
   }
}
```

Метод *Unsafe2DimArrayAccess* отмечен модификатором *unsafe*, который необходим для оператора *fixed* языка C#. При компиляции этого кода нужно задать параметр */unsafe* или установить флажок **Allow Unsafe Code** на вкладке **Build** окна свойств проекта в Microsoft Visual Studio.

Выполнив программу на своей машине я получил:

```
00:00:02.7977902: Safe2DimArrayAccess
00:00:02.2690798: SafeJaggedArrayAccess
00:00:00.2265131: Unsafe2DimArrayAccess
```

Как видите, безопасный метод самый медленный. Доступ к вложенным массивам занимает чуть меньше времени, однако следует знать, что на создание вложенного массива уходит больше времени, чем на создание многомерного массива из-за того, что при создании вложенного массива в куче создается отдельный объект для каждого измерения, что требует регулярного внимания со стороны сборщика мусора. Придется выбирать из двух вариантов: если нужно создавать много «многомерных массивов» и обращаться к их элементам нечасто, эффективнее создать многомерный массив. Если «многомерный массив» создается лишь раз, а затем выполняется многократное обращение к его элементам, вложенный массив обеспечит лучшую производительность. Ясно, что вторая ситуация встречается чаще всего.

Метод доступа к массиву с использованием небезопасного кода обеспечивает большую скорость — время доступа почти на порядок меньше, чем в случае с вложенными массивами, — при этом работа ведется с реальным двумерным массивом, что значительно экономит ресурсы по сравнению с созданием вложенного массива. «Небезопасный» метод выигрывает вчистую, но у него есть три серьезных недостатка:

- код обращения к элементам массива сложнее для чтения и записи, чем в других методах из-за использования оператора *fixed* языка C# и вычисления адресов памяти;
- в случае ошибки в расчетах адреса памяти возможна перезапись памяти, не принадлежащей массиву. Это может привести к неправильным вычислительным операциям, разрушению памяти, нарушению безопасности типов и возникновению бреши в защите;

- из-за высокой вероятности проблем CLR разрешает небезопасному коду работать только при наличии разрешений со стороны администратора или конечного пользователя. По умолчанию такое разрешение предоставляется всем сборкам, установленным на локальной машине. А вот сборке, загружаемой из интрасети или Интернета, такое разрешение не предоставляется, и при попытке загрузки такой сборки с небезопасным кодом CLR генерирует исключение. Такое поведение по умолчанию можно изменить посредством инструмента CASPol.exe, который поставляется в составе .NET Framework.

Небезопасный доступ к массивам и массивы фиксированного размера

Небезопасный доступ к массиву — очень мощный инструмент, так как позволяет получать доступ к элементам:

- управляемого объекта-массива, расположенного в куче (как показано в предыдущем разделе);
- массива, расположенного в неуправляемой куче. Пример SecureString из главы 11 демонстрирует использование небезопасного доступа к массиву, возвращенному методом *SecureStringToCoTaskMemUnicode* класса *System.Runtime.InteropServices.Marshal*;
- массива, расположенного в стеке потока.

Когда вопрос производительности стоит особенно остро, можно размещать управляемый объект-массив не в куче, а в стеке потока, используя оператор *stackalloc* языка C# (он действует так же, как функция *alloca* языка C). Оператор *stackalloc* можно использовать для создания одномерного массива элементов значимого типа с нулевой нижней границей, причем значимый тип не должен содержать никаких полей ссылочного типа. По сути можно считать это выделением блока памяти, которым можно управлять, используя небезопасные указатели, и, поэтому, нельзя передавать адрес этого буфера памяти подавляющему большинству FCL-методов. Конечно, выделенная в стеке память (массив) будет автоматически освобождена после того, как метод возвратит управление; именно за счет этого достигается выигрыш в производительности. И, конечно же, надо не забыть задать параметр */unsafe* для компилятора C#.

Метод *StackallocDemo* демонстрирует, как надо использовать оператор *stackalloc* языка C#:

```
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]; // Выделяем в стеке память для массива.

         String s = "Jeffrey Richter";    // 15 символов.

         for (Int32 index = 0; index < width; index++) {
            pc[width - index - 1] =
               (index < s.Length) ? s[index] : '.';
         }

         // Следующий оператор выводит на экран ".....rethciR yerffeJ".
         Console.WriteLine(new String(pc, 0, width));
      }
   }

   private static void InlineArrayDemo() {
      unsafe {
         CharArray ca;                // Выделяем в стеке память для массива.
         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] : '.';
         }

         // Следующий оператор выводит на экран ".....rethciR yerffeJ".
         Console.WriteLine(new String(ca.Characters, 0, width));
      }
   }
}

internal unsafe struct CharArray {
   // Этот массив встраивается в структуру.
   public fixed Char Characters[20];
}
```

Массивы являются ссылочными типами, поэтому обычно определенное в структуре поле массива фактически является указателем или ссылкой на массив, а сам массив располагается в памяти структуры. Однако можно встроить массив прямо в структуру, как в структуре *CharArray* в приведенном выше коде. Есть ряд требований к вложению массива внутрь структуры:

- тип должен быть структурой (то есть значимым), нельзя встраивать массив в класс (ссылочный тип);
- поле или структура, в которой оно определено, должно быть отмечено ключевым словом *unsafe*;
- поле массива должно быть отмечено ключевым словом *fixed*;

- массив должен быть одномерным и с нулевой нижней границей;
- разрешены элементы массива только следующих типов: *Boolean*, *Char*, *SByte*, *Byte*, *Int32*, *UInt32*, *Int64*, *UInt64*, *Single* или *Double*.

Встроенные массивы обычно применяются, когда используется небезопасный код, в котором неуправляемая структура данных также содержит встроенный массив. Но ничто не запрещает использовать их в других случаях. Метод *InlineArrayDemo* в приведенном выше коде иллюстрирует использование встроенных массивов. Он делает то же, что метод *StackallocDemo*, но другими методами.

ГЛАВА 14

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*, функцией 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 *SocketPort* наследником одновременно обоих классов: *TransmitData* и *ReceiveData*.

Некоторые языки программирования разрешают множественное наследование, позволяя создать класс *SocketPort*, наследующий двум базовым классам. Однако CLR, а значит, и все основанные на ней языки программирования, множественное наследование не поддерживают. Вместе с тем, CLR позволяет реализовать ограниченное множественное наследование через *интерфейсы*. Эта глава рассказывает об определении и использовании интерфейсов, а также приводит основные правила, когда нужно использовать интерфейсы, а не базовые классы.

Наследование в классах и интерфейсах

В .NET Framework есть класс *System.Object*, в котором определены 4 открытых экземплярных метода: *ToString*, *Equals*, *GetHashCode* и *GetType*. Этот класс является корневым, или основным базовым классом 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*, когда на самом деле он работает с экземпляром какого-либо другого класса;
- **реализацию этих методов** Разработчик может определять 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 классу *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 все сигнатуры методов и их реализации унаследует новый производный класс.

CLR также позволяет определять *интерфейс*, который, в сущности, представляет собой способ задать имя набору сигнатур методов. Интерфейс не содержит реализации методов. Класс наследует интерфейс через указание имени последнего, причем этот класс должен явно содержать реализации методов интерфейса — только в этом случае CLR посчитает определение типа надлежащим. Конечно, реализация методов интерфейса 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 типа интерфейса. Обратимся к конкретике — практическому определению интерфейсов.

Определение интерфейсов

Как уже говорилось, интерфейс — это именованный набор сигнатур методов. Заметьте: интерфейсы также могут определять события, свойства — без параметров или с ними (последний вид в C# называют индексаторами), поскольку все это просто средства упрощения синтаксиса, который обеспечивает сопоставление методов. Однако в интерфейсе нельзя определить никаких методов-конструкторов, а также экземплярных полей.

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 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-типа интерфейса, 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 типа интерфейса можно указать нужную область видимости и доступа (*public*, *protected*, *internal* и т. п.).

В соответствии с соглашением имена типов-интерфейсов начинаются с заглавной буквы *I*, что облегчает их поиск в исходном коде. CLR поддерживает обобщенные интерфейсы (как показано в некоторых предыдущих примерах) и методы интерфейса. В этой главе я расскажу лишь немного о некоторых возможностях обобщенных интерфейсов, подробнее — в главе 16.

Определение интерфейса может «наследовать» другие интерфейсы. Однако слово «наследуют» не совсем точно, поскольку в интерфейсах наследование работает иначе, чем в классах. Я предпочитаю рассматривать наследование интерфейсов как включение контрактов других интерфейсов. Например, определение интерфейса *TCollection<T>* включает контракт интерфейсов *TEnumerable<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 определять тип, реализующий интерфейс, создавать экземпляр этого типа и использовать полученный объект для вызова методов интерфейса. На самом деле в C# это очень просто, но происходящее за кулисами чуть сложнее; об этом немного позже.

Интерфейс *System.IComparable<T>* определяется (в MSCorLib.dll) следующим образом:

```
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;

// 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 методы интерфейса были виртуальными. Если метод явно не определен в коде как виртуальный, компилятор сделает его таковым и, вдобавок, изолированным. Это не позволяет производному классу переопределять методы интерфейса. Если явно задать метод как виртуальный, компилятор сделает его таковым и оставит неизолированным, что предоставит производному классу возможность переопределять методы интерфейса.

Производный класс не в состоянии переопределять методы интерфейса, объявленные изолированными, но может повторно унаследовать тот же интерфейс и предоставить собственную реализацию его методов. При вызове метода интерфейса 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 классу 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 базовому и повторно реализует IDisposable.
internal class Derived : Base, IDisposable {
   // Этот метод не может переопределить базовый 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: *IComparable*, *ICloneable*, *IConvertible*, *IEnumerable*, *IComparable<String>*, *IEnumerable<Char>* и *IEquatable<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, представляющих собой тип-интерфейс. 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, IComparable, ICloneable,
// IConvertible, IEnumerable и т. д.

// Переменная cloneable ссылается на тот же объект String.
ICloneable cloneable = s;
// Используя переменную cloneable, я могу вызвать любой метод,
// 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 переменную-интерфейс
// 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 вызывать 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 CLR могла проверить указатель и точно определить 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 методов интерфейса (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 в иерархии наследования, а также все методы, определенные типами-интерфейсами. Так что, если простой тип определен так:

```
internal sealed class SimpleType : IDisposable {
   public void Dispose() { Console.WriteLine("Dispose"); }
}
```

таблица методов типа содержит следующие записи:

- все экземплярные методы, определенные в типе *Object*, неявно унаследованные от этого базового класса;
- все методы интерфейса, определенные в явно унаследованном интерфейсе *IDisposable*. В этом примере есть только один метод, *Dispose* — единственный метод, определенный в интерфейсе *IDisposable*;
- 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 делать такое предположение, потому что метод открытый, а сигнатуры метода интерфейса 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 методу интерфейса.

Сопоставляя новый метод методу интерфейса, компилятор C# генерирует метаданные, указывающие на то, что обе записи в таблице методов типа *SimpleType* должны ссылаться на одну реализацию. Поясним это на 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();

      // 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
```

Когда в C# перед именем метода стоит имя интерфейса, в котором определен этот метод (в этом примере — *IDisposable.Dispose*), создается *явная реализация метода интерфейса* (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 областью доступа становится *private*, что запрещает любому коду использовать экземпляр класса, просто вызвав метод интерфейса. Единственный способ вызвать метод интерфейса — 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 способ подключения интерфейса (набора поведений или методов) к типу, при котором поведения/методы не становятся очевидными. Если все это кажется вам немного несуразным, вы все *правильно* поняли. Это и есть несуразно. Далее в этой главе я расскажу о стоящих причинах использовать EIMI.

Обобщенные интерфейсы

C# и CLR поддерживают обобщенные интерфейсы, что открывает разработчикам доступ ко многим замечательным функциям. В этом разделе я расскажу о преимуществах использования обобщенных интерфейсов.

Во-первых, обобщенные интерфейсы значительно снижают время компиляции. Некоторые интерфейсы (такие как необобщенный *IComparable*) определяют методы, которые принимают или возвращают параметры типа *Object*. При вызове в коде этих методов интерфейса можно передавать ссылку на экземпляр любого типа. Но обычно это не требуется. Приведем пример:

```
private void SomeMethod1() {
   Int32 x = 1, y = 2;
   IComparable 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 генерируется ArgumentException.
   c.CompareTo("2");
}
```

Ясно, что предпочтительнее обеспечить более строгий контроль типов в интерфейсном методе, поэтому FCL содержит обобщенный интерфейс *IComparable<T>*. Вот новая версия кода, измененная с учетом использования обобщенного интерфейса:

```
private void SomeMethod2() {
   Int32 x = 1, y = 2;
   IComparable<Int32> c = x;

   // CompareTo ожидает Object,
   // но вполне допустимо передать переменную y типа Int32.
   c.CompareTo(y);     // Упаковка не выполняется.

   // CompareTo ожидает Int32;
   // передача "2" (тип String) приводит к ошибке компиляции,
   // с сообщением о невозможности привести тип String к Int32.
   c.CompareTo("2");
}
```

Второе преимущество обобщенных интерфейсов заключается в том, что при работе со значимыми типами выполняется меньше упаковок. 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 реализовывать 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 способ работы с данными более универсальным и менее безопасным методом.

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, выполняя интеграцию с другим кодом, вам придется реализовывать необобщенный интерфейс просто потому, что необобщенной версии не существует. В этом случае, если любой из методов интерфейса принимает или возвращает тип *Object*, будет потерян контроль типов при компиляции и значимые типы будут упаковываться. Можно в некоторой степени исправить эту ситуацию, используя способ, описанный ниже в разделе «Улучшение контроля типов при помощи явной реализации методов интерфейса».

Третьим преимуществом обобщенных интерфейсов является то, что класс может реализовать один интерфейс многократно при условии, что просто используются параметры различного типа. Ниже приведен 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");
    }
}
```

Обобщения и ограничение интерфейса

Я уже говорил о преимуществах обобщенных интерфейсов. В этом разделе я расскажу об ограничении 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 M выполняется без проблем,
        // поскольку Int32 реализует и IComparable, и IConvertible.
        M(x);

        // Компиляция этого вызова M приводит к ошибке, поскольку
        // Guid реализует IComparable, но не реализует IConvertible.
        M(g);
    }

    // T, параметр типа M, 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 тип параметра — интерфейс, аргумент может быть

любого типа класса, если только этот класс реализует указанный интерфейс. Использование нескольких ограничений интерфейса позволяет методу указывать, что передаваемый аргумент должен реализовывать несколько интерфейсов.

На самом деле, если мы ограничим *T* до класса и двух интерфейсов, это будет означать, что тип передаваемого аргумента должен быть указанного базового класса (или быть производным от него), а также должен реализовывать два интерфейса. Такая гибкость позволяет методу диктовать условия вызывающему коду, а в случае несоответствия ограничениям будут возникать ошибки компиляции.

Второе преимущество ограничений интерфейса — избавление от упаковки при передаче экземпляров значимых значений. В предыдущем фрагменте кода методу *M* передавался аргумент *x* (экземпляр типа *Int32*, то есть значимого типа). При передаче *x* в *M* упаковка не выполнялась. Если код метода *M* вызовет *t.CompareTo(…)*, упаковка при вызове также не будет выполняться (упаковка может выполняться для аргументов, передаваемых в *CompareTo*).

С другой стороны, если *M* объявить следующим образом, то для передачи *x* в *M* придется выполнять упаковку:

```
private static Int32 M(IComparable t) {
   ...
}
```

Для ограничений интерфейсов компилятор C# генерирует определенные инструкции IL, которые вызывают метод интерфейса для значимого типа напрямую, без упаковки. Кроме использования ограничений интерфейса, нет другого способа заставить компилятор C# генерировать такие инструкции IL, а следовательно, во всех других случаях вызов метода интерфейса 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, есть два интерфейса, определенных следующим образом:

```
public interface IWindow {
   Object GetMenu();
}

public interface IRestaurant {
   Object GetMenu();
}
```

Требуется определить тип, реализующий оба этих интерфейса. Тогда нужно реализовать члены типа, используя явную реализацию методов:

```
// Этот тип производный 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* содержит реализацию какого интерфейса.

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();

// Эти строки вызывают метод IWindow.GetMenu.
IWindow window = mp;
window.GetMenu();

// Эти строки вызывают метод IRestaurant.GetMenu.
IRestaurant restaurant = mp;
restaurant.GetMenu();
```

Улучшение контроля типов при помощи явной реализации методов интерфейса

Интерфейсы — замечательная вещь, так как позволяют определять 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 методов интерфейса принимает параметры типа *System.Object* или возвращает значение типа *System.Object*, безопасность типов при компиляции будет нарушена и будет выполняться упаковка. В этом разделе я покажу, как можно использовать EIMI, чтобы несколько смягчить ситуацию.

Вот очень типичный интерфейс *IComparable*:

```
public interface IComparable {
   Int32 CompareTo(Object other);
}
```

Этот интерфейс определяет один метод, который принимает параметр типа *System.Object*. Если я определю собственный тип, реализующий этот интерфейс, определение типа будет выглядеть примерно так:

```
internal struct SomeValueType : IComparable {
   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.
}
```

«Идеальность» этого кода нарушают два недостатка:

- 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 к *SomeValueType*, возникает исключение *InvalidCastException*.

Оба недостатка можно исправить, используя EIMI. Вот модифицированная версия *SomeValueType*, в которой использована EIMI:

```
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 *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 с использованием EIMI.

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 при компиляции.
}
```

Однако, если мы определим переменную интерфейсного типа, мы 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 говорилось, при приведении экземплярного типа к интерфейсному типу CLR должна упаковывать экземпляр значимого типа. Поэтому в приведенном выше методе *Main* выполняются две упаковки.

EIMI часто используется при реализации таких интерфейсов, как *IConvertible*, *ICollection*, *IList* и *IDictionary*. Это позволяет обеспечить в методах интерфейсов безопасность типов при компиляции и избавиться от упаковки значимых типов.

Осторожно с явной реализацией методов интерфейсов!

Исключительно важно понимать некоторые особенности использования EIMI, из-за которых следует избегать EIMI везде, где это возможно. К счастью, в некоторых случаях вместо EIMI можно обойтись обобщенными интерфейсами. Но все равно остаются ситуации, когда EIMI необходима (например, при реализации двух методов интерфейса с одинаковыми именами и сигнатурами). С использованием EIMI связаны следующие большие сложности (далее я расскажу о них поподробнее):

- отсутствует документация, объясняющая, как именно тип реализует EIMI-метод, и нет поддержки Microsoft Visual Studio IntelliSense;
- при приведении к интерфейсному типу, экземпляры значимого типа упаковываются;
- EIMI нельзя вызвать из производного типа.

В документации .NET Framework по методам для типов можно найти сведения о явной реализации методов интерфейсов, но справка по конкретным типам отсутствует — доступна только общая информация о методах интерфейсов. Например, о типе *Int32* говорится, что он реализует все методы интерфейса *IConvertible*. И это хорошо, потому что разработчики могут узнать, что такие методы существуют, однако это 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); // Пытаемся вызвать метод интерфейса IConvertible.
}
```

При компиляции этого метода компилятор C# вернет следующую ошибку: «error CS0117: 'int' does not contain a definition for 'ToSingle'» (ошибка CS0117: 'int' не содержит определения для '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 *ToSingle* типа *Int32*, сначала нужно привести его к *IConvertible*:

```
public static void Main() {
   Int32 x = 5;
   Single s = ((IConvertible) x).ToSingle(null);
}
```

Требование приводить тип далеко не очевидно, и многие разработчики не смогут самостоятельно додуматься до этого. Но на этом проблемы не заканчиваются — при приведении значимого типа *Int32* к *IConvertible* он упаковывает значимый тип, на что тратится память и из-за чего снижается производительность. Это вторая большая проблема.

Третья и, наверное, самая большая проблема с EIMI состоит в том, что EIMI нельзя вызвать из производного класса. Вот пример:

```
internal class Base : IComparable {

   // Явная реализация метода интерфейса (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 возникновению ошибки:
      // "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 возникновению ошибки 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 переменной *c* типа *IComparable*. Затем я использую *c* для вызова *CompareTo*. Однако открытый метод *CompareTo* класса *Derived* служит 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*. Но иногда нельзя просто удалить интерфейс из типа, поскольку производный тип должен реализовывать метод интерфейса. Лучший способ исправить ситуацию — в дополнение к явно реализованному методу интерфейса 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 метода интерфейса (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 использовать защищенный метод. Вполне можно сделать метод защищенным, а не открытым, но это потребует небольших изменений. Как видите, EIMI действительно нужно применять с большой осторожностью. Когда разработчики впервые узнали о EIMI, многие посчитали это отличной новостью и стали использовать ее везде, где только можно. Не попадайтесь на эту удочку! EIMI полезна в некоторых случаях, но следует избегать ее использования там, где можно обойтись другими средствами.

Дилемма проектировщика: 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 типу *System.ValueType* и поэтому не могут наследовать произвольному базовому классу. В этом случае нужно определять интерфейс.
- **Простота использования** Разработчику проще определить новый тип, производный от базового, чем создавать 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 работают нормально 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 приведет к необходимости 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 рассмотреть 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 *наборов* (collections), то их специалисты Microsoft реализовали в FCL на основе интерфейсов. В пространстве имен *System.Collections.Generic* определено несколько интерфейсов для работы с наборами: *IEnumerable<T>*, *ICollection<T>*, *IList<T>* и *IDictionary<TKey, TValue>*. Затем Microsoft предоставила 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-наборов 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. Так, все они поддерживают подмножество элементов, которые поддаются перечислению, и позволяют добавлять и удалять элементы. Если есть ссылка на объект, тип которого реализует интерфейс *IList<T>*, можно написать код, который добавляет, удаляет и ищет элементы, не зная конкретный тип набора. Это очень мощный механизм.

Наконец, нужно сказать, что на самом деле можно определять интерфейс и создавать базовый класс, который реализует интерфейс. Например, FCL определяет 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 выбирать наиболее предпочтительный из двух вариантов.

ГЛАВА 15

Делегаты

В этой главе я расскажу о функциях обратного вызова — чрезвычайно полезном механизме, который программисты активно используют уже много лет. Microsoft .NET Framework поддерживает механизм функций обратного вызова при помощи *делегатов* (delegate). В отличие от механизмов обратного вызова из других платформ, таких как неуправляемый C++, функциональность делегатов намного богаче. Например, делегаты обеспечивают безопасность типов при выполнении метода обратного вызова (способствуя решению одной из важнейших задач CLR). Делегаты также поддерживают последовательный вызов нескольких методов и позволяют вызывать как статические, так и экземплярные методы.

Знакомство с делегатами

Функция *qsort* исполняющей среды C сортирует элементы массива, используя функцию обратного вызова. В Microsoft Windows функции обратного вызова требуются для выполнения оконных процедур, процедур перехвата, асинхронного вызова процедур и пр. В .NET Framework у методов обратного вызова масса приложений. Например, зарегистрировав методы обратного вызова, можно получать самые разные уведомления: о необработанных исключениях, изменении состоянии окна, выборе элементов меню, изменениях файловой системы и завершении асинхронных операций.

В неуправляемом C/C++ адрес функции — это всего лишь адрес в памяти, не несущий дополнительной информации. Он не позволяет узнать, ни сколько параметров ожидает функция, ни их тип или тип значения, возвращаемого функцией, ни правила вызова функции. Короче, в неуправляемом C/C++ функции обратного вызова не обеспечивают безопасность типов (хотя их отличает высокая скорость выполнения).

В .NET Framework функции обратного вызова играют не менее важную и всеобъемлющую роль, чем в неуправляемом программировании для Windows. Однако в .NET Framework есть механизм делегатов, обеспечивающий безопасность типов. Вот как объявляют, создают и используют делегаты:

```
using System;
using System.Windows.Forms;
using System.IO;
```

```
// Определяем тип-делегат; экземпляры ссылаются на метод,
// который принимает параметр типа Int32 и возвращает 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." необязательный.
      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 какие-либо 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();
   }
}
```

Теперь я расскажу, что делает этот код. Прежде всего обратите внимание на объявление внутреннего делегата *Feedback*. В этом примере делегат задает сигнатуру метода обратного вызова. Здесь *Feedback* определяет метод, принимающий один параметр типа *Int32* и возвращающий *void*. В некотором смысле делегат очень напоминает *typedef* из неуправляемого C/C++, представляющий адрес функции.

Класс *Program* определяет закрытый статический метод — *Counter*. Он перечисляет целые числа, находящиеся между параметрами *from* и *to*. Метод *Counter* также принимает параметр *fb*, который представляет собой ссылку на объект-делегат *Feedback*. *Counter* итеративно перечисляет целые числа, для каждого числа (если *fb* не равна null) вызывая метод обратного вызова (определенный в переменной *fb*). Методу обратного вызова передается значение обрабатываемого элемента и его номер. Метод обратного вызова может обрабатывать любой элемент как ему угодно.

Использование делегатов для обратного вызова статических методов

Теперь, когда мы разобрались в конструкции и принципе работы метода *Counter*, посмотрим, как используют делегаты для вызова статических методов. Мы разберем метод *StaticDelegateDemo* из предыдущего примера.

Сначала *StaticDelegateDemo* вызывает метод *Counter*, которому в качестве третьего параметра, соответствующего параметру *fb* метода *Counter*, передается *null*. Поскольку в этом примере параметр *fb* содержит *null*, при обработке всех элементов обратного вызова не происходит.

При втором вызове *Counter* создается и новому объекту делегата *Feedback* передается другое значение в третьем параметре. Этот объект-делегат служит оболочкой для другого метода, позволяя выполнять обратный вызов последнего косвенно, через оболочку. Конструктору типа *Feedback* передается имя статического метода (в этом примере — *Program.FeedbackToConsole*), указывающее метод, для которого требуется создать оболочку. Ссылка, которую возвращает оператор *new*, передается методу *Counter*. Теперь при обработке каждого из элементов набора метод *Counter* может вызвать статический метод *FeedbackToConsole*, определенный в типе *Program*. Метод *FeedbackToConsole* просто выводит в консоль строку, указывающую обрабатываемый элемент и его значение.

Примечание Метод *FeedbackToConsole* определен внутри типа *Program* как закрытый, однако метод *Counter* способен вызывать закрытый метод типа *Program*. Здесь не возникает проблем с безопасностью, так как *Counter* и *FeedbackToConsole* определены в одном типе. Но код будет работать без проблем, даже если *Counter* определен в другом типе. Короче говоря, не возникает никаких проблем с безопасностью или уровнем доступа, если код одного типа вызывает закрытый член другого типа через делегат при условии, что объект-делегат создан кодом, у которого есть нужные разрешения и доступ.

Третий и второй вызовы *Counter* в методе *StaticDelegateDemo* отличаются лишь тем, что объект делегата *Feedback* является оболочкой для другого статического метода — *Program.FeedbackToMsgBox*. Метод *FeedbackToMsgBox* создает строку, указывающую обрабатываемый элемент, которая затем выводится в окне с сообщением.

Ничто в этом примере не нарушает безопасности типов. Например, при создании объекта делегата *Feedback* компилятор гарантирует, что прототипы методов *FeedbackToConsole* и *FeedbackToMsgBox* типа *Program* не будут отличаться от заданного делегатом *Feedback*, то есть оба метода будут принимать один параметр (типа *Int32*) и возвращать значения одного и того же типа (*void*). Но что, если у метода *FeedbackToConsole* был бы такой прототип:

```
private static Boolean FeedbackToConsole(String value) {
   ...
}
```

Компилятор C# откажется компилировать такой код и вернет сообщение об ошибке «error CS0123: The signature of method ‘FeedbackToConsole’ does not match

this delegate type» («ошибка CS0123: Сигнатура метода 'FeedbackToConsole' не соответствует типу этого делегата»).

И C#, и CLR поддерживают прямую и обратную ковариацию ссылочных типов при привязке метода к делегату. *Ковариация* (covariance) подразумевает, что метод может возвратить тип, производный от типа, возвращаемого делегатом. *Обратная ковариация* (contra-variance) означает, что метод может принимать параметр, который является базовым для типа параметра делегата. Например, если делегат определить так:

```
delegate Object MyCallback(FileStream s);
```

можно создать экземпляр этого делегата, связанный с методом, прототип которого выглядит примерно так:

```
String SomeMethod(Stream s);
```

Здесь тип значения, возвращаемого методом *SomeMethod* (то есть String), является типом, производным от типа, возвращаемого делегатом (*Object*); такая ковариация разрешена. Тип параметра метода *SomeMethod* (то есть *Stream*) — это тип, являющийся базовым классом для типа параметра делегата (*FileStream*); такая обратная ковариация разрешена.

Заметьте: прямая и обратная ковариация поддерживаются только для ссылочных типов, но не для значимых типов или *void*. Так, например, я не могу связать следующий метод с делегатом *MyCallback*:

```
Int32 SomeOtherMethod(Stream s);
```

Хотя тип значения, возвращаемого *SomeOtherMethod*, (то есть *Int32*) является производным от типа значения, возвращаемого методом *MyCallback* (то есть *Object*); такая форма ковариации не разрешена, потому что *Int32* — значимый тип. Значимые типы и *void* не могут использоваться для прямой и обратной ковариации, потому что их структура памяти изменяется, тогда как структура памяти ссылочных типов — всегда указатель. К счастью, компилятор C# возвращает ошибку при попытке выполнить неразрешенную операцию.

Использование делегатов для обратного вызова экземплярных методов

Я показал, как вызывать с помощью делегатов статические методы, но делегаты позволяют вызывать и экземплярные методы объекта. Чтобы понять, как работает обратный вызов экземплярного метода, взгляните на метод *InstanceDelegateDemo* из показанного выше кода.

Обратите внимание, что объект *p* типа *Program* создается в методе *InstanceDelegateDemo*. У объекта *Program* нет экземплярных полей или свойств — я создал его просто для наглядности. Когда при вызове *Counter* создается новый объект-делегат *Feedback*, его конструктору передается *p.FeedbackToFile*. Это заставляет делегат стать оболочкой для ссылки на метод *FeedbackToFile*, который является экземплярным, а не статическим методом. Когда *Counter* обращается к методу обратного вызова, указанному в параметре *fb*, вызывается экземплярный метод

FeedbackToFile, а адрес только что созданного объекта *p* передается как явный параметр в экземплярный метод.

Метод *FeedbackToFile* работает подобно *FeedbackToConsole* и *FeedbackToMsgBox*, отличаясь лишь тем, что открывает файл и дописывает строку в конец файла. (Создаваемый методом файл Status располагается в каталоге AppBase приложения.)

Этот пример призван показать, что делегаты могут служить оболочкой как для экземплярных, так и для статических методов. В случае экземплярного метода делегат должен знать экземпляр объекта, который собирается обрабатывать вызываемый им метод. Создание оболочек экземплярных методов полезно, так как позволяет коду внутреннего объекта получить доступ к экземплярным членам объекта. Это означает, что у объекта может быть состояние, которое может использоваться во время работы метода обратного вызова.

Правда о делегатах

Может показаться, что использовать делегаты несложно: вы определяете делегат ключевым словом *delegate* языка C#, создаете его экземпляр с помощью знакомого оператора *new* и вызываете метод обратного вызова, пользуясь известным вам синтаксисом «вызова метода» (в котором вместо имени метода указывается переменная, ссылающаяся на объект-делегат).

Но на самом деле делегаты гораздо сложнее, чем показывают продемонстрированные примеры. Чтобы скрыть эту сложность, компиляторам и CLR приходится изрядно потрудиться «за кулисами». Мы разберем совместную работу компилятора и CLR для реализации делегатов, поскольку знание этих тонкостей поможет вам глубже понять принцип работы делегатов и научиться эффективно и рационально их применять. Я также коснусь дополнительных возможностей, доступных при использовании делегатов.

Давайте-ка еще раз изучим эту строку кода:

```
internal delegate void Feedback(Int32 value);
```

На самом деле, обнаружив такую строку, компилятор создает 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);
}
```

У определенного компилятором класса четыре метода: конструктор, *Invoke*, *BeginInvoke* и *EndInvoke*. В этой главе мы уделим основное внимание конструкто-

ру и методу *Invoke*. О методах *BeginInvoke* и *EndInvoke* я расскажу подробнее в главе 23, где речь пойдет о модели программирования асинхронных операций.

Изучив результирующий модуль при помощи ILDasm.exe, можно убедиться, что компилятор действительно автоматически сгенерировал этот класс (рис. 15-1):

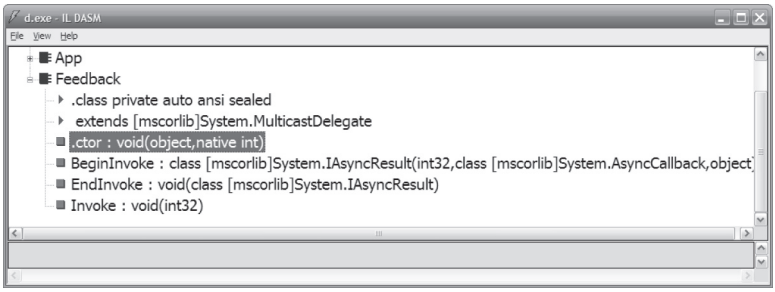


Рис. 15-1. *ILDasm.exe показывает метаданные делегата, сгенерированные компилятором*

В этом примере компилятор создал класс *Feedback*, производный от типа *System.MulticastDelegate*, определенного в библиотеке классов .NET Framework Class Library (все типы делегатов — потомки *MulticastDelegate*).

Внимание! Класс *System.MulticastDelegate* является производным от *System.Delegate*, который, в свою очередь, наследует классу *System.Object*. Наличие двух классов делегатов — факт неприятный, но так уж сложилось исторически; в FCL должен быть лишь один класс делегата. К сожалению, нужно помнить об обоих классах, потому что, даже если вы будете в качестве типов делегатов выбирать в качестве базового класс *MulticastDelegate*, иногда придется работать с типами делегатов, используя методы, определенные в классе *Delegate*, а не *MulticastDelegate*. В частности, у класса *Delegate* есть статические методы *Combine* и *Remove*. (Что эти методы делают, я объясню позже.) Сигнатуры обоих методов указывают, что они принимают параметры типа *Delegate*. Так как ваши типы делегатов являются производными от *MulticastDelegate*, который наследует *Delegate*, экземпляры вашего типа делегата можно передавать в эти методы.

Это закрытый класс, так как делегат был объявлен в исходном тексте как *internal*. Если бы он был объявлен в исходном тексте с модификатором *public*, то сгенерированный компилятором класс *Feedback* был бы соответственно открытым. Надо иметь в виду, что типы делегатов могут быть определены как внутри класса (вложенные внутрь другого типа), так и с глобальной областью действия. В сущности, поскольку делегаты — это классы, их можно определять в любом месте, где можно определить класс.

Поскольку все типы делегатов являются потомками *MulticastDelegate*, они наследуют все поля, свойства и методы *MulticastDelegate*. Наверное, самые важные из них — это три закрытых поля (табл. 15-1).

Табл. 15-1. Важнейшие закрытые поля *MulticastDelegate*

Поле	Тип	Описание
_target	*System.Object*	Когда объект-делегат служит оберткой статического метода, это поле содержит null. Когда объект-делегат служит оберткой экземплярного метода, это поле содержит ссылку на объект, с которым должны выполняться операции при вызове метода обратного вызова. Иначе говоря, это поле содержит значение, которое нужно передать в явном параметре *this* экземплярного метода
_methodPtr	*System.Int32*	Внутреннее целочисленное значение, используемое CLR для идентификации метода обратного вызова
_prev	*System.MulticastDelegate*	Обычно содержит null. Оно может ссылаться на массив делегатов при создании цепочки делегатов (об этом чуть позже)

Заметьте: у всех делегатов конструктор принимает два параметра — ссылку на объект и целое число, ссылающееся на метод обратного вызова. Но, изучив исходный текст, можно видеть, что я передаю такие значения, как *Program.FeedbackToConsole* или *p.FeedbackToFile*. Интуиция должна подсказать вам, что этот код не должен компилироваться!

Но компилятор знает, что создается делегат, и путем синтаксического анализа кода определяет объект и метод, на которые передается ссылка. Ссылка на объект передается в параметре *object* конструктора, а специальное значение *IntPtr* (получаемое из маркеров метаданных MethodDef или MethodRef), идентифицирующее метод, передается в параметре *method*. В случае статического метода в параметре *object* передается *null*. Внутри конструктора значения этих параметров сохраняются в закрытых полях *_target* и *_methodPtr*. Конструктор также заносит *null* в поле *_invocationList*. Пока я проигнорирую это поле, но в разделе «Цепочки делегатов» мы его разберем подробно.

Итак, любой объект делегата на деле является оболочкой метода и объекта, обрабатываемого этим методом. Поэтому при наличии двух следующих строк кода:

```
Feedback fbStatic   = new Feedback(Program.FeedbackToConsole);
Feedback fbInstance = new Feedback(new Program().FeedbackToFile);
```

переменные *fbStatic* и *fbInstance* ссылаются на два разных инициализированных объекта-делегата *Feedback* (рис. 15-2).

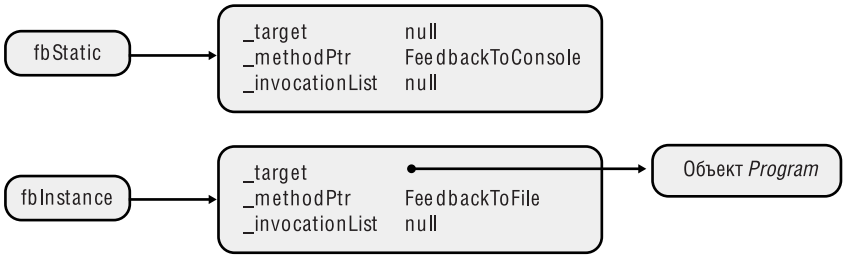


Рис. 15-2. *Переменная, ссылающаяся на делегат статического метода, и переменная, ссылающаяся на делегат экземплярного метода*

В классе *Delegate* определены два неизменяемых открытых экземплярных свойства: *Target* и *Method*. При наличии ссылки на объект делегата можно запросить значения этих свойств. *Target* возвращает ссылку на объект, обрабатываемый при обратном вызове метода. В сущности, *Target* возвращает значение, хранимое в закрытом поле *_target*. Если этот метод — статический, *Target* возвращает *null*. Свойство *Method* возвращает объект *System.Reflection.MethodInfo*, описывающий метод обратного вызова. В сущности, у свойства *Method* есть внутренний механизм, который преобразует значение из закрытого поля *_methodPtr* в объект *MethodInfo* и возвращает его.

Эту информацию можно использовать по-разному, например, чтобы проверить, не ссылается ли объект делегата на экземплярный метод определенного типа:

```
Boolean DelegateRefersToInstanceMethodOfType(MulticastDelegate d, Type type) {
   return((d.Target != null) && d.Target.GetType() == type);
}
```

Можно также написать код, проверяющий имя метода обратного вызова (например, *FeedbackToMsgBox*):

```
Boolean DelegateRefersToMethodOfName(MulticastDelegate d, String methodName) {
   return(d.Method.Name == methodName);
}
```

Есть еще масса других ситуаций, в которых могут быть полезными эти свойства.

Теперь вы знаете, как создаются объекты-делегаты, — поговорим о том, как вызываются методы обратного вызова. Для удобства я еще раз покажу код метода *Counter*:

```
private static void Counter(Int32 from, Int32 to, Feedback fb) {
   for (Int32 val = from; val <= to; val++) {
      // Если указаны какие-либо методы обратного вызова, вызвать их.
      if (fb != null)
         fb(val);
   }
}
```

Посмотрим повнимательнее на строку кода, следующую сразу после комментария. Оператор *if* сначала проверяет, не равна ли null переменная *fb*. Если нет, выполняется следующая строка, вызывающая метод обратного вызова. Проверка на неравенство null нужна, потому что *fb* — всего лишь переменная, которая *может* ссылаться на объект-делегат *Feedback*, но может быть и null. Может показаться, что я вызываю функцию *fb*, передавая ей один параметр (*val*), но такой функции нет. И в этом случае компилятор генерирует код для вызова метода *Invoke* объекта-делегата, поскольку знает, что *fb* — это переменная, ссылающаяся на объект-делегат. Иначе говоря, обнаружив строку:

```
fb(val);
```

компилятор генерирует код, как если бы он увидел в исходном тексте следующее:

```
fb.Invoke(val);
```

Изучив код метода *Counter* при помощи ILDasm.exe, можно убедиться, что компилятор генерирует код, вызывающий метод *Invoke* из типа делегата. На рис. 15-3 показан IL-код метода *Counter*. Команда в строке IL_0009 содержит вызов метода *Invoke* объекта *Feedback*.

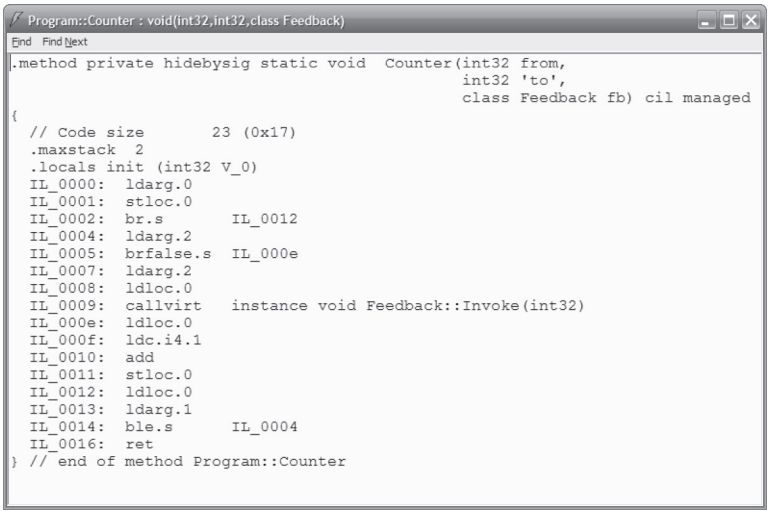


Рис. 15-3. *ILDasm.exe доказывает, что компилятор сгенерировал вызов метода Invoke делегата Feedback*

Вообще, можно в *Counter* вызывать *Invoke* явно, как здесь:

```
private static void Counter(Int32 from, Int32 to, Feedback fb) {
   for (Int32 val = from; val <= to; val++) {
      // Если указаны какие-либо методы обратного вызова, вызвать их.
      if (fb != null)
         fb.Invoke(val);
   }
}
```

Наверное, вы помните, что компилятор определил метод *Invoke* при определении класса *Feedback*. При вызове метода *Invoke* он использует поля *_target* и *_methodPtr* для вызова желаемого метода на заданном объекте. Заметьте: сигнатура метода *Invoke* соответствует таковой делегата, то есть и делегат *Feedback*, и (созданный компилятором) метод *Invoke* принимают параметр типа *Int32* и возвращают *void*.

Использование делегатов для обратного вызова множественных методов (цепочки делегатов)

Делегаты сами по себе невероятно полезны, но поддержка цепочек делает их еще полезнее. *Цепочка делегатов* (chaining) — это подмножество, или набор объектов-делегатов, которое позволяет вызывать все методы, представленные делегатами набора. Чтобы лучше понять принцип работы цепочки, обратите внимание на метод *ChainDelegateDemo1* в коде, приведенном в начале главы. В нем после

оператора *Console.WriteLine* я создаю три объекта-делегата, на которые соответственно ссылаются три переменные *fb1*, *fb2* и *fb3* (рис. 15-4).

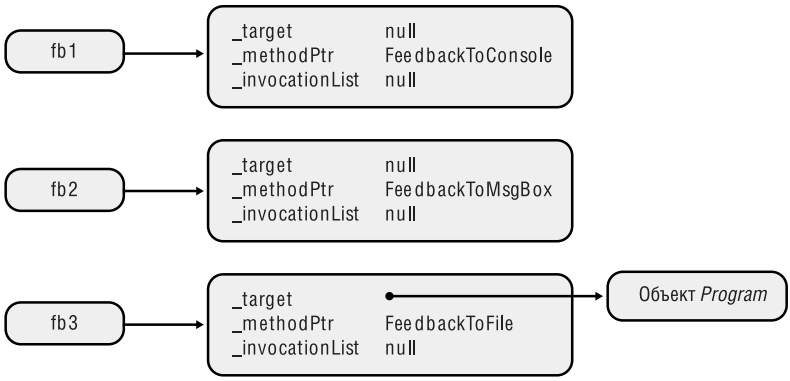


Рис. 15-4. *Начальное состояние объектов-делегатов, на которые ссылаются три переменные fb1, fb2 и fb3*

Переменная-ссылка на объект-делегат *Feedback* должна ссылаться на цепочку или набор делегатов, служащих оболочками методам обратного вызова. Инициализация *fbChain* значением *null* говорит об отсутствии методов обратного вызова. Открытый статический метод *Combine* класса *Delegate* используется для добавления делегата в цепочку:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb1);
```

При выполнении этой строчки кода метод *Combine* «видит» попытку объединить *null* и *fb1*. Код метода *Combine* просто возвращает значение *fb1*, а в переменной *fbChain* размещается ссылка на тот же объект-делегат, на который ссылается *fb1* (рис. 15-5).

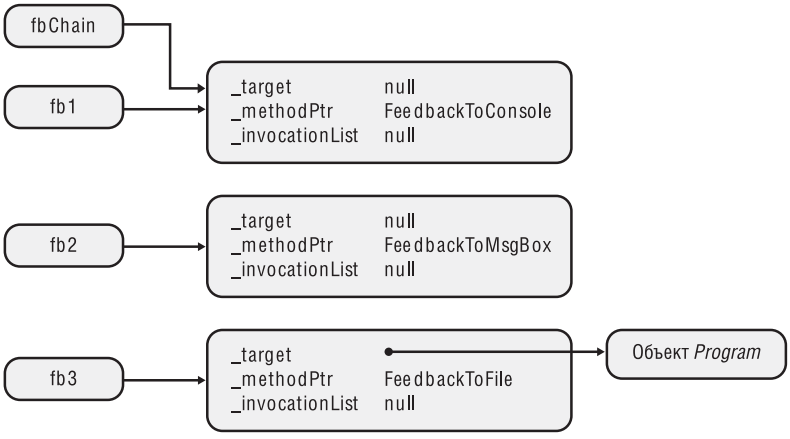


Рис. 15-5. *Состояние объектов-делегатов после добавления в цепочку второго делегата*

Чтобы добавить в цепочку еще один делегат, снова вызывается метод *Combine*:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb2);
```

Код метода *Combine* «видит», что *fbChain* уже ссылается на объект-делегат, и поэтому создает новый объект-делегат. Новый объект-делегат инициализирует свои закрытые поля *_target* и *_methodPtr*. Присваиваемые полям значения для данного обсуждения не важны, но важно то, что поле *_invocationList* инициализируется ссылкой на массив объектов-делегатов. Первый элемент массива (с индексом 0) инициализируется ссылкой на делегат, служащий оберткой метода *FeedbackToConsole* (это делегат, на который сейчас ссылается *fbChain*). Второй элемент массива (индекс 1) инициализируется ссылкой на делегат, служащий оберткой метода *FeedbackToMsgBox* (на этот делегат ссылается *fb2*). Наконец, переменной *fbChain* присваивается ссылка на вновь созданный объект-делегат (рис. 15-6).

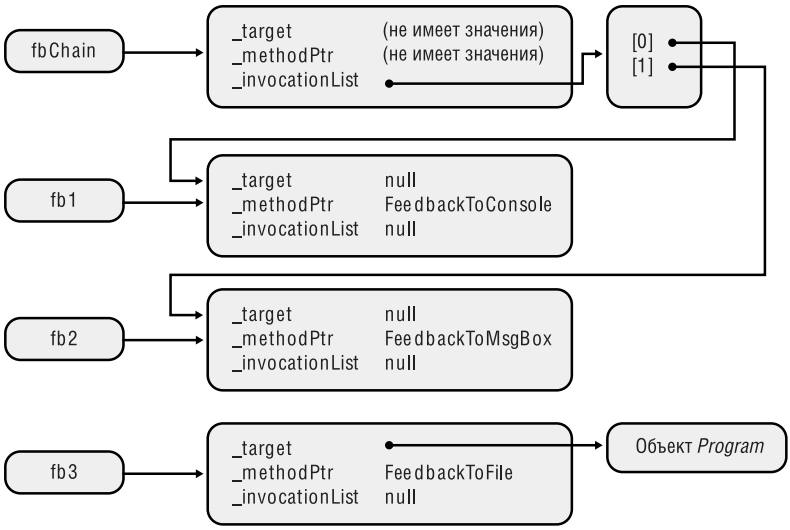


Рис. 15-6. *Состояние объектов-делегатов после добавления в цепочку второго делегата*

Для создания третьего делегата снова вызывается метод *Combine*:

```
fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
```

И снова, видя, что *fbChain* уже ссылается на объект-делегат, *Combine* создает новый объект-делегат (рис. 15-7). Как и раньше, новый объект-делегат инициализирует свои закрытые поля *_target* и *_methodPtr* какими-то значениями, а поле *_invocationList* инициализируется ссылкой на массив объектов-делегатов. Первый и второй элементы массива (индексы 0 и 1) инициализируются ссылками на те же делегаты, на которые ссылался предыдущий объект-делегат в массиве. Третий элемент массива (индекс 2) инициализируется ссылкой на делегат, служащий оберткой метода *FeedbackToFile* (на этот делегат ссылается *fb3*). Наконец, переменной *fbChain* присваивается ссылка на вновь созданный объект-делегат. Заметьте: ранее созданный делегат и массив, на который ссылается его же поле *_invocationList*, теперь подлежат обработке механизмом сборки мусора.

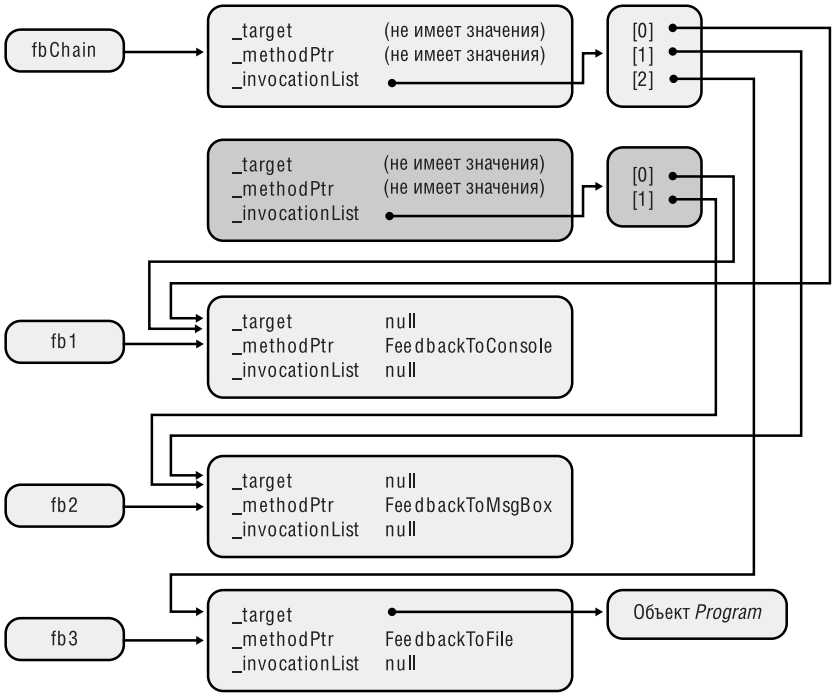


Рис. 15-7. *Окончательное состояние объектов-делегатов в готовой цепочке*

После выполнения всего кода создания цепочки, переменная *fbChain* передается методу *Counter*:

```
Counter(1, 2, fbChain);
```

Внутри *Counter* содержится код, неявно вызывающий метод *Invoke* по отношению к объекту-делегату *Feedback*, как показано на рис. 15-3. Когда *Invoke* вызывается по отношению к делегату, на который ссылается *fbChain*, делегат обнаруживает, что поле *_invocationList* не равно null, и инициируется выполнение цикла, итеративно обрабатывающего все элементы массива путем вызова метода, оболочкой которого служит указанный делегат. У нас методы вызываются в следующей последовательности: *FeedbackToConsole*, *FeedbackToMsgBox* и *FeedbackToFile*.

В псевдокоде метод *Invoke* класса *Feedback* выглядит примерно так:

```
public void Invoke(Int32 value) {
   Delegate[] delegateSet = _invocationList as Delegate[];
   if (delegateSet != null) {
      // Этот массив делегатов указывает делегаты, которые нужно вызвать.
      foreach (Feedback d in delegateSet)
         d(value); // Вызываем каждый делегат.
   } else {
      // Этот делегат определяет один метод,
      // который нужно вызвать по механизму обратного вызова.
      // Вызвать метод обратного вызова для указанного объекта.
      _methodPtr.Invoke(_target, value);
      // Предыдущая строка – очень приблизительная копия реального кода.
```

```
      // Происходящее на самом деле не поддается
      // иллюстрации средствами C#.
   }
}
```

Делегаты можно удалять из цепочки, вызывая статический метод *Remove* объекта *Delegate*. В конце кода метода *ChainDelegateDemo1* есть пример:

```
fbChain = (Feedback) Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
```

Метод *Remove* сканирует массив делегатов (с конца и до члена с индексом 0), поддерживаемых объектом-делегатом, на который ссылается первый параметр (в нашем примере *fbChain*). *Remove* ищет делегат, поля *target* и *_methodPtr* которого совпадают с соответствующими полями второго параметра (в нашем примере нового делегата *Feedback*). Если *Remove* находит совпадение и в массиве остается более одного элемента, создается новый объект-делегат — создается массив *_invocationList* и инициализируется ссылкой на все элементы исходного массива за исключением удаляемого элемента — и возвращается ссылка на новый объект-делегат. При удалении последнего элемента в цепочке *Remove* возвращает null. Заметьте: за раз *Remove* удаляет из цепочки лишь один делегат, а не все делегаты с заданными значениями в полях *_target* и *_methodPtr*.

Пока мы рассматривали тип-делегат *Feedback*, возвращающий значение *void*. Однако его можно было определить и так:

```
public delegate Int32 Feedback(Int32 value);
```

В этом случае в псевдокоде метод *Invoke* выглядел бы примерно так:

```
public Int32 Invoke(Int32 value) {
   Int32 result;
   Delegate[] delegateSet = _invocationList as Delegate[];
   if (delegateSet != null) {
      // Этот массив делегатов указывает делегаты, которые нужно вызвать.
      foreach (Feedback d in delegateSet)
         result = d(value);  // Вызываем каждый делегат.
   } else {
      // Этот делегат определяет один метод,
      // который нужно вызвать по механизму обратного вызова.
      // Вызвать метод обратного вызова для указанного объекта.
      result = _methodPtr.Invoke(_target, value);
      // Предыдущая строка - очень приблизительная копия реального кода.
      // Происходящее на самом деле не поддается
      // иллюстрации средствами C#.
   }
   return result;
}
```

По мере вызова отдельных делегатов массива возвращаемое значение сохраняется в переменной *result*. По завершении цикла переменная *result* содержит результат только последнего вызванного делегата (предыдущие возвращаемые значения отбрасываются); это значение возвращается вызывающему коду, вызвавшему *Invoke*.

Поддержка цепочек делегатов в C#

Компилятор C# облегчает жизнь разработчикам, автоматически предоставляя перегрузку операторов «+=» и «–=» для типов 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 абсолютно идентичный IL-код. Единственная разница в том, что исходный код *ChainDelegateDemo2* проще за счет использования операторов «+=» и «–=» языка C#.

Если вам нужны доказательства идентичности кода, скомпонуйте код и изучите IL-код с помощью ILDasm.exe. Это поможет убедиться, что компилятор C# действительно заменил все операторы «+=» и «–=» вызовами статических методов *Combine* и *Remove* типа *Delegate* соответственно.

Расширенное управление цепочкой делегатов

На этом этапе вы уже поняли, как строить цепочки объектов-делегатов и как вызывать все объекты в таком списке. Поскольку метод *Invoke* типа делегата включает код, итеративно перечисляющий все элементы массива, вызываются все элементы цепочки. Видно, что это очень простой алгоритм. Хотя для большинства сценариев достаточно и такого простого алгоритма, у него много ограничений. Например, отбрасываются все значения, возвращаемые методами обратного вызова, кроме последнего. Остальные значения, возвращаемые методами обратного вызова, невозможно получить при использовании простого алгоритма, но это ограничение — не единственное. А если один из вызванных делегатов генерирует исключение или надолго заблокируется? Поскольку алгоритм последовательно вызывает все делегаты цепочки, «проблема» с одним из делегатов в цепочке остановит вызов остальных. Ясно, что такой алгоритм ненадежен.

Для ситуаций, в которых такого алгоритма недостаточно, класс *MulticastDelegate* предлагает экземплярный метод *GetInvocationList*, позволяющий явно вызвать любой из делегатов цепочки по любому алгоритму, соответствующему вашим потребностям:

```
public abstract class MulticastDelegate : Delegate {
   // Создает массив делегатов, в котором
   // каждый элемент соответствует делегату цепочки.
   public sealed override Delegate[] GetInvocationList();
}
```

Метод *GetInvocationList* принимает объект, производный от *MulticastDelegate*, и возвращает массив ссылок на массив ссылок на *Delegate*, в котором каждая ссылка указывает на один из объектов-делегатов цепочки. Код *GetInvocationList* создает массив и инициализирует его элементами, каждый из которых ссылается на один делегат в цепочке; в конце возвращается ссылка на массив. Если значение поля *_invocationList* равно null, возвращаемый массив содержит один элемент, который ссылается на единственный делегат цепочки, на экземпляр самого делегата.

Можно без труда написать алгоритм, явно вызывающий каждый объект массива:

```
using System;
using System.Text;

// Определяем компонент Light.
internal sealed class Light {
   // Этот метод возвращает состояние объекта Light.
   public String SwitchPosition() {
      return "The light is off";
   }
}

// Определяем компонент Fan.
internal sealed class Fan {
   // Этот метод возвращает состояние объекта Fan.
   public String Speed() {
      throw new InvalidOperationException("The fan broke due to overheating");
   }
}

// Определяем компонент Speaker.
internal sealed class Speaker {
   // Этот метод возвращает состояние объекта Speaker.
   public String Volume() {
      return "The volume is loud";
   }
}

public sealed class Program {

   // Определение делегатов, позволяющих запрашивать состояние компонента.
   private delegate String GetStatus();

   public static void Main() {
      // Объявляем пустую цепочку делегатов.
      GetStatus getStatus = null;

      // Создаем три компонента и добавляем в цепочку делегатов
      // методы для проверки их состояния.
      getStatus += new GetStatus(new Light().SwitchPosition);
      getStatus += new GetStatus(new Fan().Speed);
      getStatus += new GetStatus(new Speaker().Volume);

      // Отображаем сводный отчет, отражающий
      // состояние трех компонентов.
      Console.WriteLine(GetComponentStatusReport(getStatus));
   }

   // Метод, запрашивающий несколько компонентов
   // и возвращающий отчет об их состоянии.
private static String GetComponentStatusReport(GetStatus status) {
```

```
      // Если цепочка пуста, то делать нечего.
      if (status == null) return null;

      // Построить отчет о состоянии.
      StringBuilder report = new StringBuilder();

      // Получаем массив, где каждый элемент – это делегат из цепочки.
      Delegate[] arrayOfDelegates = status.GetInvocationList();

      // Итеративно обрабатываем все делегаты массива.
      foreach (GetStatus getStatus in arrayOfDelegates) {

         try {
            // Получаем строку с состоянием компонента и добавляем в отчет.
            report.AppendFormat("{0}{1}{1}", getStatus(), Environment.NewLine);
         }
         catch (InvalidOperationException e) {
            // Генерируем в отчете 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);
         }
      }

      // Вернуть сводный отчет вызывающему коду.
      return report.ToString();
   }
}
```

Скомпоновав и запустив этот код, мы увидим:

```
The light is off

Failed to get status from Fan.Speed
   Error: The fan broke due to overheating

The volume is loud
```

Упрощение синтаксиса работы с делегатами в C#

Большинство программистов считает, что работать с делегатами слишком обременительно. Причина в очень сложном синтаксисе. Возьмем к примеру такую строку кода:

```
button1.Click += new EventHandler(button1_Click);
```

где *button1.Click* — это метод, выглядящий примерно так:

```
void button1_Click(Object sender, EventArgs e) {
   // Кнопка щелкнута - нужно выполнять соответствующие действия...
}
```

Идея первой строки кода — зарегистрировать адрес метода *button1_Click* в элементе управления «кнопка» с тем, чтобы по щелчку кнопки вызывался этот метод. Большинству программистов кажется совершенно неестественным создавать объект-делегат *EventHandler* только для того, чтобы определить адрес метода *button1_Click*. Однако создание объекта-делегата *EventHandler* необходимо CLR, так как этот объект служит оберткой, которая обеспечивает вызов метода со строгим соблюдением безопасности типов. Обертка также поддерживает вызов экземплярных методов и создание цепочек делегатов. К сожалению, большинство программистов не хочет думать о таких деталях. Программисты предпочли бы записать приведенный выше код так:

```
button1.Click += button1_Click;
```

К счастью, компилятор C# от Microsoft поддерживает несколько способов упрощения синтаксиса при работе с делегатами. Сейчас я об этом расскажу поподробнее, но, прежде чем начать, сделаю одно важное замечание. То, что вы сейчас узнаете, — всего лишь ряд упрощений синтаксиса в C#, которые облегчают задачу программистам по созданию IL-кода, который необходим для нормальной работы CLR и других языков программирования с делегатами. Это также означает, что все сказанное ниже относится исключительно к C# — другие компиляторы скорее всего не поддерживают такой упрощенный синтаксис делегатов.

Упрощенный синтаксис № 1: не нужно создавать объект-делегат

Как я уже показывал, C# позволяет определять имя метода обратного вызова, не создавая объект-делегат, служащий оберткой. 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);
   }
}
```

Здесь статический метод *QueueUserWorkItem* класса *ThreadPool* ожидает ссылку на объект-делегат *WaitCallback*, который в свою очередь содержит ссылку на метод *SomeAsyncTask*. Так как компилятор C# сам «догадывается», что имеется в виду, я могу опустить код создания объекта-делегата *WaitCallback*, что делает код намного читабельнее и понятнее. Конечно, в процессе компиляции компилятор C# автоматически создает IL-код создания нового объекта-делегата *WaitCallback*, а мы получаем удовольствие от возможности упростить синтаксис.

Упрощенный синтаксис № 2: не нужно определять метод обратного вызова

В приведенном выше коде имя метода обратного вызова, *SomeAsyncTask*, передается методу *QueueUserWorkItem* класса *ThreadPool*. C# позволяет встраивать реализацию метода обратного вызова непосредственно в код, а не в сам метод. Например, приведенный выше код можно переписать так:

```
internal sealed class AClass {
   public static void CallbackWithoutNewingADelegateObject() {
      ThreadPool.QueueUserWorkItem(
         delegate(Object obj) { Console.WriteLine(obj); },
         5);
   }
}
```

Заметьте: первый «параметр» метода *QueueUserWorkItem* представляет собой блок кода! Обнаружив ключевое слово *delegate* в месте, где ожидается ссылка на объект-делегат, компилятор C# автоматически определяет в классе новый закрытый метод (здесь это класс *AClass*). Этот новый метод называют анонимным из-за того, что компилятор создает имя метода автоматически и обычно вы даже никогда не узнаете его имя. Однако можно прибегнуть к ILDasm.exe, чтобы изучить сгенерированный компилятором код. После компиляции приведенного выше кода с помощью ILDasm.exe я увидел, что компилятор C# решил назвать этот метод *<CallbackWithoutNewingADelegateObject>b__0*. Но будьте осторожны: никогда не стоит рассчитывать на то, что компилятор создаст именно такое имя метода, потому что в следующей версии компилятора C# алгоритм генерации имени метода может измениться.

Используя ILDasm.exe, также можно заметить, что компилятор C# применяет к этому методу атрибут *System.Runtime.CompilerServices.CompilerGeneratedAttribute* для указания того, что метод создан компилятором, а не программистом. Код из «параметра» размещается в сгенерированном компилятором методе.

Примечание Число операторов или их типов, которые можно использовать в коде обратного вызова (анонимном методе), не ограничено. Однако при создании анонимного метода никак нельзя применить к методу нестандартный (пользовательский) атрибут. Более того — по отношению к методу нельзя применить никаких модификаторов метода (например, *unsafe*). Но это обычно не является проблемой, потому что сгенерированные компилятором анонимные методы практически всегда закрыты, а метод — статический или нестатический, в зависимости от того, к каким членам экземпляра обращается метод. Так что нет никакой нужды применять к методу такие модификаторы, как *public, protected, internal, virtual, sealed, override* и *abstract*.

Наконец, при компиляции приведенного выше кода компилятор C# перепишет код примерно так (комментарии вставил я):

```
internal sealed class AClass {
   // Это закрытое поле создается для кеширования объекта-делегата.
```

```
   // Преимущество: CallbackWithoutNewingADelegateObject не будет создавать
   // новый объект при каждом вызове.
   // Недостаток: объект в кеше никогда не убирается сборщиком мусора.
   [CompilerGenerated]
   private static WaitCallback <>9__CachedAnonymousMethodDelegate1;

   public static void CallbackWithoutNewingADelegateObject() {
      if (<>9__CachedAnonymousMethodDelegate1 == null) {
         // При первом вызове объект-делегат создается и кешируется.
         <>9__CachedAnonymousMethodDelegate1 =
            new WaitCallback(<CallbackWithoutNewingADelegateObject>b__0);
      }
      ThreadPool.QueueUserWorkItem(<>9__CachedAnonymousMethodDelegate1, 5);
   }

   [CompilerGenerated]
   private static void <CallbackWithoutNewingADelegateObject>b__0(Object obj) {
      Console.WriteLine(obj);
   }
}
```

Прототип анонимного метода должен соответствовать типу делегата *WaitCallback*: возвращать *void* и принимать параметр *Object*. Однако я создавал анонимный метод, поэтому определил имя параметра, разместив *(Object obj)* после ключевого слова *delegate*.

Также нужно помнить, что анонимный метод объявляется закрытым; это запрещает любому коду, за исключение кода типа, доступ к методу (хотя отражение позволит узнать, что метод существует). Заметьте также, что анонимный метод определен как статический, потому что код не обращается ни к каким членам экземпляра (да и не сможет этого сделать, так как *CallbackWithoutNewingADelegateObject* — сам по себе статический метод). Однако код может сослаться на любые определенные в классе статические поля или статические методы. Вот пример:

```
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.
         delegate(Object obj) { Console.WriteLine(sm_name+ ": " + obj); },
         5);
   }
}
```

Если бы метод *CallbackWithoutNewingADelegateObject* не был статическим, код анонимного метода мог бы содержать ссылки на члены экземпляра. Но даже в отсутствие в коде ссылок на члены экземпляра компилятор создаст статический анонимный метод, так как он эффективнее, чем экземплярный метод, потому что дополнительный параметр *this* не нужен. Но, если код анонимного метода действительно ссылается на член экземпляра, компилятор создаст нестатический анонимный метод:

```
internal sealed class AClass {
   private String m_name;  // Экземплярное поле.

   // Экземплярный метод.
   public void CallbackWithoutNewingADelegateObject() {
      ThreadPool.QueueUserWorkItem(
         // Код обратного вызова может ссылаться на члены экземпляра.
         delegate(Object obj) { Console.WriteLine(m_name+ ": " + obj); },
         5);
   }
}
```

Упрощенный синтаксис № 3: не нужно определять параметры метода обратного вызова

Обычная ситуация, в которой используется описанный выше упрощенный синтаксис, — когда нужно выполнить определенный код при нажатии кнопки:

```
button1.Click +=
   delegate(Object sender, EventArgs e)
      { MessageBox.Show("The Button was clicked!"); };
```

Удобно иметь возможность встроить код, вызываемый по механизму обратного вызова, не определяя другой метод. Но в этом примере код обратного вызова совсем не ссылается на параметры метода обратного вызова — *sender* и *e*. Если коду обратного вызова не нужны эти параметры, C# позволяет сократить приведенный выше код до следующего:

```
button1.Click +=
   delegate { MessageBox.Show("The Button was clicked!"); };
```

Заметьте: я просто удалил подстроку *(Object sender, EventArgs e)*. Генерируя анонимный метод, компилятор также создает метод, прототип которого в точности соответствует делегату — CLR жестко требует выполнения этого условия для обеспечения безопасности типов. В этом случае компилятор все равно создает анонимный метод, соответствующий делегату *EventHandler* (тип делегата, ожидаемый событием *Click* типа *Button*). Однако в этом примере код анонимного метода не ссылается на параметры делегата.

Если код обратного вызова ссылается на какой-либо из параметров, после ключевого слова *delegate* нужно ввести круглые скобки, типы параметров и имена переменных. Возвращаемый тип все равно определяется типом делегата, и, если возвращаемый тип не *void*, обязательно предусмотреть оператор *return* во встроенном коде обратного вызова.

Упрощенный синтаксис № 4: не нужно вручную создавать обертку локальных переменных класса для передачи их в метод обратного вызова

Я уже показал, как код обратного вызова может ссылаться на другие члены класса. Однако иногда нужно, чтобы код обратного вызова ссылался на локальные

параметры или переменные, имеющиеся в определяемом методе. Вот интересный пример:

```
internal sealed class AClass {
   public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {
      // Некоторые локальные переменные.
      Int32[] squares = new Int32[numToDo];
      AutoResetEvent done = new AutoResetEvent(false);

      // Выполняем некоторые задачи в других потоках.
      for (Int32 n = 0; n < squares.Length; n++) {
         ThreadPool.QueueUserWorkItem(
            delegate(Object obj) {
               Int32 num = (Int32) obj;

               // В других условиях на выполнение этой задачи требуется больше времени.
               squares[num] = num * num;

               // Если это последняя задача, оставляем (не закрываем) поток.
               if (Interlocked.Decrement(ref numToDo) == 0)
                  done.Set();
            },
            n);
      }

      // Ожидаем завершения всех остальных потоков.
      done.WaitOne();

      // Отображаем результаты.
      for (Int32 n = 0; n < squares.Length; n++)
         Console.WriteLine("Index {0}, Square={1}", n, squares[n]);
   }
}
```

Этот пример демонстрирует, как просто в C# выполняется то, что считалось довольно сложной задачей. В показанном выше методе определяется один параметр, *numToDo*, и две локальные переменные, *squares* и *done*. А код обратного вызова в делегате ссылается на эти переменные.

А теперь представьте, что код обратного вызова размещен в отдельном методе (как того требует CLR). Как же передать значения переменных в метод обратного вызова? Единственный способ — определить новый вспомогательный класс, где также определено по полю для каждого значения, которое передается коду обратного вызова. Кроме того, код обратного вызова в этом вспомогательном классе нужно определить как экземплярный метод. В этом случае метод *UsingLocalVariablesInTheCallbackCode* должен создать экземпляр вспомогательного класса, инициализировать поля значениями его локальных переменных и затем создать объект-делегат, связанный с вспомогательным классом и экземплярным методом.

Это очень нудная и чреватая ошибкам работа, которую, конечно же, компилятор C# выполняет за вас автоматически. Показанный выше код компилятор C# переписывает примерно так (комментарии вставил я):

```
internal sealed class AClass {
   public static void UsingLocalVariablesInTheCallbackCode(Int32 numToDo) {

      // Некоторые локальные переменные.
      WaitCallback callback1 = null;

      // Создаем экземпляр вспомогательного класса.
      <>c__DisplayClass2 class1 = new <>c__DisplayClass2();

      // Инициализируем поля вспомогательного класса.
      class1.numToDo = numToDo;
      class1.squares = new Int32[class1.numToDo];
      class1.done = new AutoResetEvent(false);

      // Выполняем некоторые задачи в других потоках.
      for (Int32 n = 0; n < class1.squares.Length; n++) {
         if (callback1 == null) {
            // Новый объект-делегат привязывается к объекту вспомогательного класса
            // и к его анонимному экземплярному методу.
            callback1 = new WaitCallback(
               class1.<UsingLocalVariablesInTheCallbackCode>b__0);
         }

         ThreadPool.QueueUserWorkItem(callback1, n);
      }

      // Ожидаем завершения всех остальных потоков.
      class1.done.WaitOne();

      // Отображаем результаты.
      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 возможных конфликтов и
   // предотвратить доступ из внешнего класса Aclass.
   [CompilerGenerated]
   private sealed class <>c__DisplayClass2 : Object {

      // В коде обратного вызова каждой локальной переменной
      // соответствует одно открытое поле.
      public Int32[] squares;
      public Int32 numToDo;
      public AutoResetEvent done;

      // Открытый конструктор без параметров.
      public <>c__DisplayClass2 { }
```

```
      // Открытый экземплярный метод, содержащий код 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, программистам ничего не стоит начать злоупотреблять анонимными методами в C#. Когда я приступил к изучению анонимных методов, мне потребовалось некоторое время, чтобы привыкнуть к ним. В конце концов, код, который вы пишете в методе, в действительности не относится к нему, что значительно затрудняет отладку и пошаговое выполнение. Честно говоря, я был поражен тем, как замечательно отладчик Visual Studio выполняет пошаговую отладку анонимных методов.

Я установил для себя правило: если мне нужен метод обратного вызова, содержащий больше трех строк кода, я не стану прибегать к анонимным методам, а создам метод вручную и назначу ему имя по собственному усмотрению. Но невзирая ни на что, разумное использование анонимных методов способно значительно повысить производительность работы программиста, а также упростить поддержку кода. Ниже приводится код, в котором использование анонимных методов смотрится исключительно гармоничным и естественным. Без анонимных методов этот код было значительно сложнее писать, читать и поддерживать.

```
// Создание и инициализация массива String.
String[] names = { "Jeff", "Kristin", "Aidan" };

// Извлекаем имена, в которых есть строчная буква 'i'.
Char charToFind = 'i';
names = Array.FindAll(names, delegate(String name)
   { return (name.IndexOf(charToFind) >= 0); });

// Преобразуем все символы строки в верхний регистр.
names = Array.ConvertAll<String, String>(names, delegate(String name)
   { return name.ToUpper(); });

// Сортируем имена.
Array.Sort(names, String.Compare);

// Отображаем результаты.
Array.ForEach(names, Console.WriteLine);
```

Делегаты и отражение

Все примеры использования делегатов, показанные до сих пор, требовали, чтобы разработчик заранее знал прототип метода обратного вызова. Так, если *fb* — переменная, ссылающаяся на делегат *Feedback*, код для вызова делегата будет примерно такой:

```
fb(item); // Параметр item определен как Int32.
```

Как видите, во время кодирования разработчик должен знать, сколько параметров требует метод обратного вызова и тип каждого из них. К счастью, у вас почти всегда есть эта информация, поэтому написать код вроде предыдущего — не проблема.

Но порой у разработчика нет этих сведений на момент компиляции. Достаточно вспомнить пример из главы 10, в котором обсуждался тип *EventSet*. В этом примере словарь поддерживается набором разных типов делегатов. Чтобы событие сработало во время выполнения, производится поиск и вызов делегата из словаря. При компиляции нельзя точно узнать, какой делегат будет вызван и какие параметры передать его методу обратного вызова.

К счастью, тип *System.Delegate* предлагает методы, позволяющие создавать и вызывать делегаты, даже если на момент компиляции нет всех сведений о делегате. Вот эти методы:

```
public abstract class Delegate {
   // Создать делегат «тип», служащий оболочкой заданного статического метода.
   public static Delegate CreateDelegate(Type type, MethodInfo method);
   public static Delegate CreateDelegate(Type type, MethodInfo method,
      Boolean throwOnBindFailure);

   // Создать делегат «тип», служащий оболочкой заданного экземплярного метода.
   public static Delegate CreateDelegate(Type type,
      Object firstArgument, MethodInfo method); // firstArgument means 'this'
   public static Delegate CreateDelegate(Type type,
      Object firstArgument, MethodInfo method, Boolean throwOnBindFailure);

   // Вызываем делегат, передав ему параметры.
   public Object DynamicInvoke(params Object[] args);
}
```

Все версии метода *CreateDelegate* создают новый объект типа — потомок *Delegate*, заданного первым параметром, *type*. Параметр *MethodInfo* указывает, что метод должен вызываться по методу обратного вызова; для получения этого значения нужно использовать API отражения (см. главу 22). Если делегат должен быть оболочкой для экземплярного метода, надо передать в *CreateDelegate* параметр *firstArgument*, указывающий, что объект нужно передать в экземплярный метод в параметре *this* (первый аргумент). Наконец, *CreateDelegate* генерирует исключение *ArgumentException*, если делегат не может связаться с методом, указанным в параметре *method*. Это может произойти, если сигнатура метода *method*, заданная переменной, не соответствует сигнатуре, требуемой делегатом и заданной в парамет-

ре *type*. Однако если передать в параметре *throwOnBindFailure* значение false, вместо этого будет возвращено значение null.

Внимание! У класса *System.Delegate* намного больше перегруженных версий метода *CreateDelegate*, чем показано здесь. Никогда не следует вызывать ни один из этих других методов. В Microsoft даже жалеют, что вообще определили их. Причина в том, что в остальных методах в процессе привязки вызываемый метод идентифицируется с использованием типа *String*, а не *MethodInfo*. Это означает, что возможна неоднозначная привязка, приводящая к непрогнозируемому поведению приложения.

Метод *DynamicInvoke* типа *System.Delegate* позволяет вызвать метод обратного вызова объекта-делегата, передав набор параметров, определяемых во время выполнения. При вызове *DynamicInvoke* его код проверяет совместимость переданных параметров с параметрами, ожидаемыми методом обратного вызова. Если параметры совместимы, выполняется обратный вызов, в противном случае генерируется исключение *ArgumentException*. *DynamicInvoke* возвращает объект, который вернул метод обратного вызова.

Следующий код демонстрирует использование методов *CreateDelegate* и *DynamicInvoke*:

```
using System;
using System.Reflection;
using System.IO;

// Вот 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:" +
         "{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;
      }
```

```
    // Преобразовываем параметр delType в тип делегата.
    Type delType = Type.GetType(args[0]);
    if (delType == null) {
       Console.WriteLine("Invalid delType argument: " + args[0]);
       return;
    }

    Delegate d;
    try {
       // Преобразовываем параметр Arg1 в метод.
       MethodInfo mi = typeof(Program).GetMethod(args[1],
          BindingFlags.NonPublic | BindingFlags.Static);

       // Создаем объект-делегат, служащий оболочкой статического метода.
       d = Delegate.CreateDelegate(delType, mi);
    }
    catch (ArgumentException) {
       Console.WriteLine("Invalid methodName argument: " + args[1]);
       return;
    }

    // Создаем массив, который содержит только параметры,
    // передаваемые методу через объект-делегат.
    Object[] callbackArgs = new Object[args.Length - 2];

    if (d.GetType() == typeof(TwoInt32s)) {
       try {
          // Преобразуем параметры типа String в параметры типа 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)) {
       // Просто копируем параметр типа String.
       Array.Copy(args, 2, callbackArgs, 0, callbackArgs.Length);
    }

    try {
       // Вызываем делегат и показываем результат.
       Object result = d.DynamicInvoke(callbackArgs);
       Console.WriteLine("Result = " + result);
    }
    catch (TargetParameterCountException) {
       Console.WriteLine("Incorrect number of parameters specified.");
    }
}
```

```
   // Это метод обратного вызова, принимающий два параметра Int32.
   private static Object Add(Int32 n1, Int32 n2) {
      return n1 + n2;
   }

   // Это метод обратного вызова, принимающий два параметра Int32.
   private static Object Subtract(Int32 n1, Int32 n2) {
      return n1 - n2;
   }

   // Это метод обратного вызова, принимающий один параметр String.
   private static Object NumChars(String s1) {
      return s1.Length;
   }

   // Это метод обратного вызова, принимающий один параметр String.
   private static Object Reverse(String s1) {
      Char[] chars = s1.ToCharArray();
      Array.Reverse(chars);
      return new String(chars);
   }
}
```

ГЛАВА 16

Обобщения

Разработчикам, использующим объектно-ориентированное программирование, хорошо известны его преимущества. Одно из ключевых преимуществ — возможность повторно использовать код, то есть создавать производный класс, наследующий 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 общеязыковой исполняющей средой (CLR) и языками программирования, который является новой формой повторного использования кода, а именно повторным использованием алгоритма.

По сути, разработчик определяет алгоритм, например сортировку, поиск, замену, сравнение или преобразование, но не указывает типы данных, с которыми тот работает. Поэтому алгоритм может обобщенно применяться к объектам разных типов. Используя готовый алгоритм, другой разработчик должен указать конкретные типы данных, например для алгоритма сортировки — *Int32s*, *String* и т. д., а для алгоритма сравнения — *DateTime*, *Versions*...

Большинство алгоритмов инкапсулированы 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, 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 для определения этого класса выглядит так (это сильно сокращенный вариант):

```
[Serializable]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
   IList, ICollection, IEnumerable {
```

```
    public List();
    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; }
}
```

Выражением <*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-типами* (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* используется в качестве параметра (метод *Add* принимает параметр типа *T*) и возвращаемого значения (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 указан тип данных, а значит, и при определении локальных переменных внутри метода или полей внутри типа.

Итак, после определения обобщенного типа *List*<*T*> готовый обобщенный алгоритм могут использовать другие разработчики, указав точный тип данных, с которым должен работать этот алгоритм. В случае обобщенного типа или метода указанный тип данных называют *аргументом-типом*. Например, разработчик может использовать алгоритм *List*, указав аргумент-тип *DateTime*.

```
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 при 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. А при работе с шаблонами C++ или обобщениями Java разработчику нужен был 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 компиляции или ошибке во время выполнения. В этом примере передача объекта *String* методу *Add* вызвала ошибку компиляции.
- **Более простой и понятный код** Поскольку компилятор обеспечивает безопасность типов, в исходном тексте нужно меньше приведений типов, и такой код проще писать и поддерживать. В последней строке *SomeMethod* разработчику не нужно использовать приведение *(DateTime)*, чтобы присвоить переменной *dt* результат индексатора (при запросе элемента с индексом 0).
- **Повышение производительности** До появления обобщений одним из способов определения обобщенного алгоритма было определение всех его членов так, чтобы они по определению «умели» работать с типом данных *Object*. Чтобы этот алгоритм работал с экземплярами значимого типа, перед вызовом членов алгоритма CLR должна была упаковать этот экземпляр. В главе 5 показано, что упаковка требует выделения памяти в управляемой куче, что приводит к более частому сбору мусора, а это, в свою очередь, ухудшает производительность приложения. Поскольку отныне обобщенный алгоритм можно создавать для работы с конкретным значимым типом, экземпляры значимого типа могут передаваться по значению и CLR не нужно выполнять упаковку. Приведения типов также не нужны (см. предыдущий пункт), поэтому CLR не нужно проверять 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 протестировать работу алгоритмов с объектами значимого и ссылочного типа.

```
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 сбора 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 сбора 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 сбора 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 сбора 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 такой 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, обобщенный алгоритм *List* с типом *Int32* работает гораздо быстрее, чем необобщенный алгоритм *ArrayList* с тем же типом. А ведь разница огромная: десятая доля секунды против целых 2 секунд, то есть в 20 раз быстрее! Кроме того, использование значимого типа (*Int32*) с *ArrayList* требует много операций упаковки, и, как результат, 30 сборок мусора, а в алгоритме *List* сбор мусора вообще не нужен.

Результаты проверки ссылочного типа не столь впечатляющие: временные показатели и число сборок мусора здесь примерно одинаковы. Поэтому в данном случае у обобщенного алгоритма *List* реальных преимуществ нет. Но 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 ухудшает производительность. Подробнее об этом мы поговорим чуть позже в разделе «Инфраструктура обобщений».

Обобщения в библиотеке FCL

Разумеется, обобщения применяются с классами наборов, и в FCL определено несколько таких обобщенных классов. Microsoft рекомендует программистам отказаться от необобщенных классов наборов в пользу новых обобщенных классов наборов по нескольким причинам. Во-первых, необобщенные классы наборов, 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 наборы, предоставляют массу новых возможностей. В табл. 16-1 перечислены обобщенные классы наборов и соответствующие им необобщенные классы наборов.

Табл. 16-1. Обобщенные классы наборов и их необобщенные аналоги

Обобщенный класс набора	Необобщенный класс набора
List<T>	*ArrayList*
Dictionary<TKey, TValue>	*Hashtable*
SortedDictionary<TKey, TValue>	*SortedList*
Stack<T>	*Stack*
Queue<T>	*Queue*
LinkedList<T>	нет

Многие из этих классов наборов используют вспомогательные классы. Классы *Dictionary* и *SortedDictionary* используют класс *KeyValuePair<TKey, TValue>*, обобщенный аналог необобщенного класса *DictionaryEntry*. А класс *LinkedList* применяет класс *LinkedListNode<T>*.

Примечание В своем руководстве для разработчиков Microsoft требует, чтобы обобщенные переменные-параметры назывались *T* или хотя бы начинались с заглавной буквы *T* (например, *TKey* и *TValue*). Заглавная *T* означает «тип» так же, как и заглавная *I* — «интерфейс» (например, *IComparable*).

Классы наборов реализуют много интерфейсов, а объекты, добавляемые в наборы, могут реализовывать интерфейсы, используемые классами наборов для таких операций, как сортировка и поиск. В составе FCL поставляется множество определений обобщенных интерфейсов, поэтому при работе с интерфейсами также доступны преимущества обобщений. В табл. 16-2 перечислены обобщенные интерфейсы и соответствующие им необобщенные интерфейсы.

Табл. 16-2. Обобщенные интерфейсы наборов и их необобщенные аналоги

Обобщенный интерфейс	Необобщенный интерфейс
IList<T>	*IList*
IDictionary<TKey, TValue>	*IDictionary*
ICollection<T>	*ICollection*
IEnumerator<T>	*IEnumerator*
IEnumerable<T>	*IEnumerable*
IComparer<T>	*IComparer*
IComparable<T>	*IComparable*

Новые обобщенные интерфейсы не заменяют необобщенные: во многих ситуациях приходится использовать оба вида интерфейсов. Причина — необходимость обратной совместимости. Например, если бы класс *List<T>* реализовывал только интерфейс *IList<T>*, в коде нельзя было бы рассматривать объект *List< DateTime>* как *IList*.

Также отмечу, что класс *SystemArray*, базовый для всех типов массивов, поддерживает много 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 : ICloneable, IList, ICollection, IEnumerable {
   public static void  Sort<T>(T[] array);
   public static void  Sort<T>(T[] array, IComparer<T> comparer);

   public static Int32 BinarySearch<T>(T[] array, T value);
   public static Int32 BinarySearch<T>(T[] array, T value,
      IComparer<T> comparer);
   ...
}
```

В следующем коде показано применение нескольких из этих методов.

```
public static void Main() {
   // Создание и инициализация массива байт.
   Byte[] byteArray = new Byte[] { 5, 1, 4, 2, 3 };

   // Вызов алгоритма сортировки Byte[].
   Array.Sort<Byte>(byteArray);

   // Вызов алгоритма двоичного поиска Byte[].
   Int32 i = Array.BinarySearch<Byte>(byteArray, 1);
   Console.WriteLine(i);  // Отображает "0".
}
```

Библиотека Power Collections от Wintellect

По заказу Microsoft компания Wintellect создала библиотеку Power Collections, основная цель которой — сделать некоторые классы наборов из STL-библиотеки из C++ доступными для CLR. Классы библиотеки Power Collections распространяются бесплатно. Подробнее см. на Web-сайте *http://Wintellect.com*. Эти классы наборов сами по себе являются обобщенными, и в них широко используют обобщения. В табл. 16-3 приведен неполный список классов наборов из библиотеки Power Collections.

Табл. 16-3. Обобщенные классы наборов из библиотеки Power Collections от Wintellect

Класс набора	Описание
BigList<T>	Набор упорядоченных объектов *T*. Очень эффективен, когда объектов больше 100
Bag<T>	Набор неупорядоченных объектов *T*. Этот набор хешируется и допускает дублирование объектов
OrderedBag<T>	Набор упорядоченных объектов *T*. Допускается дублирование объектов
Set<T>	Набор неупорядоченных элементов *T*. Дублирование не поддерживается
OrderedSet<T>	Набор упорядоченных элементов *T*. Дублирование не поддерживается
Deque<T>	Очередь с двусторонним доступом. Похожа на список, но более эффективна при добавлении/удалении элементов в начале
OrderedDictionary <TKey,TValue>	Словарь с упорядоченными однозначными ключами
MultiDictionary <TKey,TValue>	Словарь с многозначными ключами. Ключи хешируются, допускается дублирование, элементы не упорядочены
OrderedMultiDictionary <TKey,TValue>	Словарь с упорядоченными многозначными ключами (также в отсортированном порядке). Допускается дублирование ключей

Инфраструктура обобщений

Над реализацией обобщений в CLR долго трудилось много специалистов. Для обеспечения работы обобщений Microsoft нужно было сделать следующее.

- Создать новые IL-команды, работающие с аргументами-типами.
- 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 другие), 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-типами, 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. Также пришлось определить новые предоставляющие информацию отражения члены, что позволило разработчикам создавать обобщенные определения типов и методов во время исполнения.
- Изменить отладчик, чтобы он отображал и работал с обобщенными типами, членами, полями и локальными переменными.

- Изменить механизм IntelliSense в Microsoft Visual Studio для отображения 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 при проектировании и создании обобщенных алгоритмов или 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-типами*. Типы с обобщенными параметрами-типами также считаются типами, причем для каждого из них CLR создает внутренний объект-тип. Это справедливо для ссылочных типов (классов), значимых типов (структур), интерфейсов и делегатов. Но тип с обобщенными параметрами-типами называют открытым типом, а в CLR запрещено конструирование экземпляров открытых типов (как и экземпляров типов-интерфейсов).

При ссылке на обобщенный тип в коде можно определить набор обобщенных аргументов-типов. Если всем аргументам-типам определенного типа передать действительные типы данных, то такой тип будут называть *закрытым*. CLR разрешает создание экземпляров закрытых типов. Но в коде, ссылающемся на обобщенный тип, можно не определять все обобщенные аргументы-типы. 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> {
}

public static class Program {
   public static void Main() {
      Object o = null;

      // Dictionary<,> – это открытый тип с двумя параметрами-типами.
      Type t = typeof(Dictionary<,>);

      // Попытка создания экземпляра этого типа (неудачная).
      o = CreateInstance(t);
      Console.WriteLine();

      // DictionaryStringKey<> – это открытый тип с одним параметром-типом.
      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.Write("Created instance of {0}", t.ToString());
      }
      catch (ArgumentException e) {
         Console.WriteLine(e.Message);
      }
      return o;
   }
}
```

Скомпилировав и выполнив этот код, вы увидите:

```
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* генерирует 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* типа, то есть число необходимых для него параметров-типов. Например, арность класса *Dictionary* равна 2, потому что требуется определить типы *TKey* и *TValue*. Арность класса *DictionaryStringKey* — 1, так как требуется указать лишь один тип — *TValue*.

Также замечу, что CLR размещает статические поля типа в самом объекте-типе (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 кон-

структор для обобщенного типа, чтобы аргументы-типы соответствовали определенным критериям. Например, так определяется обобщенный тип, используемый только с перечислимыми типами.

```
internal sealed class GenericTypeThatRequiresAnEnum<T> {
   static GenericTypeThatRequiresAnEnum() {
      if (!typeof(T).IsEnum) {
         throw new ArgumentException("T must be an enumerated type");
      }
   }
}
```

В CLR есть функция под названием *ограничения* — это более удачный способ определения обобщенного типа с указанием допустимых для него аргументов-типов. Но подробнее о них чуть позже. К сожалению, эта функция не позволяет ограничить аргументы-типы 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-типов в CLR определяется новый объект-тип, производный от того же типа, что и обобщенный тип. Например, *List<T>* является производным от *Object*, поэтому *List<String>* и *List<Guid>* тоже производные от *Object*. Аналогично, *DictionaryStringKey<TValue>* — производный от *Dictionary<String, TValue>*, поэтому *DictionaryStringKey<Guid>* также производный от *Dictionary<String, Guid>*. Понимание того, что определение аргументов-типов не имеет ничего общего с иерархиями наследования, позволяет разобраться, какие приведения типов допустимы, а какие нет.

Например, пусть класс *Node* связного 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 связного 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 выше классе *Node* поле *m_next* должно ссылаться на другой узел, поле *m_data* которого содержит тот же тип данных. Это значит, что узлы связного списка должны иметь одинаковый (или производный) тип данных. Например, нельзя использовать класс *Node* для создания связного 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 связный список с произвольным типом данных у каждого узла. Приведу 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 связного 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());
}
```

Проблемы с идентификацией и тождеством 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-типы. Например, если нужно упростить следующий код:

```
List<DateTime> dt = new List<DateTime>();
```

некоторые разработчики сначала определят класс, вот так:

```
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 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>*. Но методу, который должен принимать *List<DateTime>*, можно передать *DateTimeList*, потому что *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 влияя на эквивалентность типов. Для этого в начало файла с исходным текстом нужно добавить старую добрую директиву *using*, вот так:

```
using DateTimeList = System.Collections.Generic.List<System.DateTime>;
```

Здесь директива *using* просто определяет символ *DateTimeList*. При компиляции кода компилятор заменяет все *DateTimeList* на *System.Collections.GenericList <System.DateTime>*. Таким образом, разработчики могут использовать упрощенный синтаксис, не влияя на смысл кода и тем самым сохраняя идентификацию и тождество типов. И теперь при выполнении следующей строки кода *sameType* инициализируется *true*.

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

«Распухание» кода

При JIT-компиляции метода, в котором используются обобщенные параметры-типы, CLR подставляет в IL-код метода указанные аргументы-типы, а затем создает 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), и в итоге существенно увеличивается рабочий набор приложения и производительность ухудшается.

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-типом, 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 AppDomain), CLR компилирует методы для *List<DateTime>* всего один раз. Это существенно снижает распухание 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 (32-разрядное значение в 32-разрядной и 64-разрядное значение в 64-разрядной версии Windows) на объекты в куче, а все указатели на объекты обрабатываются одинаково.

Но если аргументы-типы имеют значимый тип, 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* — 32-разрядные значения), CLR все равно не может использовать для них один код, потому что для обработки этих значений могут применяться различные машинные команды.

Обобщенные интерфейсы

Конечно же, основное преимущество обобщений — в их способности 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 упаковке и потере безопасности типов в процессе компиляции. Это сильно сузило бы применение обобщенных типов. Вот почему CLR поддерживает обобщенные интерфейсы. Ссылочный и значимый тип реализуют обобщенный интерфейс путем задания аргументов-типов, или же любой тип реализует обобщенный интерфейс, не определяя аргументы-типы. Рассмотрим несколько примеров.

Вот определение обобщенного интерфейса из библиотеки FCL (из пространства имен *System.Collections.Generic*):

```
public interface IEnumerator<T> : IDisposable, IEnumerator {
   T Current { get; }
}
```

А этот тип реализует данный обобщенный интерфейс и задает аргументы-типы. Обратите внимание, что объект *Triangle* может перечислять набор объектов *Point*, а тип свойства *Current* — *Point*.

```
internal sealed class Triangle : IEnumerator<Point> {
   private Point[] m_vertices;

   // Тип свойства Current в IEnumerator<Point> - Point.
   Point Current { get { ... } }
}
```

Теперь рассмотрим пример типа, реализующего тот же обобщенный интерфейс, но без задания аргументов-типов:

```
internal sealed class ArrayEnumerator<T> : IEnumerator<T> {
   private T[] m_array;

   // Тип свойства Current в IEnumerator<T> - T.
   T Current { get { ... } }
}
```

Заметьте: 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 14.

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 говорилось в главе 15, делегат — это просто определение класса с помощью четырех методов: конструктора и методов *Invoke*, *BeginInvoke* и *EndInvoke*. При определении типа-делегата с параметрами-типами, компилятор определяет методы класса делегата, а параметры-типы применяются ко всем методам, параметры и возвращаемые значения которых относятся к указанному параметру-типу.

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);
}
```

В составе FCL есть много обобщенных типов-делегатов. Большинство из них используется при работе с наборами. Например:

```
// Обычно служит для выполнения действия над элементом набора.
public delegate void Action<T>(T obj);

// Обычно используется при сравнении элементов двух наборов в целях сортировки.
public delegate Int32 Comparison<T>(T x, T y);

// Обычно служит для преобразования типа элемента набора.
public delegate TOutput Converter<TInput, TOutput>(TInput input);

// Обычно применяется, чтобы узнать, прошел ли элемент набора тест.
public delegate Boolean Predicate<T>(T obj);
```

А следующий обобщенный делегат, поставляемый с FCL, используется для событий и обсуждается в главе 10:

```
public delegate void EventHandler<TEventArgs>(
   object sender, TEventArgs e) where TEventArgs : EventArgs;
```

Оператор *where* в этом примере называют *ограничением*. Подробнее об ограничениях чуть позже в этой главе.

Обобщенные методы

При определении обобщенного ссылочного и значимого типа или интерфейса все методы, определенные в этих типах, могут ссылаться на любой параметр-тип, заданный этим типом. Параметр-тип может использоваться как параметр метода, возвращаемое значение метода или как заданная внутри него локальная переменная. Но CLR также позволяет методу задавать собственные параметры-типы, которые могут использоваться в качестве параметров, возвращаемых значений или локальных переменных. Вот немного искусственный пример типа, определяющего параметр-тип, и метода с собственным параметром-типом:

```
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 собственный параметр-тип (*T*), а в методе *Converter* — собственный параметр-тип (*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. Наличие параметров-типов и параметров 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 возможных нарушений безопасности типов. Эта особенность параметров *out/reg* обсуждается в главе 8. В сущности, именно поэтому методы *Exchange* и *CompareExchange* класса *Interlocked* поддерживают обобщенную перегрузку:

```
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;
}
```

Логический вывод обобщенных методов и типов

Синтаксис обобщений в C# со всеми его знаками «меньше» и «больше» приводит в замешательство многих разработчиков. Для упрощения создания, чтения и работы с кодом в компиляторе C# имеется *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 показан в следующем коде:

```
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.
}
```

Заметьте: в этом коде в вызовах *Swap* аргументы-типы не задаются с помощью знаков «меньше» и «больше». В первом вызове *Swap* компилятор C# сумел установить, что тип *n1* и *n2* — *Int32*, поэтому он 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: «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» («ошибка CS0411: аргументы-типы для метода *Program.Swap<T>(ref T, ref T)* не могут быть выведены. Попробуйте явно задать аргументы-типы»).

В типе могут определяться несколько методов так, что один из них будет принимать конкретный тип данных, а другой — обобщенный параметр-тип, 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 вызове компилятор может вызвать либо метод *Display*, принимающий *String*, либо обобщенный метод *Display* (заменяя *T* на *String*). Но компилятор C# всегда выбирает явное, а не обобщенное соответствие, поэтому генерирует вызов необобщенного метода *Display*, принимающего *String*. Во втором вызове компилятор не может вызвать необобщенный метод *Display*, принимающий *String*, поэтому он должен вызвать обобщенный метод *Display*. Кстати, это очень удачно, что компилятор всегда выбирает более явное соответствие. Ведь, если бы компилятор выбрал обобщенный метод *Display*, тот вызвал бы *ToString*, возвращающий *String*, что привело бы к бесконечной рекурсии.

Третий вызов *Display* задает обобщенный аргумент-тип, *String*. Для компилятора это означает, что вместо попытки вывести аргументы-типы он должен использовать аргументы-типы, которые указаны. В данном случае компилятор также считает, что непременно нужно вызвать обобщенный метод *Display*, поэтому он его и вызывает. Внутренний код обобщенного метода *Display* вызовет *ToString* для переданной ему строки, а полученная в результате строка затем передается необобщенному методу *Display*.

Обобщения и другие члены

В C# у свойств, индексаторов, событий, методов операторов, конструкторов и деструкторов не может быть параметров-типов. Но их можно определить в обобщенном типе с тем, чтобы в коде этих членов использовать параметры-типы этого типа.

C# не поддерживает задание собственных обобщенных параметров-типов у этих членов, поскольку создатели C# из Microsoft считают, что разработчикам вряд ли потребуется использовать эти члены в качестве обобщенных. Вдобавок, чтобы эти члены могли применяться как обобщенные, для C# пришлось бы разработать специальный синтаксис, что довольно затратно. Например, при использовании в коде оператора «+» компилятор вызывает метод перегрузки оператора. В коде, где есть оператор «+», нельзя указать никакие аргументы-типы.

Верификация и ограничения

В процессе компиляции обобщенного кода компилятор C# анализирует его, убеждаясь, что он будет 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 присвоения переменных и несколько вызовов методов. Он работает с любым типом *T* — ссылочным, значимым, перечислимым, типом-интерфейсом или типом-делегатом, существующим типом или типом, который определят в будущем, — потому что любой тип поддерживает присвоения 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 выше код появится сообщение об ошибке: «error CS0117: 'T' does not contain a definition for 'CompareTo'» («ошибка CS0117: *T* не содержит определение метода *CompareTo*»).

Поэтому может показаться, что при использовании обобщений можно лишь объявлять переменные обобщенного типа, назначать переменные, вызывать методы, определенные *Object*, и все! Но ведь в таком случае от обобщений пользы мало. К счастью, компиляторы и CLR поддерживают механизм под названием *ограничения* (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 серым):

```
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-тип, удовлетворяющий этим ограничениям. Например, при компиляции следующего кода появляется сообщение: «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)'|» («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)*»).

```
private static void CallMin() {
   Object o1 = "Jeff", o2 = "Richter";
   Object oMin = Min<Object>(o1, o2);  // Ошибка CS0309.
}
```

Компилятор выдает эту ошибку, потому что *System.Object* не реализует интерфейс *IComparable<Object>*. Честно говоря, *System.Object* вообще не реализует никаких интерфейсов.

Вы получили общее представление об ограничениях и их работе. Познакомимся с ними поближе. Ограничения можно применять к параметрам-типам как обобщенных типов, так и обобщенных методов (как показано в методе *Min*). CLR не поддерживает перегрузку по именам параметров-типов или 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 AType<T>, у которого нет ограничений.
internal sealed class AType<T> where T : IComparable<T> {}

// Ошибка: конфликт с 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 M<T>, у которого нет ограничений.
   private static void M<T>() where T : IComparable<T> {}

   // Ошибка: конфликт с M<T1, T2>.
   private static void M<T3, T4>() {}
}
```

При переопределении виртуального обобщенного метода в переопределяющем методе нужно задавать то же число параметров-типов, 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-типов, но может переименовывать параметры-типы. Аналогично, при реализации интерфейсного метода в нем должно задаваться то же число параметров-типов, что и в интерфейсном методе, причем эти параметры-типы 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>()
      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: «Error CS0460: Constraints for override and explicit interface implementation methods are inherited from the base method so cannot be specified directly» («ошибка CS0460: ограничения для методов интерфейсов с переопределением и явной реализацией наследуются от базового метода и поэтому не могут быть заданы явно»). Если из метода *M<T3, T4>* класса *Derived* убрать две строки *where*, код успешно скомпилируется. Заметьте: разрешается переименовывать параметры-типы (в этом примере *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-типам. К параметру-типу могут применяться следующие ограничения: *основное*, *дополнительное* и/или *ограничение конструктора*. Речь о них пойдет в следующих трех разделах.

Основные ограничения

В параметре-типе можно задать не более одного основного ограничения. Основным ограничением может быть ссылочный тип, указывающий на неизолированный 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-тип будет либо того же типа, что и ограничение, либо производного от него типа. 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-тип *T* наложено основное ограничение *Stream* (из пространства имен *System.IO*), сообщающее компилятору, что код, использующий *PrimaryConstraintOfStream*, должен задавать аргумент-тип *Stream* или производный от него тип (например, *FileStream*). Когда параметр-тип не задает основное ограничение, автоматически задается *System.Object*. Но, если в исходном тексте явно задать *System.Object*, компилятор C# выдаст ошибку: «error CS0702: Constraint cannot be special class 'object'» («ошибка CS0702: ограничение не может быть конкретным классом 'object'»).

Есть два особых основных ограничения: *class* и *struct*. Ограничение *class* гарантирует компилятору, что указанный аргумент-тип будет ссылочного типа. 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 T должен быть ссылочного типа.
   }
}
```

В этом примере присвоение *temp* значения *null* допустимо, потому что известно, что *T* — ссылочного типа, а любая переменная ссылочного типа может быть равна *null*. При отсутствии у *T* ограничений этот код бы не скомпилировался, потому что *T* мог бы быть значимого типа, а переменные значимого типа нельзя приравнять к *null*.

Ограничение *struct* гарантирует компилятору, что указанный аргумент-тип будет значимого типа. Этому ограничению удовлетворяют все значимые типы, и перечисления тоже. Но компилятор и CLR рассматривают любой значимый тип *System.Nullable<T>* как особый, а значимые типы с поддержкой значения null не подходят под это ограничение. Это объясняется тем, что для параметра-типа *Nullable<T>* действует ограничение *struct*, а CLR запрещает такие рекурсивные типы, как *Nullable<Nullable<T>>*. Значимые типы с поддержкой значения null обсуждаются в главе 18.

Вот пример класса, где параметр-тип ограничивается с помощью *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* — значимого типа, а у всех значимых типов неявно есть открытый конструктор без параметров. Если бы *T* был не ограничен, ограничен ссылочным типом или *class*, этот код не скомпилировался бы, потому что у некоторых ссылочных типов нет открытых конструкторов без параметров.

Дополнительные ограничения

В параметре-типе можно задать несколько дополнительных ограничений или не задавать их вообще. При задании ограничения типа-интерфейса компилятору гарантируется, что указанный аргумент-тип будет типом, реализующим этот интерфейс. А так как можно задать несколько ограничений интерфейса, в аргументе-типе должен указываться тип, реализующий все ограничения интерфейса (и все основные ограничения, если они есть). Подробнее об ограничениях интерфейса см. главу 14.

Другой тип дополнительных ограничений называют *ограничением параметра-типа*. Оно используется гораздо реже, чем ограничение интерфейса, и позволяет обобщенному типу или методу указать, что между указанными аргументами-типами должны быть определенные отношения. К параметру-типу может применяться 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-типом *TBase*. Это значит, что какой бы аргумент-тип ни был задан для *T*, он должен быть совместим с аргументом-типом, заданным для *TBase*. В следующем методе показаны допустимые и недопустимые вызовы *ConvertIList*.

```
private static void CallingConvertIList() {
   // Создает и инициализирует 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);// Error
}
```

В первом вызове *ConvertIList* компилятор проверяет, чтобы *String* был совместим с *Object*. Поскольку *String* является производным от *Object*, первый вызов удовлетворяет ограничению параметра-типа. Во втором вызове *ConvertIList* компилятор проверяет, чтобы *String* был совместим с *IComparable*. Поскольку *String* реализует интерфейс *IComparable*, второй вызов соответствует ограничению параметра-типа. В третьем вызове *ConvertIList* компилятор проверяет, чтобы *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 *String* совместим сам с собой. В пятом вызове *ConvertIList* компилятор проверяет, чтобы *String* был совместим с *Exception*. Так как *String* не совместим с *Exception*, пятый вызов не соответствует ограничению параметра-типа, и компилятор возвращает ошибку: «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.Collectons.Generic.IList<T>'» («ошибка CS0309: тип *string* должен быть преобразован в *System.Exception*, чтобы он мог использоваться как параметр *T* в обобщенном коде или методе *SomeType.ConvertIList<T,TBase>(System.Collectons.Generic.IList<T>*»).

Ограничения конструктора

В параметре-типе можно задавать не более одного ограничения конструктора. Ограничение конструктора указывает компилятору, что указанный аргумент-тип будет неабстрактного типа, реализующего открытый конструктор без параметров. Заметьте: 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-типов 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* по отношению 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 аргумент-тип.

Иногда разработчики предпочитают объявлять параметр-тип с помощью ограничения конструктора, при котором сам конструктор принимает различные параметры. На сегодняшний день CLR (и, как следствие, компилятор C#) поддерживают только конструкторы без параметров. По мнению специалистов компании Microsoft, в большинстве случаев этого вполне достаточно, и я с ними полностью согласен.

Другие вопросы верификации

В оставшейся части этого раздела я рассмотрю 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 быть любого типа и успех приведения типов 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 используется 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.
}
```

Присвоение 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 - Cannot convert null to type parameter 'T'
                       // because it could be a value type…
                       // (Ошибка CS0403 - нельзя преобразовать null в параметр-тип T,
```

```
                                    // потому что T может быть значимого типа.)
}
```

Так как *T* не ограничен, он может быть значимого типа, а приравнять переменную значимого типа к *null* нельзя. Если же *T* был бы 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* — ссылочного типа, и обнуляющий все биты переменной *temp*, если *T* — значимого типа.

Сравнение переменной обобщенного типа с null

Сравнение переменной обобщенного типа с *null* с помощью операторов «==» и «!=» допустимо, не зависимо 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 *T* не ограничен, он может быть ссылочного или значимого типа. Во втором случае *obj* нельзя приравнять *null*. Обычно в таком случае компилятор C# должен выдать ошибку. Но этого не происходит — код успешно компилируется. При вызове этого метода с использованием аргумента-типа значимого типа JIT-компилятор, обнаружив, что выражение *if* никогда не равно *true*, просто не создаст машинный код для оператора *if* и кода в фигурных скобках. Если бы я использовал оператор «!=», JIT-компилятор также не сгенерировал бы код для оператора *if* (поскольку его значение всегда *true*), но сгенерировал бы код из 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-тип 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

T ограничен ссылочным типом, перегружающим метод *operator*==, компилятор сгенерирует вызовы этого метода при виде оператора ==. Ясно, что все вышесказанное относится и к оператору !=.

При написании кода для сравнения элементарных значимых типов (*Byte*, *Int32*, *Single*, *Decimal* и так далее) компилятор C# сгенерирует код правильно, а для других значимых типов генерировать код для сравнений он не умеет. Поэтому, если у *T* метода *ComparingTwoGenericTypeVariables* есть ограничение *struct*, компилятор выдаст ошибку. А ограничивать параметр-тип 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 уже не будет 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 рассказал, как C# обрабатывает элементарные типы — *Byte*, *Int16*, *Int32*, *Int64*, *Decimal* и другие. В частности, я отметил, что C# умеет интерпретировать операторы (например +, -, * и /), применяемые к элементарным типам. Но эти операторы нельзя использовать с переменными обобщенного типа, потому что во время компиляции компилятор не знает их тип. Получается, что вы не сможете написать 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 *default(T)*, чтобы *sum* и *n* инициализировались нулем. Но при компиляции кода появились три сообщения об ошибке:

- error CS0019: Operator '<' cannot be applied to operands of type 'T' and 'T' (ошибка CS0019: оператор «<» нельзя применять к операндам типа *T* и *T*).
- error CS0023: Operator '++' cannot be applied to operand of type 'T' (ошибка CS0023: оператор «++» нельзя применять к операнду типа *T*).
- error CS0019: Operator '+=' cannot be applied to operands of type 'T' and 'T' (ошибка CS0019: оператор «+=» нельзя применять к операндам типа *T* и *T*).

Это существенно ограничивает поддержку обобщений в CLR-среде, и многие разработчики (особенно из научных и математических кругов) испытали глубокое разочарование. Многие пытались создать методы, призванные обойти это ограничение с помощью отражения (см. главу глава 22), перегрузку оператора и т. п. Но все эти решения сильно снижают производительность или ухудшают читабельность кода. Остается надеяться, что в следующих версиях CLR и компиляторов Microsoft устранит этот недостаток.

ГЛАВА 17

Нестандартные атрибуты

В этой главе я расскажу об одной из самых новаторских особенностей Microsoft .NET Framework — *нестандартных*, или *пользовательских атрибутах* (custom attributes). Они позволяют использовать новую парадигму — *декларативное программирование*. Это программирование подразумевает использование данных, а не написание кода для того, чтобы заставить приложение или компонент выполнить что-либо. (Написание исходного кода иногда называют *императивным программированием*.)

Пример декларативного программирования — создание текстового файла и явное написание HTML-тегов, используя такой редактор, как Notepad.exe. В такой ситуации HTML-теги выступают командами, которые обрабатываются Интернет-браузером и позволяют последнему правильно размещать текст в окне. HTML-теги определяют, как программа (Web-страница) должна отображаться и вести себя, и именно программист решает, где и какие теги использовать. В отличие от меня, многие матерые программисты не считают программирование HTML «настоящим» программированием.

Нестандартные атрибуты позволяют одновременно использовать декларативное и императивное (исходный код на C#) программирование. Комбинирование двух типов программирования предоставляет массу возможностей программисту, а также позволяет в очень сжатой форме выражать свои намерения. Я считаю, что со временем декларативное программирование получит намного большее распространение, чем сейчас. Примеры подобных технологий уже сейчас можно увидеть в Microsoft ASP.NET и Microsoft Windows Communication Foundation. Даже в Windows Communication Foundation программисты могут разрабатывать пользовательский интерфейс, объявляя (декларируя) его разметку и поведение, используя язык разметки XAML.

Нестандартные атрибуты позволяют определять информацию практически для любых элементов таблиц метаданных. Эту расширяемую метаинформацию можно получить во время выполнения для динамического изменения хода выполнения программы. Различные технологии .NET Framework (Windows Forms, Web Forms, Web-сервисы XML и т. д.) применяют нестандартные атрибуты, позволяя разработчикам легко реализовывать в коде (но без кодирования) свои замыслы. Разработчики, ориентирующиеся на .NET Framework, должны четко понимать нестандартные атрибуты.

Применение нестандартных атрибутов

Такие атрибуты, как *public*, *private*, *static* и другие, применяются к типам и членам. Думаю, польза их очевидна. А не было бы еще полезней определять собственные атрибуты? Что, если при определении типа как-то указать, что он может быть сериализован? Или применить некоторый атрибут к методу, указывая, что перед обращением к нему нужно проверить права доступа?

Конечно, было бы удобно создавать и применять определяемые пользователем атрибуты к типам и методам, но компилятор должен понимать эти атрибуты и заносить соответствующую информацию в метаданные. Поскольку поставщики компиляторов предпочитают не открывать свой исходный код, Microsoft предложила свой способ работы с атрибутами, определяемыми пользователем. Этот исключительно мощный механизм — *нестандартные атрибуты* — полезен как в период разработки, так и во время выполнения приложений. Любой разработчик может определить и задействовать нестандартные атрибуты, а все CLR-совместимые компиляторы должны их распознавать и генерировать для них результирующие метаданные.

Первое, что нужно понять: нестандартные атрибуты — это просто способ связать дополнительную информацию с некоторой сущностью. Компилятор помещает эту информацию в метаданные управляемого модуля. Большинство атрибутов ничего не значит для компилятора — он просто обнаруживает их в исходном коде и создает соответствующие метаданные.

Библиотека классов .NET Framework (FCL) включает определения сотен нестандартных атрибутов, которые вы вправе задействовать в своем коде. Вот несколько примеров.

- Применение атрибута *DllImport* к методу говорит CLR, что метод реализован в виде неуправляемого кода, размещенного в указанной DLL-библиотеке.
- Применение атрибута *Serializable* к типу информирует форматировщики сериализации, что поля экземпляра могут сериализоваться и десериализоваться.
- Применение атрибута *AssemblyVersion* к сборке служит для задания номера версии этой сборки.
- Применение атрибута *Flags* к перечислимому типу приводит к тому, что этот тип ведет себя, как набор битовых флагов.

Следующий код на C# содержит несколько атрибутов. В C# нестандартный атрибут применяется путем размещения имени атрибута, заключенного в квадратные скобки, непосредственно перед целым классом, объектом и т. п. Не важно, что этот код делает, я хочу лишь показать, как выглядят атрибуты.

```
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*, *MarshalAs* — к полю *CSDVersion*, *DllImport* — к методу *GetVersionEx*, а *In* и *Out* — к параметру *ver* метода *GetVersionEx*. В каждом языке свой синтаксис применения нестандартных атрибутов. Скажем, в Microsoft Visual Basic .NET вместо квадратных скобок используются угловые (<, >).

CLR позволяет применять атрибуты практически ко всему, что может быть представлено метаданными. Чаще всего атрибуты применяются к элементам таблиц определений TypeDef (классы, структуры, перечисления, интерфейсы и делегаты), MethodDef (конструкторы), ParamDef, FieldDef, PropertyDef, EventDef, AssemblyDef и ModuleDef. Реже атрибуты применяются к ссылочным таблицам AssemblyRef, ModuleRef, TypeRef и MemberRef. В частности, C# позволяет применять нестандартные атрибуты только к исходному коду, где определены такие сущности, как сборки, модули, типы (классы, структуры, перечисления, интерфейсы и делегаты), поля, методы (в том числе конструкторы), параметры методов, значения, возвращаемые методами, свойства, события и обобщенный параметр-тип.

C# позволяет задавать префикс, указывающий сущность, к которой применяется атрибут. В следующем примере показаны все возможные префиксы. Во многих случаях при отсутствии префикса компилятор может определить, к чему относится атрибут (как в предыдущем примере). Префиксы, выделенные курсивным начертанием, являются обязательными.

```
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 обобщенному параметру-типу.

   [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 методу-аксессору get.
      get { return null; }
   }

   [event:  SomeAttr]            // Применяется к событию.
   [field:  SomeAttr]            // Применяется к созданному компилятором полю.
   [method: SomeAttr]            // Применяется к созданным компилятором методам
                                 // add и remove.
   public event EventHandler SomeEvent;
}
```

Теперь, зная, как применять нестандартные атрибуты, давайте разберемся, что они собой представляют. Нестандартный атрибут — это просто один экземпляр некоторого типа. Чтобы нестандартный атрибут соответствовал общеязыковой спецификации (CLS), он должен прямо или косвенно наследовать типу *System.Attribute*. C# допускает только CLS-совместимые атрибуты. В документации к .NET Framework SDK можно обнаружить определения следующих типов из предыдущего примера: *StructLayoutAttribute*, *MarshalAsAttribute*, *DllImportAttribute*, *InAttribute* и *OutAttribute*. Все они определены в пространстве имен *System.RuntimeInteropService*, хотя классы атрибутов могут определяться в любом пространстве имен. Вы можете заметить, что все перечисленные типы порождаются из *SystemAttribute*, как и должно быть для CLS-совместимых атрибутов.

Примечание Компилятор C# позволяет опустить суффикс *Attribute* при определении атрибута, что упрощает программирование и повышает читабельность кода. В примерах я активно использую эту возможность. В частности, в исходном коде я задаю *[DllImport(…)]* вместо *[DllImportAttribute(…)]*.

Как я сказал, любой атрибут является экземпляром некоторого типа. Такой тип должен иметь открытый конструктор для создания его экземпляров. Следовательно, синтаксис применения атрибута к некоторой сущности аналогичен вызову одного из конструкторов типа. Кроме того, используемый язык может поддерживать специальный синтаксис определения открытых полей или свойств атрибутного типа. Рассмотрим пример. Вернемся к приложению, в котором атрибут *DllImport* применяется к методу *GetVersionEx*:

```
[DllImport("Kernel32", CharSet=CharSet.Auto, SetLastError=true)]
```

Довольно странный синтаксис для вызова конструктора. Согласно описанию типа *DllImportAttribute* в документации его конструктор требует единственного параметра типа *String*. В данном примере этим параметром передается строка «Kernel32». Параметры конструктора называются *позиционными* (positional parameters) и являются обязательными: при применении атрибута обязательно указывается каждый такой параметр.

А как насчет еще двух «параметров»? Показанный особый синтаксис позволяет установить любые открытые поля или свойства объекта *DllImportAttribute* после его создания. В нашем примере, когда создается объект *DllImportAttribute* и конструктору передается строка «Kernel32», открытым экземплярным полям *CharSet* и *SetLastError* этого объекта присваиваются *CharSetAuto* и *true*. «Параметры», устанавливающие поля или свойства, называются *именованными* и являются необязательными. Чуть позже я объясню, как инициируется конструирование экземпляра *DllImportAttribute*.

Замечу также, что к одной сущности можно применить несколько атрибутов. Так, к параметру *ver* метода *GetVersionEx* применяются атрибуты *In* и *Out*. При применении нескольких атрибутов к одной сущности порядок атрибутов не имеет значения. В C# отдельные атрибуты можно заключить в квадратные скобки или разделить несколько атрибутов запятыми, перечислив их в одних квадратных скобках. Необязательными являются суффикс *Attribute* и круглые скобки в конструкторе, если у последнего нет параметров. Приведенные далее строки имеют одинаковый эффект и демонстрируют все возможные способы применения нескольких атрибутов:

```
[Serializable][Flags]
[Serializable, Flags]
[FlagsAttribute, SerializableAttribute]
[FlagsAttribute()][Serializable()]
```

Определение собственного класса атрибутов

Вы уже знаете, что любой атрибут наследует классу *SystemAttribute*, и умеете применять атрибуты. Теперь посмотрим, как определять собственные нестандартные атрибуты. Представьте себе, что вы сотрудник Microsoft и вам поручили реализовать поддержку битовых флагов в перечислимых типах. Для начала нужно определить тип *FlagsAttribute*:

```
namespace System {
   public class FlagsAttribute : System.Attribute {
      public FlagsAttribute() {
      }
   }
}
```

Заметьте: тип *FlagsAttribute* наследует *Attribute*, что делает его CLS-совместимым нестандартным атрибутом. Кроме того, у всех неабстрактных атрибутов должны быть модификатор доступа *public*, а имена атрибутных типов — заканчиваться словом *Attribute*. И, наконец, все неабстрактные атрибуты должны содержать хотя бы один открытый конструктор. Простейший конструктор *FlagsAttribute* не имеет параметров и ничего не делает.

Внимание! Атрибут следует рассматривать как логический контейнер состояния. Иначе говоря, хотя тип-атрибут и является классом, этот класс должен быть простым, то есть содержать один открытый конструктор, принимающий обязательную (или позиционную) информацию о состоянии атрибута. Дополнительно класс может содержать открытые поля и свойства, которые принимают дополнительную (или поименованную) информацию о состоянии атрибута. У класса не должно быть никаких открытых методов, событий и других членов.

В общем случае я обычно рекомендую использовать открытые поля и не меняю эти советы и в отношении атрибутов. Намного лучше использовать свойства, потому что они обеспечивают большую гибкость в случае, если вы решите внести изменения в реализацию класса атрибутов.

Получается, что экземпляры класса *FlagsAttribute* могут применяться к любой сущности, но на самом деле этот атрибут должен применяться только к перечислимым типам. Нет смысла применять такой атрибут к свойству или методу. Чтобы указать компилятору, где допускается применять такой атрибут, применим к нашему типу-атрибуту экземпляр класса *SystemAttributeUsageAttribute*:

```
namespace System {
   [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
   public class FlagsAttribute : System.Attribute {
      public FlagsAttribute() {
      }
   }
}
```

В этой новой версии я применил к атрибуту экземпляр *AttributeUsageAttribute*. В конце концов, тип-атрибут — всего лишь один из классов, а к ним можно применять атрибуты. Атрибут *AttributeUsageAttribute* представляет собой простой класс, который позволяет указать компилятору, где допускается применять ваш нестандартный атрибут. Все компиляторы имеют встроенную поддержку этого атрибута и выдают ошибку, когда определенный пользователем нестандартный атрибут применяется к недопустимой сущности. В данном примере атрибут *AttributeUsage* указывает, что экземпляры атрибута *Flags* могут применяться только к перечислимым типам.

Поскольку атрибуты — такие же типы, как и все остальные, разобраться в *AttributeUsageAttribute* несложно. Вот исходный код этого типа 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, у класса *AttributeUsageAttribute* есть открытый конструктор, позволяющий передавать битовые флаги, которые указывают, где может применяться атрибут. Перечислимый тип *SystemAttributeTargets* 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, которые могут быть установлены при применении этого атрибута к атрибутному типу: *AllowMultiple* и *Inherited*.

Большинство атрибутов нет смысла применять к одной сущности более одного раза. Так, многократное применение атрибутов *Flags* или *Serializable* ничего не даст. На самом деле компилятор выдаст ошибку «error CS0579: Duplicate 'Flags' attribute» (ошибка CS0579: дублирующийся атрибут Flags) при выполнении такого кода:

```
[Flags][Flags]
enum Color {
   Red
}
```

Однако многократное применение некоторых атрибутов к одной сущности имеет смысл. В FCL класс атрибутов *ConditionalAttribute* и многие классы атрибутов, связанные с разрешениями (такие как *EnvironmentPermissionAttribute*, *FileIOPermissionAttribute*, *ReflectionPermissionAttribute*, *RegistryPermissionAttribute* и другие), допускают применение нескольких своих экземпляров к одной сущности. Если вы явно не укажете *AllowMultiple*, ваш атрибут будет вести себя по умолчанию, то есть не позволит применять себя к одной сущности более одного раза.

Другое свойство типа *AttributeUsageAttribute* — *Inherited* — указывает, применяется ли атрибут к производным классам или переопределенным методам при его применении к базовому классу. Суть наследования атрибута демонстрирует такой 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() { }
}
```

Здесь класс *AnotherType* и его метод *DoSomething* имеют атрибут *Tasty*, так как он наследуемый. Однако *AnotherType* — несериализуемый, так как FCL-тип *SerializableAttribute* определен как ненаследуемый атрибут.

В .NET Framework наследование атрибутов допустимо только для классов, методов, свойств, событий, полей, возвращаемых значений методов и параметров. Не забывайте об этом, присваивая *Inherited* значение *true* в собственном типе атрибутов. Кстати, при наличии наследуемых атрибутов дополнительные метаданные в управляемый модуль для производных типов не добавляются. Об этом мы еще поговорим.

Примечание Если при определении собственного класса атрибутов вы забыли применить атрибут *AttributeUsage*, компилятор и CLR будут считать, что ваш атрибут применим к любым сущностям, может применятся к любой из них только один раз и является наследуемым. Такое допущение соответствует значениям по умолчанию полей класса *AttributeUsageAttribute*.

Конструктор атрибута и типы данных полей/свойств

Определяя собственный нестандартный атрибут, вы можете указать конструктор с параметрами, которые должен задавать разработчик, использующий экземпляр атрибута. Кроме того, вы можете определить нестатические открытые поля и свойства своего типа, которые не обязательно устанавливать разработчику, применяющему ваш атрибут.

При определении конструктора, полей и свойств экземпляра класса атрибута следует ограничиться небольшим подмножеством типов данных. Если конкретно, допустимый набор типов данных ограничивается типами *Boolean*, *Char*, *Byte*, *SByte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Int64*, *UInt64*, *Single*, *Double*, *String*, *Type*, *Object* и перечислимыми типами. Кроме того, можно использовать одномерные массивы этих типов с нулевой нижней границей, однако это не рекомендуется, потому что нестандартный класс атрибутов, конструктор которого принимает массив, не совместим с CLS.

Указывая атрибут, нужно передать определенное при компиляции константное выражение, соответствующее типу, определенному в классе атрибутов. Для параметров, полей и свойств, определенных в классе атрибута с типом *Type,* нужно применять оператор *typeof* языка C#, как показано в следующем коде. Для параметров, полей и свойств, определенных в атрибутном классе с типом *Object* можно задавать *Int32*, *String* или другие константные выражения (в том числе null). Если константное выражение имеет размерный тип, он будет упакован в период выполнения при создании экземпляра атрибута.

Вот пример некоторого атрибута и его применения:

```
using System;

internal enum Color { Red }

[AttributeUsage(AttributeTargets.All)]
internal sealed class SomeAttribute : Attribute {
     public SomeAttribute(String name, Object o, Type[] types) {
          // 'name'  указывает на String.
          // 'o' указывает на один из допустимых типов
          // (при необходимости упакованный).
          // 'types' указывает на одномерный массив типа Type с нулевой
          // нижней границей.
     }
}
```

```
[Some("Jeff", Color.Red, new Type[] { typeof(Math), typeof(Console) })]
internal sealed class SomeType {
}
```

Когда компилятор обнаруживает применение нестандартного атрибута, он создает экземпляр класса атрибутов, вызывая его конструктор и передавая ему указанные параметры. Затем компилятор инициализирует указанные открытые поля и свойства. Инициализировав таким образом объект, представляющий собой нестандартный атрибут, компилятор сериализует его и сохраняет в таблице метаданных.

Внимание! Я считаю, что лучше всего представлять себе нестандартный атрибут так: это экземпляр некоторого типа, сериализованный в байтовый поток, находящийся в метаданных. Затем, в период выполнения, экземпляр этого типа создается путем десериализации байт, содержащихся в метаданных. На самом деле происходит следующее: компилятор генерирует информацию, необходимую для создания экземпляра класса атрибутов, и размещает ее в метаданных. За 1-байтным идентификатором параметра конструктора записывается его значение. «Сериализовав» параметры конструктора, компилятор генерирует значения для каждого указанного поля и свойства, записывая его имя, 1-байтный идентификатор типа и собственно значение. Для массивов сначала указывается число элементов, а затем следует перечисление элементов.

Обнаружение использования нестандартных атрибутов

Само по себе определение атрибута бесполезно. Конечно, можно определить какие угодно атрибутные типы и указывать их где попало, но это приведет лишь к дополнительным метаданным в управляемом модуле, а поведение приложения при этом не изменится.

В главе 12 вы видели, что применение атрибута *Flags* к перечислимому типу *System.Enum* изменяет поведение его методов *ToString* и *Format*. Эти методы работают по-разному, так как в период выполнения они проверяют, не связан ли с перечислимым типом, с которым они имеют дело, атрибут *Flags*. Код может анализировать наличие атрибутов, используя *отражение* (reflection). Здесь я лишь вскользь коснусь отражения, а подробно мы обсудим его в главе 22.

Если бы вы отвечали в Microsoft за реализацию метода *Format* типа *Enum*, вы бы могли реализовать его примерно 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; исполняем код, трактующий значение
      // как перечислимый тип с битовыми флагами.
      ...
   } else {
```

```
      // Нет; исполняем код, трактующий значение
      // как обычный перечислимый тип.
      ...
   }
   ...
}
```

Этот код обращается к методу *IsDefined* типа *Type*, чтобы система просмотрела метаданные этого перечислимого типа и определила, связан ли с ним экземпляр типа *FlagsAttribute*. Если *IsDefined* возвращает *true*, с перечислимым типом связывается экземпляр *FlagsAttribute*, и метод *Format* считает, что переданное значение содержит набор битовых флагов. Если *IsDefined* возвращает *false*, *Format* трактует переданное значение как обычный перечислимый тип.

Таким образом, определив собственные атрибутные типы, нужно написать код, проверяющий существование экземпляра класса атрибута (для определенной сущности) и изменяющий порядок выполнения программы. Именно тогда от нестандартного атрибута будет польза!

FCL предлагает множество способов проверки существования атрибута. В случае объекта типа *System.Type* можно вызывать метод *IsDefined*, как показано выше. Однако порой требуется проверка наличия атрибута не для типа, а для сборки, модуля или метода. Рассмотрим методы класса *System.Attribute*. Как вы помните, CLS-совместимые атрибуты порождаются из *System.Attribute*; этот класс определяет три статических метода для получения атрибутов *IsDefined*, *GetCustomAttributes* и *GetCustomAttribute*. Каждая из этих функций имеет перегруженные версии. Так, каждый из этих методов работает с членами типов (классами, структурами, перечислениями, интерфейсами, делегатами, конструкторами, методами, свойствами, полями, событиями и возвращаемыми типами), а также с параметрами, модулями и сборками. Есть также версии, позволяющие просматривать иерархию наследования и включать наследуемые атрибуты. Методы кратко описаны в табл. 17-1.

Табл. 17-1. Методы *System.Attribute*, определяющие наличие в метаданных CLS-совместимых нестандартных атрибутов

Метод	Описание
IsDefined	Возвращает true при наличии хотя бы одного экземпляра, указанного производного от *Attribute* типа. Метод выполняется быстро, так как не создает (десериализует) экземпляры атрибутного класса
GetCustomAttributes	Возвращает массив, каждый элемент которого является экземпляром указанного атрибутного класса, применяемого к данной сущности. Если методу не передать никакой атрибутный класс, массив будет содержать экземпляры всех примененных атрибутов, независимо от их класса. Каждый экземпляр создается (десериализуется), используя параметры, поля и свойства, указанные при компиляции. Если сущность не имеет экземпляров указанного атрибутного типа, возвращается пустой массив. Этот метод обычно используется с атрибутами, у которых *AllowMultiple* равно true или списку примененных атрибутов

Табл. 17-1. *(окончание)*

Метод	Описание
GetCustomAttribute	Возвращает экземпляр указанного атрибутного типа, применяемого к данной сущности. Этот экземпляр создается (десериализуется), используя параметры, поля и свойства, указанные при компиляции. Если сущность не имеет экземпляров указанного атрибутного класса, возвращается null. Если к сущности применяется несколько экземпляров указанного атрибута, генерируется исключение *System.Reflection.AmbiguousMatchException*. Этот метод обычно используется с атрибутами, у которых *AllowMultiple* равно false

Если нужно установить сам факт применения атрибута, используйте *IsDefined*, поскольку он гораздо быстрее двух других методов. Как вы помните, применяя атрибут к сущности, можно задать параметры конструктору атрибута, а также, при необходимости, определить поля и свойства. При использовании *IsDefined* объект-атрибут не создается, его конструктор не вызывается и его поля и свойства не определяются.

Если надо создать объект-атрибут, вызовите *GetCustomAttributes* или *GetCustomAttribute*. При каждом вызове этих методов создаются экземпляры указанных классов атрибутов, и на основе значений, указанных в исходном коде, устанавливаются поля и свойства экземпляров. Эти методы возвращают ссылки на созданные экземпляры атрибутных классов.

Каждый из этих методов, выполняясь, просматривает метаданные управляемого модуля и сравнивает строки для поиска указанного класса нестандартного атрибута. На это, конечно, уходит время. Если вас волнует быстродействие, подумайте о кешировании результатов выполнения этих методов, вместо того чтобы вызывать их раз за разом, запрашивая одну и ту же информацию.

В пространстве имен *System.Reflection* определено несколько классов, позволяющих анализировать содержимое метаданных модуля: *Assembly*, *Module*, *Enum*, *ParameterInfo*, *MemberInfo*, *Type*, *MethodInfo*, *ConstructorInfo*, *FieldInfo*, *EventInfo*, *PropertyInfo* и соответствующие им классы *•Builder*. Все эти классы предлагают методы *IsDefined* и *GetCustomAttributes*. Только *System.Attribute* предлагает очень удобный метод *GetCustomAttribute*.

Версия *GetCustomAttributes*, определенная в классах, связанных с отражением, возвращает массив типа *Object* (*Object[]*), а не *Attribute* (*Attribute[]*). Это объясняется тем, что типы, связанные с отражением, могут возвращать объекты атрибутных классов, не соответствующих спецификации CLS. Не стоит переживать, так как атрибуты, не совместимые с CLS, встречаются крайне редко. Лично я не видел ни одного за все время работы с .NET Framework.

Примечание Имейте в виду, что только классы *Attribute*, *Type* и *MethodInfo* реализуют методы отражения, поддерживающие булев параметр *inherit*. Все остальные методы, просматривающие атрибуты, игнорируют этот параметр и не проверяют иерархию наследования. Если нужно проверять наличие унаследованного атрибута в событиях, свойствах, полях, конструкторах или параметрах, придется вызывать один из методов класса *Attribute*.

И еще один момент: когда какой-то тип передается методам *IsDefined*, *GetCustomAttribute* или *GetCustomAttributes*, они ищут указанный атрибутный тип или любой другой, производный от него. Если вам нужен конкретный класс атрибута, нужно выполнить дополнительную проверку возвращенного значения, чтобы убедиться, что эти методы вернули именно тот класс, который нужен. Можно определить свой атрибутный класс как изолированный, чтобы избежать возможных недоразумений и дополнительных проверок.

Вот пример, в котором рассматриваются все методы некоторого типа и выводятся атрибуты каждого из них. Этот код чисто демонстрационный; в нормальной ситуации не следует применять показанные здесь нестандартные атрибуты по отношению к тем сущностям, к которым я их здесь применяю.

```
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() {
      // Получить и вывести атрибуты, примененные к данному типу.
      ShowAttributes(typeof(Program));

      // Получить набор методов типа.
      MemberInfo[] members = typeof(Program).FindMembers(
         MemberTypes.Constructor | MemberTypes.Method,
         BindingFlags.DeclaredOnly | BindingFlags.Instance |
         BindingFlags.Public | BindingFlags.Static,
         Type.FilterName, "*");

      foreach (MemberInfo member in members) {
         // Отобразить атрибуты, примененные к данному 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) {
         // Отобразить тип каждого примененного атрибута.
         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 увидим:

```
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.STAThreadAttribute
  System.CLSCompliantAttribute
    IsCompliant=True

Attributes applied to .ctor: None
```

Сравнение двух экземпляров атрибута

Теперь, когда код может проверить, применяется ли экземпляр атрибута к некоторой сущности, может понадобиться проанализировать значения полей атрибута. Один из способов — написать код, анализирующий значения полей атрибутного класса. Однако ваш атрибутный класс также может переопределить метод *Match* типа *SystemAttribute*, чтобы учесть тот факт, что при каждом возвращении атрибута механизмом отображения создается новый экземпляр. Поскольку реализация *AttributeMatch* по умолчанию просто вызывает *Equals*, нужно соблюдать осторожность. Затем код может создать экземпляр атрибутного класса и вызвать *Match* для сравнения его с экземпляром, применяемым к сущности. Вот пример:

```
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 Match и базовым классом
      // не является Attribute, раскомментируйте следующую строку.
      // if (!base.Match(obj)) return false;

      // Поскольку 'this' не null, то при obj, равном null,
      // объекты не могут быть равны.
      // ПРИМЕЧАНИЕ: вы можете удалить эту строку, если уверены,
      // что базовый класс реализует Match корректно.
      if (obj == null) return false;

      // Объекты разного типа не могут быть равны.
      // ПРИМЕЧАНИЕ: вы можете удалить эту строку, если уверены,
      // что базовый класс реализует Match корректно.
      if (this.GetType() != obj.GetType()) return false;

      // Приводим obj к нашему типу, чтобы получить доступ к полям.
      // ПРИМЕЧАНИЕ: такое приведение работает всегда,
      // поскольку известно, что объекты одного типа.
      AccountsAttribute other = (AccountsAttribute) obj;
```

```
      // Сравниваем поля, как считаем нужным.
      // Проверяем, является ли accounts 'this'
      // подмножеством accounts объекта other.
      if ((other.m_accounts & m_accounts) != m_accounts)
         return false;

      return true; // Объекты равны.
   }

   public override Boolean Equals(Object obj) {
      // Если в базовом классе реализован Equals и базовым классом
      // не является Object, раскомментируйте следующую строку.
      // if (!base.Equals(obj)) return false;

      // Поскольку 'this' не null, то при obj, равном null,
      // объекты не могут быть равны.
      // ПРИМЕЧАНИЕ: вы можете удалить эту строку, если уверены,
      // что базовый тип реализует Match корректно.
      if (obj == null) return false;

      // Объекты разного типа не могут быть равны.
      // ПРИМЕЧАНИЕ: вы можете удалить эту строку, если уверены,
      // что базовый класс реализует Match корректно.
      if (this.GetType() != obj.GetType()) return false;

      // Приводим obj к нашему типу, чтобы получить доступ к полям.
      // ПРИМЕЧАНИЕ: такое приведение работает всегда,
      // поскольку известно, что объекты одного типа.
      AccountsAttribute other = (AccountsAttribute) obj;

      // Сравниваем значения полей.
      // Проверяем, равен ли accounts 'this'
      // accounts объекта other.
      if (other.m_accounts != m_accounts)
         return false;

      return true; // Объекты равны.
   }

   // Переопределяем GetHashCode, так как Equals 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());

      // Демонстрирует, что метод работает корректно
      // для типа без атрибута AccountsAttribute.
      CanWriteCheck(new Program());
   }

   private static void CanWriteCheck(Object obj) {
      // Создаем и инициализируем экземпляр атрибутного типа.
      Attribute checking = new AccountsAttribute(Accounts.Checking);

      // Создаем экземпляр атрибута, примененного к типу.
      Attribute validAccounts = Attribute.GetCustomAttribute(
         obj.GetType(), typeof(AccountsAttribute), false);

      // Если атрибут применен к данному типу и определяет счет "Checking",
      // указываем, что этот счет работает с чеками.
      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());
      }
   }
}
```

Скомпоновав и запустив это приложение, получим:

```
ChildAccount types can NOT write checks.
AdultAccount types can write checks.
Program types can NOT write checks.
```

Если вы определите нестандартный атрибут, не переопределив метод *Match*, вы унаследуете реализацию метода *Match* типа *Attribute*, а эта реализация просто вызывает *Equals* и не годится для сравнения новых экземпляров атрибутов, даже с одинаковым содержимым.

Обнаружение использования нестандартных атрибутов без создания объектов, производных от *Attribute*

Сейчас мы поговорим об альтернативном методе обнаружения нестандартных атрибутов, примененных к элементам метаданных. В определенных ситуациях, в которых требуется повышенная безопасность, этот метод заботится о том, чтобы не исполнялся никакой код класса, производного от *Attribute*. Вообще говоря, при вызове методов *GetCustomAttribute(s)* типа *Attribute* внутренний код этих методов вызывает конструктор атрибутного класса и также может вызывать методы-аксессоры, задающие значения свойств. Кроме того, при первом обращении к типу CLR

вызывает конструктор типа (если он есть). Конструктор, аксессор *set* и методы конструктора типа могут содержать код, выполняющийся каждый раз при поиске атрибута. Это позволяет выполнить в домене приложения неизвестный код, что создает определенную угрозу безопасности.

Для обнаружения атрибутов без выполнения кода атрибутного класса используется класс *System.Reflection.CustomAttributeData*. В нем определен один статический метод *GetCustomAttributes*, который служит для получения информации о примененных атрибутах. У этого метода четыре перегруженные версии: первая принимает *Assembly*, вторая — *Module*, третья *ParameterInfo* и четвертая — *MemberInfo*. Этот класс определен в пространстве имен *System.Reflection* (см. главу 22). Обычно для анализа атрибутов в метаданных сборки, загружаемой статическим методом *ReflectionOnlyLoad* (также обсуждается в главе 22) типа *Assembly*, используется класс *CustomAttributeData*. Короче говоря, при загрузке сборки метод *ReflectionOnlyLoad* не позволяет CLR выполнять какой-либо код из нее, в том числе конструкторы типов.

Метод *GetCustomAttributes* типа *CustomAttributeData* действует как фабрика, то есть возвращает набор объектов *CustomAttributeData* в *IList<CustomAttributeData>*. Каждому элементу набора соответствует нестандартный атрибут, примененный к какой-либо сущности. Запрашивая отдельные объекты *CustomAttributeData*, можно получать некоторые неизменяемые свойства и узнавать, как *мог бы* быть сконструирован и инициализирован объект-атрибут. В частности, свойство *Constructor* указывает, какой конструктор *мог бы* быть вызван, свойство *ConstructorArguments* возвращает аргументы, которые *могли бы* быть переданы конструктору в качестве экземпляра *IList<CustomAttributeTypedArgument>*, а свойство *NamedArguments* возвращает поля и свойства, которые *могли бы* быть созданы как экземпляр *IList<CustomAttributeNamedArgument>*. Заметьте: я говорю «могли бы», потому что в действительности конструктор и методы-аксессоры не вызываются — мы позаботились о безопасности и запретили выполнение любых методов атрибутных классов.

Вот обновленная версия предыдущего примера, в которой для получения применяемых 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() {
      // Отображаем атрибуты, примененные к данному типу.
      ShowAttributes(typeof(Program));

      // Получаем набор методов данного типа.
      MemberInfo[] members = typeof(Program).FindMembers(
         MemberTypes.Constructor | MemberTypes.Method,
         BindingFlags.DeclaredOnly | BindingFlags.Instance |
         BindingFlags.Public | BindingFlags.Static,
         Type.FilterName, "*");

      foreach (MemberInfo member in members) {
         // Отображаем атрибуты, примененные 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) {
         // Отображаем тип 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();
   }
}
```

Скомпоновав и запустив это приложение, получим:

```
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.STAThreadAttribute
      Constructor called=Void .ctor()
      Positional arguments passed to constructor: None
      Named arguments set after construction: None

Attributes applied to .ctor: None
```

Условные атрибутные классы

Программисты все чаще используют атрибуты из-за простоты их определения, применения и отображения. Атрибуты — это еще и очень простой способ комментирования кода и одновременно реализации богатых возможностей. До недавнего времени программисты использовали атрибуты для облегчения написания и отладки кода. В частности, пользователи NUnit применяют такие атрибуты, как *TestFixtureSetUp*, *Setup*, *TearDown*, *TestFixtureTearDown* и *Test*, к типам и методам, с которыми должна работать утилита NUnit в процессе блочного тестирования.

Эти атрибуты нужны только утилите NUnit — при обычной работе программы они не используются. В отсутствие NUnit нужные ей атрибуты занимают лишнее место в метаданных, увеличивая объем файла и рабочий набор процесса и отрицательно сказываясь на производительности. Было бы неплохо, если бы компилятор генерировал атрибуты для NUnit только при выполнении блочного тестирования. К счастью, такая возможность существует, и реализуется она с использованием условных атрибутных классов.

Атрибутный класс, к которому применен атрибут *System.Diagnostics.ConditionalAttribute*, называют *условным атрибутным классом*. Вот пример:

```
// #define TEST
#define VERIFY

using System;
using System.Diagnostics;

[Conditional("TEST")][Conditional("VERIFY")]
public sealed class CondAttribute : Attribute {
}

[Cond]
public sealed class Program {
   public static void Main() {
      Console.WriteLine("CondAttribute is {0}applied to Program type.",
         Attribute.IsDefined(typeof(Program),
            typeof(CondAttribute)) ? "" : "not ");
   }
}
```

Обнаружив применение *CondAttribute*, компилятор разместит в метаданных информацию об атрибуте, только если при компиляции кода определен идентификатор TEST или VERIFY. Тем не менее метаданные определения и реализации атрибутного класса все равно останутся в сборке.

ГЛАВА 18

Значимые типы, допускающие присвоение null

Как вы знаете, значение переменной значимого типа никогда не может быть равно null — только значению соответствующего типа. Собственно поэтому значимые типы так и называют. К сожалению, есть ситуации, в которых это вызывает затруднения. Например, при проектировании базы данных можно определить тип данных столбца как «32-разрядное целое», которое соответствует типу *Int32* в FCL. Но в столбце базы данных значение может отсутствовать, что в терминах баз данных обозначается значением null. И это обычная практика. В таких условиях взаимодействие с базами данных средствами Microsoft .NET Framework может оказаться довольно сложным, так как общеязыковая среда CLR не позволяет представить значение типа *Int32* как null.

Примечание Адаптеры таблиц Microsoft ADO.NET поддерживают типы, допускающие присвоение null. Но, к сожалению, типы в пространстве имен *System.Data.SqlTypes* не заменяются типами, допускающими присвоение null, отчасти из-за отсутствия взаимно-однозначного соответствия между этими типами. Например, тип *SqlDecimal* содержит максимум 38 разрядов, тогда как обычный тип *Decimal* может иметь только 29. Кроме того, тип *SqlString* поддерживает собственные региональные стандарты и порядок сравнения, которые не поддерживаются обычным типом *String*.

Примечание В технологиях Microsoft Language Integrated Query (LINQ) типы, допускающие присвоение null, используются для естественной интеграции запросов данных в язык программирования.

Вот еще пример: в Java класс *java.util.Date* является ссылочным типом, поэтому переменной этого типа можно присвоить null. Однако в CLR тип *System.DateTime* — значимый и не может быть равным null. Если Java-приложению потребуется передать информацию о дате и времени Web-службе, работающей на платформе CLR, возможны проблемы, если Java-приложение отправит значение null, потому что CLR не «понимает» это значение и не знает, что с ним делать.

Для исправления ситуации специалисты Microsoft разработали для CLR *значимые типы, допускающие присвоение null* (nullable value type). Чтобы понять, как они работают, познакомимся с определенным в FCL классом *System.Nullable<T>*. Ниже приведено схематическое представление реализации этого типа.

```
[Serializable, StructLayout(LayoutKind.Sequential)]
public struct Nullable<T> where T : struct {

   // 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;
   }
}
```

Этот класс реализует значимый тип, который может принимать значение null. *Nullable<T>* — также значимый тип, поэтому его экземпляры достаточно производительны. Причина в том, что хотя экземпляры нового типа и размещаются в стеке, их размер практически равен размеру исходного типа за исключением небольшого добавления — одного булева поля. Заметьте: параметр *T* типа *Nullable* ограничен struct, потому что переменные ссылочного типа и так могут принимать значение null.

Теперь при желании использовать в своем коде тип *Int32*, допускающий присвоение null, можно написать так:

```
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 компоновки и выполнения этого кода получим:

```
x: HasValue=True, Value=5
y: HasValue=False, Value=0
```

Поддержка значимых типов, допускающих присвоение null, в C#

В приведенном выше примере используется довольно простой синтаксис C# для инициализации x и y — двух переменных типа *Nullable<Int32>*. Команда разработчиков C# стремилась интегрировать значимые типы, допускающие присвоение null, в язык C# и сделать их полноправными членами существующего семейства типов. На сегодняшний день, C# поддерживает довольно удобный синтаксис управления значимыми типами, допускающими присвоение null. C# позволяет в коде объявлять и инициализировать переменные x и y, используя знак вопроса:

```
Int32? x = 5;
Int32? y = null;
```

В C# нотация *Int32?* является синонимом *Nullable<Int32>*. Но в C# такой подход означает намного большее. C# позволяет выполнять преобразования и приводить экземпляры, допускающие присвоение null, к другим типам. Также C# поддерживает применение операторов к экземплярам значимых типов, допускающих присвоение null. Следующий код содержит несколько примеров:

```
private static void ConversionsAndCasting() {
   // Неявное преобразование из обычного типа Int32 в Nullable<Int32>.
   Int32? a = 5;

   // Явное преобразование из 'null' в Nullable<Int32>.
   Int32? b = null;

   // Явное преобразование из Nullable<Int32> в обычный тип Int32.
   Int32 c = (Int32) a;

   // Прямое и обратное приведение элементарного типа в тип,
   // допускающий присвоение null.
   Double? d = 5; // Int32->Double?  (d is 5.0 as a double)
   Double? e = b; // Int32?->Double? (e is null)
}
```

C# также позволяет применять операторы к экземплярам типов, допускающих присвоение null. 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 C# интерпретирует операторы:

- **Унарные операторы** (+, ++, —!, ~). Если операнд равен null, результат тоже null.
- **Бинарные операторы** (+, –, *, /, %, &, |, ^, <<, >>). Если хотя бы один операнд равен null, результат тоже null.
- **Операторы равенства** (==, !=). Если оба операнда равны null, операнды считаются равными. Если один операнд равен null, операнды не равны. Если ни один операнд не равен null, сравниваются значения операндов на предмет равенства.

- **Операторы сравнения** (<>, <=, >=). Если любой из операндов равен null, результат — false. Если ни один операнд не равен null, сравниваются значения операндов.

Имейте в виду, что операции с экземплярами типов, допускающих присвоение null, вызывают генерацию большого объема кода. Вот пример метода:

```
private static Int32? NullableCodeSize(Int32? a, Int32? b) {
   return a + b;
}
```

При компиляции создается достаточно много IL-кода. Вот эквивалент на языке C# сгенерированного компилятором IL-кода:

```
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());
}
```

Оператор интеграции null в языке C#

В C# есть оператор, который называют *оператором интеграции null* (null-coalescing operator). Он обозначается знаком *??* и принимает два операнда. Если левый операнд не равен null, оператор возвращает значение этого операнда. Если левый операнд равен null, возвращается значение правого операнда. Оператор интеграции null очень удобен для задания значения переменной по умолчанию.

Основное преимущество этого оператора — поддержка как ссылочных типов, так и значимых типов, допускающих присвоение null. Вот код, демонстрирующий использование оператора интеграции null:

```
private static void NullCoalescingOperator() {
   Int32? b = null;

   // Приведенная ниже строка эквивалентна следующей:
   // x = (b.HasValue) ? b.Value : 123
   Int32 x = b ?? 123;
   Console.WriteLine(x);  // "123"

   // Приведенная ниже строка эквивалентна следующему коду:
   // String temp = GetFilename();
   // filename = (temp != null) ? temp : "Untitled";
   String filename = GetFilename() ?? "Untitled";
}
```

CLR обеспечивает специальную поддержку значимых типов, допускающих присвоение null

CLR предлагает встроенную поддержку значимых типов, допускающих присвоение null. Она предусматривает упаковку и распаковку, вызов *GetType* и методов интерфейса — все это призвано обеспечить более тесную интеграцию значимых типов, допускающих присвоение null, в CLR. Такие типы ведут себя более естественно, как ожидает большинство разработчиков. Давайте поближе познакомимся с поддержкой этих типов в CLR.

Упаковка значимых типов, допускающих присвоение null

Представьте себе переменную типа *Nullable<Int32>*, которая логически инициализируется значением null. Далее эту переменную передают к методу, ожидающему *Object*; переменную нужно упаковать и передать в метод ссылку на упакованный тип *Nullable<Int32>*. Это не очень хорошо, потому что в метод передается не равное null значение, несмотря на то, что переменная типа *Nullable<Int32>* логически содержит null. Для устранения этой шероховатости CLR выполняет некоторый специальный код при упаковке переменной, допускающей присвоение null, чтобы создать иллюзию того, что значимые типы, допускающие присвоение null, являются полноценными типами.

В частности, при упаковке экземпляра *Nullable<T>* среда CLR выясняет, не равен ли он null; если да, CLR не выполняет упаковку и возвращается null. Если экземпляр не равен null, CLR упаковывает его значение. Иначе говоря, *Nullable<Int32>* со значением 5 упаковывается, как обычный тип *Int32* со значением 5. Вот код, демонстрирующий это поведение:

```
// После упаковки Nullable<T> возвращается null или упакованный T.
Int32? n = null;
Object o = n;  // o is 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 значимых типов, допускающих присвоение null

CLR позволяет распаковать значимый тип *T* в тот же тип *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:

```
// Создаем упакованный тип Int32.
Object o = 5;

// Распаковываем его в Nullable<Int32> и в Int32.
Int32? a = (Int32?) o;  // a = 5
Int32  b = (Int32)  o;  // b = 5
```

```
// Создаем ссылку, инициализированную значением null.
o = null;

// "Распаковываем" ее в Nullable<Int32> и в Int32.
a = (Int32?) o;      // a = null.
b = (Int32)  o;      // Исключение NullReferenceException.
```

При распаковке значимого типа в значимый тип, допускающий присвоение null, среде CLR, скорее всего, придется выделить память. Это довольно необычно, потому что во всех других ситуациях распаковка не требует выделения памяти. Вот пример:

```
private static void UnboxingAllocations() {
   const Int32 count = 1000000;

   // Создаем упакованный тип Int32.
   Object o = 5;

   Int32 numGCs = GC.CollectionCount(0);
   for (Int32 x = 0; x < count; x++) {
      Int32 unboxed = (Int32) o;
   }
   Console.WriteLine("Number of GCs={0}", GC.CollectionCount(0) - numGCs);

   numGCs = GC.CollectionCount(0);
   for (Int32 x = 0; x < count; x++) {
      Int32? unboxed = (Int32?) o;
   }
   Console.WriteLine("Number of GCs={0}", GC.CollectionCount(0) - numGCs);
}
```

После компиляции и выполнения этого метода получим:

```
Number of GCs=0
Number of GCs=30
```

Единственное различие между двумя циклами в приведенном выше коде в том, что первый распаковывает *o* в *Int32*, а второй — в *Nullable<Int32>*. Но во втором цикле выполняется 30 операций сборки мусора, что ясно свидетельствует о создании объектов в стеке в процессе распаковки. В главе 5 говорилось, что распаковка — это просто операция получения ссылки на распакованную часть упакованного объекта. Проблема в том, что упакованный значимый тип нельзя просто распаковать в nullable-версию значимого типа, потому что у упакованного значимого типа отсутствует булево поле *hasValue*. Поэтому при распаковке значимого типа в nullable-версию, CLR должна выделить память для объекта *Nullable<T>*, инициализировать поле *hasValue* значением true и присвоить полю значение, содержащееся в упакованном значимом типе. Эти операции снижают производительность приложения.

Вызов *GetType* через значимый тип, допускающий присвоение null

При вызове *GetType* по отношению к объекту *Nullable<T>* CLR фактически «лжет» и возвращает тип *T* вместо *Nullable<T>*. Вот пример:

```
Int32? x = 5;

// Приведенный ниже оператор отображает "System.Int32", а не "System.Nullable<Int32>".
Console.WriteLine(x.GetType());
```

Вызов интерфейсных методов через значимый тип, допускающий присвоение null

В приведенном ниже коде переменная *n* типа *Nullable<Int32>* приводится к интерфейсному типу *IComparable<Int32>*. Однако в отличие от *Int32* тип *Nullable<T>* не реализует интерфейс *IComparable<Int32>*. Тем не менее компилятор C# успешно компилирует код, а верификатор CLR считает этот код прошедшим проверку. Все это нужно для обеспечения программистам более удобного синтаксиса.

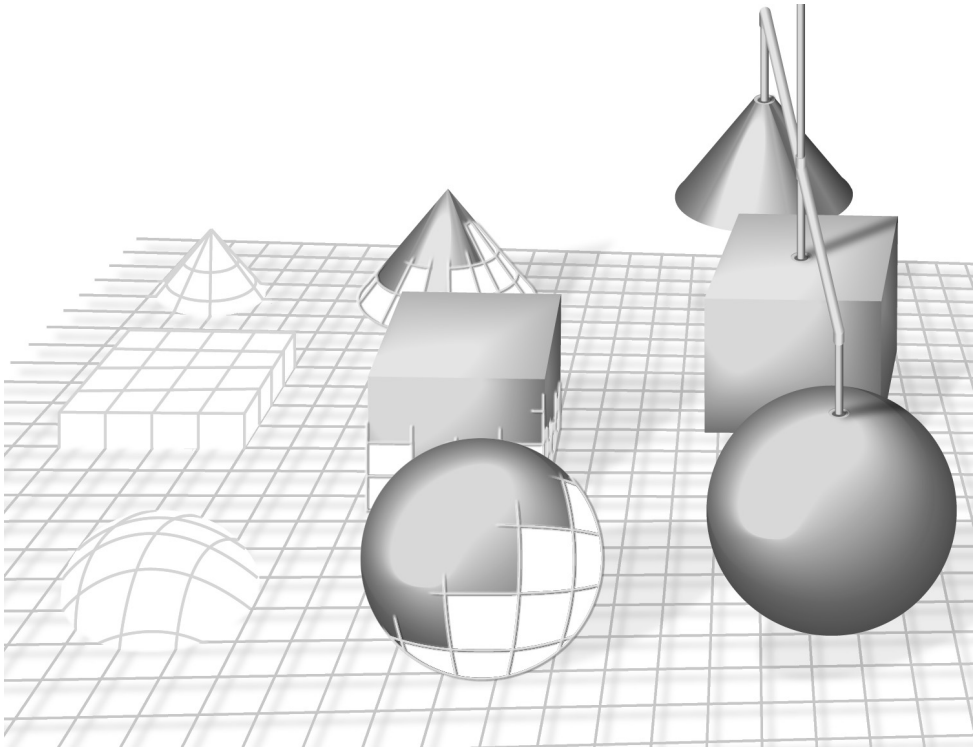
```
Int32? n = 5;
Int32 result = ((IComparable) n).CompareTo(5);    // Без проблем компилируется
                                                  // и выполняется.
Console.WriteLine(result);                        // 0
```

Если бы CLR не обеспечивала такую особую поддержку, пришлось бы писать громоздкий код вызова метода интерфейса через значимый тип, допускающий присвоение null. Чтобы выполнить вызов, вы должны были бы приводить распакованный значимый тип перед его приведением к интерфейсу:

```
Int32 result = ((IComparable) (Int32) n).CompareTo(5);  // Громоздко.
```

ЧАСТЬ V

СРЕДСТВА CLR



ГЛАВА 19

Исключения

В этой главе я расскажу о мощном механизме, позволяющем писать устойчивый и простой в сопровождении код, — об *обработке исключений* (exception handling). Вот лишь некоторые из его достоинств.

- **Компактное размещение кода, выполняющего очистку, и его гарантированное исполнение** Вынос кода, выполняющего очистку ресурсов, за пределы основной логики приложения в специально отведенное место облегчает написание, понимание и сопровождение приложения. Гарантированное исполнение этого кода означает, что приложение скорее всего останется в согласованном состоянии. Например, если код, записывающий данные в файл, не сможет завершить работу из-за исключения, файл непременно будет закрыт.
- **Централизованное хранение кода, имеющего дело с исключительными ситуациями** Исполнение строки кода может окончиться неудачей по разным причинам: переполнение при арифметической операции или переполнение стека, нехватка памяти, выход за допустимые границы значения аргумента или индекса массива, попытка обращения к ресурсу (например, к файлу) после его закрытия. Без обработки исключений было бы очень сложно, если вообще возможно, написать код, корректно определяющий подобные сбои и устраняющий их последствия. Распределение кода для обнаружения потенциальных сбоев по главной логике приложения затрудняет написание, понимание и сопровождение кода приложения. К тому же исполнение такого кода сильно снижает быстродействие.

  Обработка исключений позволяет избежать написания кода для обнаружения указанных потенциальных сбоев. Вместо этого можно спокойно писать свой код, не беспокоясь о сбоях. Этот подход не только облегчает написание, понимание и сопровождение программ, но и позволяет им работать быстрее. Кроме того, это дает возможность собрать в одном месте весь код для восстановления после сбоев. Обработка исключений «вступает в дело», только чтобы исполнить код для восстановления после сбоя.
- **Облегчение поиска и исправления ошибок в коде** При сбое CLR просматривает стек потока в поисках кода, способного обработать исключение. Если такового нет, сбой станет «необработанным исключением», о котором вы получите уведомление. После этого можно без труда локализовать сбой в исходном тексте, определить его причину и внести исправления, чтобы устранить сбой. Это значит, что ошибки можно будет обнаруживать при разработ-

ке и тестировании и исправлять их до развертывания приложения. Приложения станут устойчивее после развертывания, и, следовательно, впечатление пользователя от работы с ними улучшится.

Обработка исключений — замечательный инструмент, облегчающий исполнение рутинных задач разработчика. Но при неверном использовании она способна разочаровать и измучить программиста, маскируя серьезные ошибки в программе или сигнализируя о ложных неполадках. В этой главе я расскажу, как правильно использовать обработку исключений.

Эволюция обработки исключений

При разработке API Win32 и COM в Microsoft решили отказаться от использования исключений для уведомления вызывающего кода о сбоях. Вместо этого большинство функций Win32 возвращает *false*, когда что-то работает не так. Получив *false*, вызывающий код пытается найти нарушение с помощью функции *GetLastError*. С другой стороны, методы COM возвращают значение *HRESULT*. Старший бит этого значения, равный 1, указывает на сбой, остальные биты представляют значение, позволяющее определить причину нарушения.

Microsoft отказалась от использования исключений в API Win32 и COM по ряду причин.

- Большинство разработчиков не знакомо с обработкой исключений.
- Большинство языков программирования, включая C и ранние версии C++, не поддерживает исключения.
- Некоторые разработчики считают, что разбираться в исключениях слишком сложно, не говоря уже об их применении, поэтому в Microsoft решили не заставлять всех программистов использовать исключения (что до меня, то я считаю, что выгода от этого с лихвой перевешивает неудобства от необходимости их изучения).
- Обработка исключений снижает быстродействие приложения. Иногда обработка исключений работает медленнее, чем простой возврат кода ошибки. Но, если исключения возникают редко, издержки незначительны. Считаю, что и в этом случае выгода от использования исключений перевешивает неудобство, доставляемое незначительным периодическим снижением быстродействия. Более того, я уверен, что корректное использование обработки исключений во всем коде приложения даже повышает его быстродействие. Ниже мы еще поговорим о быстродействии. Кроме того, «стоимость» обработки исключений в управляемом коде намного ниже, чем в других системах, таких как C++.

Старые способы уведомления о нарушениях накладывали слишком много ограничений, так как вызывающий код получал лишь 32-разрядное число. Если это код недопустимого аргумента, то не ясно, какой из аргументов оказался недопустимым. Если же это код деления на 0, то нельзя было точно назвать строку, в которой оно имело место, поэтому исправить код было непросто. Разрабатывать программы и без того нелегко, поэтому потеря важной информации — просто непозволительная роскошь! Если код приложения обнаруживает ошибку, нужны самые полные сведения о ней, чтобы как можно скорее принять меры для ее устранения.

Механизм обработки исключений предлагает кое-что получше 32-разрядных кодов. В исключение входит строка описания, позволяющая точно определить

аргумент, вызвавший ошибку. В этой строке также могут быть дополнительные сведения, которые помогут улучшить код. Наконец, в исключении есть трассировка стека, помогающая выяснить, по какому пути пошло исполнение приложения, что привело к возникновению исключения.

Другое преимущество исключений в том, что их не обязательно перехватывать или обнаруживать там же, где они возникают — их обработку можно поручить коду, размещаемому в стеке вызовов потока. Это заметно упрощает кодирование, так как можно не снабжать каждый оператор или метод, который может дать сбой, кодом для обнаружения и устранения ошибок.

С предыдущим связано, наверное, главное преимущество исключений: их не так-то просто игнорировать. Если вызванная Win32-функция возвращает код ошибки, вызывающий ее код может запросто проигнорировать его, предполагая, что функция сработала, как ожидалось, и позволить программе работать дальше, как ни в чем ни бывало. Но если метод генерирует исключение, значит, он не может работать, как ожидается. Если приложение не перехватывает исключение, CLR прерывает его исполнение. Некоторым такое поведение может показаться излишне радикальным и суровым, но я думаю, что так и надо. Если при вызове метода возникло исключение, нельзя продолжать исполнение приложения, поскольку остальная часть программы предполагает, что все предыдущие операции завершены нормально. В противном случае приложение продолжит работу, но ее результаты будут непредсказуемыми. Так, при повреждении пользовательских данных в приложении нельзя продолжать работать с ними. При использовании 32-разрядных кодов в Win32 и COM вероятность того, что приложение продолжит работу и будут получены непредсказуемые результаты, все же слишком высока, а если применяются исключения, это невозможно.

Все методы типов из Microsoft .NET Framework генерируют исключения, а не возвращают 32-разрядные коды состояния, уведомляющие о нарушении. Поэтому каждый программист должен знать все об исключениях и их обработке в своем коде. Здесь Microsoft было принято замечательное решение! В использовании обработки исключений нет ничего хитрого, она позволяет писать понятный код, который легко внедрять и сопровождать. Код, применяющий обработку исключений, устойчив и способен восстановиться практически всегда, что бы ни случилось с приложением. Правильное использование обработки исключений может предотвратить крах, а пользователи всегда будут довольны вашими программами.

Механика обработки исключений

В этом разделе я познакомлю вас с механизмами и конструкциями языка C# для обработки исключений, но досконального разъяснение этих материй не входит в мои намерения. Цель этой главы — рассказать, когда и как использовать обработку исключений. Подробная информация о механизмах и конструкциях языка, отвечающих за обработку исключений, есть в документации по .NET Framework и спецификации языка C#. Замечу также, что механизм обработки исключений .NET Framework построен с использованием механизма *структурной обработки исключений* (structured exception handling, SEH) Windows. SEH обсуждается во многих источниках, в том числе в моей книге Programming Applications for Microsoft Windows, 4 Edition, Microsoft Press, 1999 (Windows для профессионалов: создание эффектив-

ных Win32-приложений с учетом специфики 64-разрядной версии Windows, 4-е изд., СПб.: Питер, М.: Издательско-торговый дом «Русская редакция», 2001 г.).

Следующий код на C# демонстрирует стандартное применение механизма обработки исключений. Он дает представление о том, как выглядят и для чего нужны блоки обработки исключений. В комментариях дано формальное описание блоков *try*, *catch* и *finally* и указано их назначение.

```
private void SomeMethod() {

   try {
      // Внутрь блока try помещают код, требующий корректного
      // восстановления работоспособности или очистки ресурсов.
   }
   catch (InvalidOperationException) {
      // В такие блоки catch помещают код, который должен восстанавливаться
      // после исключений типа InvalidOperationException (или любого исключения,
      // производного от него).
   }
   catch (IOException) {
      // В такие блоки catch помещают код, который должен восстанавливаться
      // после исключений типа IOException (или любого исключения,
      // производного от него).
   }
   catch {
      // В такие блоки catch помещают код, который должен
      // восстанавливаться после исключений любого типа.

      // После перехвата исключений их, как правило,
      // генерируют повторно. Ниже я еще вернусь к этому.
      throw;
   }
   finally {
      // Внутрь блоков finally помещают код, выполняющий очистку ресурсов
      // после любых действий, начатых в блоке try. Код из этого блока
      // исполняется ВСЕГДА независимо от того, было исключение или нет.
   }
   // Код после блока finally исполняется, если в блоке try не было
   // исключения или если исключение было перехвачено блоком catch
   // и не было сгенерировано то же самое или другое исключение.
}
```

Этот код демонстрирует один из возможных способов использования блоков для обработки исключений. Пусть его вид не пугает вас — обычно метод окружают лишь два блока, скажем, *try* и соответствующий ему одиночный блок *finally* или блоки *try* и *catch*. В реальных программах редко встречается столько блоков *catch*, сколько я показал. Я поместил их здесь в таком количестве лишь для наглядности.

Блок *try*

Блок *try* содержит код, требующий общей очистки ресурсов или восстановления после исключения. Код очистки должен размещаться в единственном блоке *finally*. В блоке *try* также может быть код, способный с определенной вероятностью сгенерировать исключение. Код восстановления после исключения надо разместить в одном или нескольких блоках *catch*. Следует создать по одному блоку *catch* для каждого вида событий, после которых, по вашим прогнозам, придется восстанавливать приложение. Блок *try* должен быть связан хотя бы с одним блоком *catch* или *finally*, сами по себе блоки *try* бессмысленны.

Блок *catch*

Блок *catch* содержит код, который должен выполняться при возникновении исключения. У блока *try* может быть несколько или ни одного связанного с ним блока *catch*. Если код из блока *try* не вызывает исключение, CLR никогда не исполнит код из соответствующих блоков *catch*. Поток пропускает все эти блоки и начинает исполнять код блока *finally* (если таковой существует). Исполнив код блока *finally*, поток переходит к оператору, следующему за этим блоком.

Выражение, расположенное в скобках после ключевого слова *catch*, называют *типом исключения* (catch type). В C#-программах тип исключения должен содержать тип *SystemException* или его потомок. Так, показанный выше код содержит блоки *catch*, обрабатывающие исключения типа *InvalidOperationException* (и его производные), *IOException* (и его производные). Последний блок *catch* (в нем тип исключения не указан) обрабатывает любые другие исключения; такой блок равносилен блоку *catch* с типом исключения *System.Exception*, если только нельзя получить доступ об исключении из кода, расположенного в блоке *catch*.

Примечание Чтобы увидеть последний сгенерированный объект-исключение при отладке блока *catch* средствами Microsoft Visual Studio, надо добавить в окно контроля переменных специальную переменную с именем «$exception».

CLR просматривает блоки *catch* сверху вниз, поэтому «узкоспециализированные» исключения надо размещать выше. Вначале должны следовать наиболее удаленные потомки, потом — их базовые классы и лишь в конце — *System.Exception* (или блоки без указания типа исключения). В противном случае компилятор C# генерирует ошибку, так как более специализированный блок *catch*, размещенный после менее специализированного, будет недостижим.

Если при исполнении кода из блока *try* (или любого метода, вызванного в блоке *try*) возникает исключение, CLR начинает поиск блоков *catch* с типами, соответствующими этому исключению. Если ни один из этих фильтров не принимает исключение, CLR продолжает просматривать стек вызовов в поисках фильтра перехвата, который принял бы это исключение. Если по достижении вершины стека вызовов блок *catch*, способный обработать это исключение, не найден, оно считается необработанным. О необработанных исключениях я расскажу чуть позже.

Обнаружив фильтр блока, способного обработать исключение, CLR исполняет все внутренние блоки *finally*. Первым исполняется тот, что связан с блоком *try*,

в котором возникло исключение, а последним — блок *catch*, тип которого соответствует исключению. Заметьте: ни один блок *finally*, связанный с блоками *catch*, не исполняется, пока не завершено исполнение кода из блока *catch*, обрабатывающего исключение.

В конце концов, по исполнении кода блоков *finally* исполняется код из обрабатывающего блока *catch*. Как правило, этот код выполняет некоторые действия для восстановления после исключения. Затем можно выбрать одно из трех:

- сгенерировать то же исключение повторно, чтобы уведомить о нем код, расположенный выше по стеку вызовов;
- сгенерировать исключение другого типа, чтобы передать коду, расположенному выше по стеку вызовов, больше сведений об исключении;
- позволить потоку покинуть блок catch.

Ниже я расскажу о том, когда использовать эти методы.

Если выбран первый или второй вариант, генерируется исключение и CLR действует как обычно: просматривает стек вызовов в поисках блока *catch*, тип которого соответствует сгенерированному исключению.

Если выбран последний вариант, поток выходит из блока *catch* и тут же переходит к исполнению кода из блока *finally*, если таковой есть. После исполнения блока *finally* поток покидает этот блок и переходит к выполнению оператора, расположенного сразу после блока *finally*. Если блока *finally* нет, поток переходит к исполнению оператора, стоящего за последним блоком *catch*.

В C# после типа перехвата можно указать имя переменной. При перехвате исключения эта переменная ссылается на сгенерированный объект, потомок *System.Exception*. В коде блока *catch* ее можно использовать для получения информации об исключении (в частности, трассировочного следа в стеке вплоть до исключения). Подробнее о типе *Exception* и о том, что с ним можно делать, я расскажу чуть позже.

Блок *finally*

Блок *finally* содержит код, исполнение которого гарантируется. Обычно этот код выполняет операции очистки, необходимые в результате выполненных в блоке *try* действий. Так, если в блоке *try* был открыт файл, то в блоке *finally* должен быть код, закрывающий этот файл:

```
private void ReadData(String pathname) {

   FileStream fs = null;
   try {
      fs = new FileStream(pathname, FileMode.Open);
      // Обрабатываем данные файла.
      ...
   }
   catch (IOException) {
      // В этот блок catch помещается код для восстановления после исключения
      // типа IOException (или любого производного от него).
      ...
   }
```

```
   finally {
      // Обязательно закройте файл.
      if (fs != null) fs.Close();
   }
}
```

Если код из блока *try* не вызовет исключение, то файл гарантированно будет закрыт; и, даже если возникнет исключение, код из блока *finally* будет выполнен, а файл — закрыт независимо от того, перехвачено ли исключение. Не следует помещать оператор, закрывающий файл, после блока *finally* — он не будет исполнен, если сгенерированное исключение не будет перехвачено, и в итоге файл останется открытым.

Вовсе не обязательно, чтобы с блоком *try* был связан блок *finally*. Иногда код в блоке *try* просто не требует никакого кода для очистки. Но если блок *finally* все же имеется, он должен размещаться после всех блоков *catch*, причем блоку *try* может соответствовать не более одного блока *finally*.

Дойдя до конца кода в блоке *finally*, поток просто начинает исполнять операторы, расположенные за блоком *finally*. Помните: код в блоке *finally* предназначен для очистки и должен делать только то, что необходимо для отмены операций, начатых в блоке *try*. Избегайте в блоке *finally* кода, способного сгенерировать исключение. Но даже если возникло исключение в блоке *finally*, это еще не конец света — механизм обработки исключений среды CLR продолжит исполнение, как будто бы исключение было сгенерировано после блока *finally*. Но при этом CLR напрочь «забудет» об исходном исключении, возникшем в блоке *try* (если таковое было), и вся информация (в частности, трассировочный след в стеке вплоть до исключения) о таком исключении будет потеряна.

Общеязыковая спецификация (CLS) и исключения, отличные от CLS-совместимых

Все CLR-совместимые языки программирования должны поддерживать генерацию объектов, производных от *Exception*, — этого требует общеязыковая спецификация CLS. Однако в действительности CLR разрешает генерировать экземпляры любого типа, и некоторые языки программирования позволят генерировать из кода несовместимые с CLS объекты-исключения, например *String*, *Int32*, *DateTime* и т. д. Компилятор C# разрешает генерировать только производные от Exception объекты, а IL-код и код C++/CLI позволяют генерировать как объекты, производные от Exception, так и любые другие.

Многие программисты не в курсе, что CLR позволяет генерировать любой объект, чтобы сообщить об исключении, и полагают, что допустимы только объекты, производные от *Exception*. До версии 2.0 среды CLR при создании блоков *catch* для перехвата исключений использовались только CLS-совместимые исключения. Если метод на C# вызывал метод, написанный на другом языке, и тот генерировал не совместимое с CLS исключение, код на C# вообще был не состоянии перехватить такое исключение, при этом возникала опасность нарушения защиты.

В версию 2.0 CLR компания Microsoft ввела новый класс — *RuntimeWrappedException* (он определен в пространстве имен *System.Runtime.CompilerServices*). Он

производный от *Exception*, поэтому представляет собой CLS-совместимый тип исключения. В *RuntimeWrappedException* есть закрытое поле типа *Object* (к которому можно обратиться, используя неизменяемое свойство *WrappedException* этого же класса). В CLR версии 2.0 при генерации исключения, не совместимого с CLS, CLR автоматически создает экземпляр класса *RuntimeWrappedException* и инициализирует его закрытое поле ссылкой на объект, который собственно был сгенерирован. На деле получается, что CLR теперь превращает все не совместимые с CLS исключения в CLS-совместимые. Любой код, который теперь перехватывает исключения типа *Exception*, будет перехватывать не совместимые с CLS исключения, что устраняет возможную угрозу безопасности.

Хотя компилятор C# позволяет разработчикам генерировать только объекты, производные от Exception, в действительности, до C# версии 2.0 он разрешал разработчикам перехватывать не совместимые с CLS исключения, используя примерно такой код:

```
private void SomeMethod() {
   try {
      // Внутрь 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 также позволяет перехватывать исключения,
      // не совместимые с CLS.
      throw; // Повторная генерация перехваченного (исключения).
   }
   catch {
      // Во всех версиях C# этот блок перехватывает
      // и совместимые, и не совместимые с CLS исключения.
      throw; // Повторная генерация перехваченного (исключения).
   }
}
```

Некоторые разработчики разузнали, что CLR поддерживает оба вида исключений, и стали писать два блока *catch* (показаны выше), чтобы перехватывать исключения обоих видов. Если приведенный выше код перекомпилировать для CLR 2.0, второй блок *catch* никогда не будет выполняться и компилятор C# укажет на это, выдав предупреждение «CS1058: A previous catch clause already catches all exceptions. All non-exceptions thrown will be wrapped in a System.Runtime.CompilerServices.RuntimeWrappedException» (CS1058: предыдущий блок *catch* уже перехватывает все исключения. Все сгенерированные исключения будут инкапсулированы в *System.Runtime.CompilerServices.RuntimeWrappedException*).

Есть два пути переноса кода более ранней версии в .NET Framework 2.0:

- объединить код двух блоков *catch* в один блок *catch*, а лишний блок *catch* удалить;
- 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, CLR должна деинкапсулировать объект, не совместимый с CLS, и вызывать ваш код, только если в коде есть

блок *catch*, в котором не определен никакой тип. Такое сообщение для CLR передается путем применения к сборке экземпляра *RuntimeCompatibilityAttribute*:

```
using System.Runtime.CompilerServices;
[assembly:RuntimeCompatibility(WrapNonExceptionThrows = false)]
```

Примечание Этот атрибут оказывает влияние на всю сборку. В одной сборке нельзя совмещать инкапсулированные и неинкапсулированные исключения. Нужно соблюдать особую осторожность при добавлении нового кода (который ожидает от CLR инкапсуляции исключений) в сборку, где есть старый код (в котором CLR не инкапсулирует исключения).

Что же это такое — исключение?

Я постоянно сталкиваюсь с разработчиками, которые считают, что исключение — это какое-то редкое, «исключительное» событие. Я всегда просил их дать определение «исключительного события», на что мне отвечали: «Знаете ли, это нечто неожиданное». И добавляли: «Читая байты из файла, когда-нибудь вы достигнете его конца. Поскольку это ожидаемое событие, оно не должно вызывать исключение. Вместо этого, достигнув конца файла, метод *Read* должен вернуть некоторое особое значение».

А вот мой ответ: «Допустим, у меня есть приложение, которое должно считать из файла структуру размером в 20 байт, но файл оказался лишь 10-байтовым. В этом случае я не ожидаю встретить конец файла при чтении, но это неожиданно происходит. Наверное, здесь должно возникнуть исключение, не так ли?» Фактически большинство файлов содержит структурированные данные. Довольно редко бывает так, что приложение читает из файла байт за байтом, тут же обрабатывая прочитанные байты, пока не достигнет конца файла. Поэтому я думаю, логичнее будет, если при попытке чтения за пределами файла метод *Read* всегда будет генерировать исключение.

Внимание! Многих разработчиков вводит в заблуждение термин «обработка исключений». Они считают, что слово «исключение» имеет отношение к частоте некоторого события. Вполне вероятно, что, конструируя метод *Read* для чтения из файла, такой разработчик подумает: «Все равно при чтении я дойду до конца файла. Поскольку это неизбежно, сделаю-ка я так, чтобы мой метод Read сообщал о достижении конца файла, возвращая специальное значение, и не буду использовать для этого исключение». Здесь проблема в том, что так думает создатель метода *Read*, а не его пользователь.

Во время конструирования разработчик метода *Read* не может знать всех возможных ситуаций, в которых может вызываться его метод, а значит, он не может знать и того, как часто вызвавший метод *Read* будет пытаться читать за пределами файла. Это действительно редкое событие, поскольку большинство файлов содержит структурированные данные.

> Другое распространенное заблуждение — считать термины «исключение» и «ошибка» синонимами. Термин «ошибка» предполагает неверное действие программиста. Но и в этом случае разработчик, конструирующий метод *Read*, не может предугадать, когда его метод будет вызван некорректно с точки зрения приложения. Это может определить лишь тот, кто вызывает метод, поэтому только он может сказать, является ли результат вызова «ошибкой». Поэтому избегайте мышления по принципу «здесь мы будем генерировать исключение для уведомления об ошибке». На самом деле исключение вовсе не обязательно является следствием ошибки, поэтому в этой главе я избегаю термина «обработка ошибок» (кроме этого предложения, конечно).

В предыдущем примечании я попытался разъяснить, как *не следует* понимать термин «исключение», а теперь расскажу, что он *должен* означать. При разработке типа вы сначала пытаетесь представить себе разнообразные ситуации, в которых будет применяться тип. Сначала определяется имя типа — обычно существительное, например *FileStream* (файловый поток) или *StringBuilder* (построитель строк), затем приступают к определению членов (типов данных свойств, параметров методов, возвращаемых значений и т. п.). Особенности определения этих членов становятся программным интерфейсом типа. Эти члены действия обычно обозначаются глаголами — *Read* (считать), *Write* (записать), *Flush* (очистить), *Append* (присоединить), *Insert* (вставить), *Remove* (удалить) и т. п. Если член, обозначающий действие, не может выполнить свою задачу, он должен сгенерировать исключение.

Взгляните на определение класса:

```
internal class Account {
   public static void Transfer(Account from, Account to, Decimal amount) {
      ...
   }
}
```

Метод *Transfer* принимает пару объектов *Account* и значение *Decimal*, идентифицирующее сумму, которую нужно перевести с одного счета на другой. Ясно, что задача *Transfer* — снять деньги с одного счета и начислить их на другой. Метод может потерпеть сбой по многим причинам: аргумент *from* или *to* может быть равным null или не ссылаться на открытый счет, на счете *from* может быть недостаточно средств или на счете *to* может быть так много денег, что добавление заданной суммы приводит к числовому переполнению, кроме того аргумент-счет может быть равным нулю, отрицательной величине или содержать более двух знаков после запятой.

При вызове метода *Transfer* его код должен проверить все перечисленные возможности и, обнаружив любую из них, отказаться от перевода средств и уведомить вызывающий код об отказе путем генерации исключения. Заметьте: метод возвращает *void*. Причина в том, что методу *Transfer*, собственно, нечего возвращать — если он возвращает управление, значит все прошло нормально, в противном случае генерируется исключение.

При вызове метода всегда возможен сбой

Есть масса причин, по которым вызванный метод может сгенерировать исключение:

- если недостаточно памяти в стеке, генерируется исключение *StackOverflowException*;
- если не удается обнаружить сборку, в которой определен тип, генерируется исключение *FileNotFoundException*;
- если IL-код метода не поддается верификации, генерируется исключение *VerificationException*;
- если недостаточно памяти для JIT-компиляции IL-кода, генерируется исключение *OutOfMemoryException*.

Список можно продолжать бесконечно, но самое главное — понять, что исключение может возникнуть в любой момент. Представьте себе такой метод:

```
private void InfiniteLoop() {
      while (true) ;
}
```

Цикл может успешно выполниться 1000 раз, а на 1001 раз может быть сгенерировано исключение. Почему? Ну хотя бы потому, что другой поток может попытаться завершить работу потока, вызвав метод *Abort* типа *Thread*. От этого поток, выполняющий бесконечный цикл, приостанавливается и вынужден сгенерировать исключение *ThreadAbortException*. Честно говоря, в процессе обычной работы CLR в таких решениях, как Microsoft ASP.NET и Microsoft SQL Server, очень часто возникают ситуации, когда прекращается выполнение потока, то есть это вполне рядовое событие.

Класс *System.Exception*

CLR позволяет генерировать объекты исключения любого типа — от *Int32* до *String* (и не только их). Однако в Microsoft решили не заставлять все языки генерировать и перехватывать исключения любого типа, определили тип *System.Exception* и объявили, что каждый CLS-совместимый язык должен быть способен генерировать и перехватывать типы исключений, производные от *System.Exception*, о которых говорят, что они являются CLS-совместимыми. Компилятор C# и многих других языков позволяет коду генерировать только CLS-совместимые исключения.

System.Exception — очень простой тип (табл. 19-1).

Табл. 19-1. Открытые свойства типа *System.Exception*

Свойство	Доступ	Тип	Описание
Message	Только чтение	*String*	Содержит текст сообщения с указанием причины возникновения исключения. Сообщение обычно записывается в журнал, если возникшее исключение не удается обработать. Поскольку конечные пользователи не видят это сообщение, оно должно содержать максимум технических подробностей, чтобы разработчики смогли использовать информацию журнала для устранения ошибок при создании новой версии
Data	Только чтение	*IDictionary*	Ссылка на набор пар «параметр-значение». Обычно код, генерирующий исключение, перед этим добавляет записи в этот набор. Код, перехвативший исключение, может запрашивать записи и использовать эту информацию для обработки исключения
Source	Чтение/запись	*String*	Содержит имя сборки, сгенерировавшей исключение
StackTrace	Только чтение	*String*	Содержит имена и сигнатуры методов, вызов которых привел к возникновению исключения. Очень полезно для отладки
TargetSite	Только чтение	*MethodBase*	Содержит метод, сгенерировавший исключение
HelpLink	Только чтение	*String*	Содержит URL (например, file://C:\MyApp\Help.htm#MyExceptionHelp), указывающий на документацию с толкованием исключения. Согласно всем канонам программирования и безопасности, пользователи никогда не должны видеть информацию необработанных исключений (если только не нужно передать информацию другим разработчикам), поэтому это свойство используется редко
InnerException	Только чтение	*Exception*	Указывает предыдущее исключение, если текущее было сгенерировано при обработке предыдущего исключения. Обычно это поле содержит null. Тип — *Exception* также поддерживает открытый метод *GetBaseException*, просматривающий связный список внутренних исключений и возвращающий исходно сгенерированное исключение

Классы исключений, определенные в FCL

В библиотеке классов .NET Framework Class Library определено множество типов исключений (все они в конечном счете являются потомками *System.Exception*). Следующая иерархия демонстрирует типы, определенные в сборке MSCorLib.dll; еще больше типов исключений определено в других сборках (приложение, позволившее получить эту иерархию, см. в главе 22).

```
System.Exception
    System.ApplicationException
        System.Reflection.InvalidFilterCriteriaException
        System.Reflection.TargetException
        System.Reflection.TargetInvocationException
        System.Reflection.TargetParameterCountException
        System.Threading.WaitHandleCannotBeOpenedException
    System.IO.IsolatedStorage.IsolatedStorageException
    System.Runtime.CompilerServices.RuntimeWrappedException
    System.SystemException
        System.AccessViolationException
        System.AppDomainUnloadedException
        System.ArgumentException
            System.ArgumentNullException
            System.ArgumentOutOfRangeException
            System.DuplicateWaitObjectException
            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.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.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.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
```

Идея Microsoft была такова: *System.Exception* должен быть базовым типом для всех исключений, в том числе для двух других типов, прямых потомков *System.Exception* — *System.SystemException* и *System.ApplicationException*. CLR генерирует исключения, производные от *SystemException*, а все исключения, сгенерированные в приложениях, должны наследовать *ApplicationException*. В этом случае можно создавать блок *catch*, перехватывающий все CLR-исключения или все исключения приложения.

Однако, как мы могли убедиться, это правило соблюдалось не полностью: некоторые типы исключений являются прямыми потомками *Exception* (например, *IsolatedStorageException*), другие CLR-исключения наследуют *ApplicationException* (в частности, *TargetInvocationException*), а третьи исключения приложений являются производными от *SystemException* (к примеру, *FormatException*). Таким образом, все запуталось, и типы *SystemException* и *ApplicationException* уже не несут никакого особого значения. В Microsoft хотели бы вообще убрать их из иерархии классов исключений, но это невозможно, так как приведет к нарушению работы кода, в котором уже используются эти классы.

Генерация исключений

Создавая собственные методы, вы вправе генерировать исключения, если метод не в состоянии выполнить задачу, на которую указывает его имя. При этом нужно ответить на два важных вопроса.

- **Какой производный от *Exception* тип исключения будет генерироваться?** Настоятельно рекомендуется выбирать содержательный тип. Представьте себе код, расположенный выше в стеке вызовов, и как тот код должен узнать, что метод потерпел неудачу, чтобы выполнить корректное восстановление после сбоя. Можно задействовать тип, уже определенный в FCL, но может быть, что в FCL нет типа, полностью соответствующего нужной семантике. Поэтому придется определять собственный тип, непосредственно наследующий *System.Exception*. Если нужно определить иерархию типов исключений, настоятельно рекомендуется создавать максимально плоскую и широкую иерархию, чтобы свести до минимума число базовых классов. Причина в том, что базовые классы действуют так, чтобы обеспечить обработку многих ошибок как одной ошибки, и это обычно опасно. Таким образом, никогда не следует генерировать тип *System.Exception* и надо соблюдать максимальную осторожность при генерации любого другого типа исключений, являющегося базовым классом.
- **Какое строковое сообщение передавать в конструктор типа исключения?** При генерации исключения нужно предоставлять строковое сообщение с детальной информацией о том, почему метод не смог выполнить свою задачу. Если исключение перехватывается и обрабатывается, это строковое сообщение не видно. А вот сообщения необработанных исключений обычно регистрируются в журнале. Необработанное исключение информирует о насто-

ящем дефекте приложения, и разработчик программы должен позаботиться об искоренении дефекта. У конечного пользователя нет исходного текста и возможности исправить недостаток кода и перекомпоновать программу. Вообще говоря, это строковое сообщение конечный пользователь видеть не должен, поэтому оно может содержать массу технических деталей и предоставлять всю необходимую разработчикам информацию для устранения дефекта в коде. Кроме того, так как все разработчики должны понимать английский (ведь языки программирования и FCL-классы и методы содержат много английских слов), обычно нет нужды локализовывать строковые сообщения исключений. Впрочем, вы вправе локализовать строки, если создаете библиотеку классов для разработчиков, говорящих на других языках. В Microsoft локализуют сообщения исключений, генерируемых FCL, так как эту библиотеку классов используют разработчики, разговаривающие на самых разных языках.

Определение собственных классов исключений

Допустим, вы определяете метод, которому передается ссылка на объект, в типе которого должны быть реализованы интерфейсы *IFormattable* и *IComparable*, например так:

```
internal sealed class SomeType {
   public void SomeMethod(Object o) {
      if (!((o is IFormattable) && (o is IComparable))) {
         throw new MissingInterfaceException(...);
      }
      // Здесь находится код, обрабатывающий o.
      ...
   }
}
```

Поскольку в FCL нет подходящего типа исключения, придется определить тип *MissingInterfaceException* самостоятельно. Заметьте: по соглашению имена типов исключений должны заканчиваться словом «Exception». Сначала надо выбрать для него базовый тип. Но какой: *Exception*, *ArgumentException* или совсем другой? Я месяцами размышлял над этим вопросом, но, увы, так и не смог придумать никакого правила, и вот почему.

Если породить *MissingInterfaceException* от *ArgumentException*, то любой существующий код, который перехватывает *ArgumentException*, будет перехватывать и это новое исключение. С одной стороны, это полезная возможность, с другой — ошибка. Это полезно, потому что любой код, перехватывающий все типы исключений, связанные с аргументами (посредством перехвата *ArgumentException*), автоматически будет перехватывать новый вид исключений, связанных с аргументами (*MissingInterfaceException*). Ошибка — поскольку исключение *MissingInterfaceException* идентифицирует новое событие, не предусмотренное кодом, перехватывающим *ArgumentException*. Определяя тип *MissingInterfaceException*, можно подумать, что он так похож на *ArgumentException*, что должен обрабатываться так же. Однако непредвиденные связи подобного рода могут привести к непредсказуемой работе программы.

Если же породить *MissingInterfaceException* прямо от *Exception*, код станет генерировать новый тип исключений, о котором приложение ничего не знает. Скорее всего при этом возникнет необработанное исключение, которое прервет исполнение приложения. Легко считать такую реакцию программы желательной, так как метод не в состоянии выполнить свою задачу и у приложения нет средств для восстановления. Если приложение перехватит новое исключение, «проглотит» его и продолжит работу, результаты могут быть непредсказуемы и слишком высок риск нарушения безопасности.

Необходимость отвечать на подобные вопросы делает разработку приложений больше искусством, чем наукой. Определяя новый тип исключений, продумайте, как будут перехватываться исключения этого типа (не забудьте и о перехвате базового типа этого исключения), затем выберите базовый тип, оказывающий минимальное негативное влияние на вызывающий код.

Определяя собственные типы исключений, вы вольны создавать вложенные иерархии, если это диктуется поставленной задачей. Можно выводить эти иерархии прямо из *Exception* или другого базового типа. Здесь тоже нужно следить, чтобы создаваемая иерархия имела смысл для тех, кто будет вызывать ваш код. Если определяемый тип не будет базовым для других типов исключений, можно пометить его как *sealed*.

Базовый тип *Exception* определяет четыре стандартных конструктора:

- открытый конструктор без параметров (конструктор по умолчанию), создающий экземпляр типа и устанавливающий для всех его полей и свойств значения по умолчанию;
- открытый конструктор, принимающий параметр *String* и создающий экземпляр типа с заданным текстом сообщения;
- конструктор, принимающий в качестве параметров строку (*String*) и экземпляр типа, производного от *Exception*, и создающий экземпляр с заданными текстом сообщения и внутренним исключением;
- закрытый конструктор, принимающий объекты *SerializationInfo* и *StreamingContext*, которые десериализуют экземпляры объекта, производного от *Exception*. Заметьте: этот метод должен быть закрытым, если производный от *Exception* тип является изолированным; также надо позаботиться, чтобы конструктор вызывал такой же конструктор базового класса, чтобы обеспечить корректную десериализацию полей базового класса.

Определяя собственный тип исключений, следует реализовать в нем эти четыре конструктора и вызывать соответствующий конструктор из базового типа. Конечно, ваш тип исключения унаследует все поля и свойства, определенные в *Exception*. Кроме того, к нему можно добавить собственные поля и аргументы. Скажем, в исключение *SystemArgumentException* добавлено свойство типа *String* — *ParamName* (помимо всего, что наследуется от типа *Exception*). Тип *ArgumentException* также определяет новый конструктор (помимо четырех стандартных конструкторов) с дополнительным параметром типа *String*, инициализирующим свойство *ParamName*, которое идентифицирует имя параметра, нарушившего допущения метода.

При перехвате *ArgumentException* можно прочитать свойство *ParamName*, чтобы точно определить имя параметра, вызвавшего сбой. Стоит ли говорить, что это

невероятно удобно при отладке приложения! Если вы на самом деле добавляете поля к собственному типу исключения, проследите, чтобы были определены конструкторы, обеспечивающие их инициализацию. Также не забудьте определить свойства и остальные члены, что возвращают значения полей в код приложения, который перехватывает исключения вашего типа.

Все типы исключений нужно делать сериализуемыми, чтобы обеспечить маршалинг их объектов через границы домена приложения или за пределы машины. Если сделать тип исключения сериализуемым, его можно будет записывать в журнал или базу данных. Чтобы сделать собственный тип исключения сериализуемым, пометьте его атрибутом *[Serializable]* и реализуйте интерфейс *ISerializable* с методом *GetObjectData* (с атрибутом *SecurityPermission*) и специальным конструктором, оба должны принимать параметры *SerializationInfo* и *StreamingContext*. Заметьте: если класс изолированный, конструктор должен быть закрытым, в противном случае конструктор является защищенным. Вот как определять собственный тип исключения:

```
using System;
using System.Text;
using System.Runtime.Serialization;
using System.Security.Permissions;

// Разрешаем сериализацию экземпляров DiskFullException.
[Serializable]
public sealed class DiskFullException : Exception, ISerializable {
   // Создаем закрытое поле.
   private String m_diskpath;

   // Определяем неизменяемое свойство, возвращающее это поле.
   public String DiskPath { get { return m_diskpath; } }

   // Переопределяем открытое свойство Message,
   // чтобы включить в сообщение содержимое поля (если оно задано).
   public override String Message {
      get {
         if (m_diskpath == null) return base.Message;
         StringBuilder msg = new StringBuilder(base.Message);
         msg.AppendFormat(
              " (DiskPath={0}){1}",
              m_diskpath, Environment.NewLine);
         return msg.ToString();
      }
   }

   // Три открытых конструктора.
   public DiskFullException() : base() {  }
   public DiskFullException(String message) : base(message) { }
   public DiskFullException(String message, Exception innerException)
      : base(message, innerException) { }

   // Определяем дополнительные конструкторы,
   // задающие значение поля.
```

```
   public DiskFullException(String message, String diskpath)
      : this(message) { m_diskpath = diskpath; }

   public DiskFullException(String message, String diskpath, Exception innerException)
      : this(message, innerException) { m_diskpath = diskpath; }

   // Один конструктор для десериализации.
   // Так как это изолированный класс, конструктор должен быть закрытым.
   // В противном случае этот конструктор должен быть защищенным.
   private DiskFullException(SerializationInfo info, StreamingContext context)
      : base(info, context) {
      // Десериализация каждого поля.
      m_diskpath = info.GetString("DiskPath");
   }

   // Метод для сериализации; SecurityPermission гарантирует,
   // что вызывающий код получит информацию о внутреннем состоянии этого объекта.
   [SecurityPermission(SecurityAction.Demand, SerializationFormatter = true)]
   public override void GetObjectData(
      SerializationInfo info, StreamingContext context) {
      // Заставляем базовый тип сериализовать свои поля.
      base.GetObjectData(info, context);

      // Сериализуем поля этого типа.
      info.AddValue("DiskPath", m_diskpath);
   }
}
```

Как правильно использовать исключения

Понимать механизм работы исключений важно, но не менее важно понимать, как их разумно использовать. Слишком часто я встречал библиотеки, перехватывающие все исключения без разбора, 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 библиотеки классов* и занимаетесь созданием типов, которые будут использовать другие разработчики, отнеситесь к этим правилам очень серьезно. На вас лежит огромная ответственность за разработку интерфейса, который будет использоваться для широкого спектра приложений. Помните: вы не знаете всех тонкостей кода, который вызываете через делегаты, виртуальные методы или методы интерфейсов. 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 придется туго при использовании типов вашей библиотеки классов.

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 это представляется необходимым. Политику приложения (например, более агрессивный перехват исключений кодом приложения) определяете именно вы.

Проверяйте аргументы своих методов

Для типов, которые являются частью библиотеки повторно используемых классов, настоятельно рекомендуется, чтобы в открытых типах присутствовала проверка правильности параметров открытых и защищенных методов, до начала выполнения какой-либо операции методом. На то есть две причины. Во-первых, это позволяет разработчикам, вызывающим метод, узнать, правильно ли вызывается метод. Во-вторых, если все параметры правильны, вероятность корректного выполнения и возвращения управления методом намного выше, а исключения генерируются реже. Это также означает, что вероятность того, что объекты останутся в непротиворечивом состоянии, больше.

Вспомните открытый статический метод *Account* класса *Transfer*. Он перечисляет средства с одного счета на другой. Если бы метод сразу же не проверял правильность поступающих параметров, метод мог бы успешно снять средства, а затем обнаружить, что входной параметр для остатка на этом счете — null. На этом этапе метод вынужден будет сгенерировать исключение, потому что перечисление невозможно. Однако метод должен также вернуть средства на счет *from*. Если этого не сделать, состояние счета *from* станет некорректным. Наличие ранней проверки параметров в методе облегчает программирование, потому что разработчику не приходится думать и учитывать возможность сбоя транзакции или другие сложные условия.

Реализованный метод повторно используемого класса должен проверять правильность получаемых параметров и, если хотя бы один из них некорректен, генерировать исключение, производное от *SystemArgumentException*. Самые полезные классы исключений, производные от *SystemArgumentException*, — *SystemArgumentNullException*, *SystemArgumentOutOfRangeException* и *System.DuplicateWaitObjectException*. Если ни одно из них не подходит, вы вправе или определить собственный тип исключений, наследующий типу *ArgumentException*, или самостоятельно генерировать *ArgumentException*. Заметьте: *System.ComponentModel.InvalidEnumArgumentException* также производный от *ArgumentException*, однако в Microsoft этот тип исключения считают ошибкой и не рекомендуют его использовать, потому что он относится к необычному пространству имен и определен в System.dll, а не MSCorLib.dll.

Примечание Если ваши методы проверяют правильность входящих параметров, можно использовать в коде меньше вызовов *DebugAssert*. Этот метод верификации нужно использовать для проверки правильности определенных предположений в сборке. Использование верификации позволяет обнаруживать некоторые ошибки программирования на этапе разработки, позволяя исправить код, скомпоновать и протестировать

новую версию. Кода, способного перехватить или выполнить корректное восстановление после сбоя верификации, попросту нет — ведь в финальной версии отсутствуют вызовы *DebugAssert*, а исключения остаются. По этой и другим причинам в сборке нельзя использовать *DebugAssert* для проверки корректности вызова метода из кода другой сборки; для этого применяются исключения. Однако помимо исключений вы вправе включить вызовы *DebugAssert*, так как при сбое верификации можно немедленно подключать отладчик к сбойному коду и легко проверять состояние параметров и локальных переменных.

Раз уж речь зашла о проверке правильности параметров, я хотел бы указать кое-что очень важное. Как известно, метод можно объявить с параметрами как ссылочного, так и значимого типа. В случае значимого типа значение переданного параметра никто, кроме кода самого метода, изменить не может. С другой стороны, объект, на который ссылается параметр ссылочного типа, может быть изменен кодом, не относящимся к методу (это не относится к объектам *String*, так как строки неизменяемы). Это вполне возможно, к примеру, в многопоточном приложении.

Чтобы создаваемая библиотека классов была безопасной и надежной, методы, принимающие изменяющиеся ссылочные типы, должны делать копию параметров, проверять правильность копий и только после этого использовать их в методе. Это гарантирует, что метод будет работать с неизменными входными данными. Следующий код демонстрирует возможные проблемы:

```
#define BADCODE

using System;
using System.Threading;

public static class Program {
   public static void Main() {
      Int32[] denominators = { 1, 2, 3 };

      // Используем другой поток для выполнения работы.
      ThreadPool.QueueUserWorkItem(Divide100By, denominators);

      // ПРОВЕРКА: предоставляем Divide100By шанс
      // проверить корректность элементов массива.
      Thread.Sleep(50);

      // Делаем элемент массива непригодным для использования в методе Divide100By.
      denominators[2] = 0;

      Console.WriteLine("Press <Enter> when you see the results.");
      Console.ReadLine();
   }

   private static void Divide100By(Object o) {
```

```
#if BADCODE
      // Этот код демонстрирует проблему.
      Int32[] denominators = (Int32[]) o;
#else
      // Этот код устраняет проблему, создавая копию массива.
      // Копия проходит проверку и используется в остальной части метода.
      Int32[] denominatorsInput = (Int32[]) o;
      Int32[] denominators = new Int32[denominatorsInput.Length];
      Array.Copy(denominatorsInput, denominators, denominators.Length);
#endif

      // Проверяем все элементы массива, полученного в качестве параметра.
      for (Int32 index = 0; index < denominators.Length; index++) {
         if (denominators[index] == 0)
            throw new ArgumentOutOfRangeException("denominators",
               String.Format("Index {0} contains 0", index));
      }
      Console.WriteLine(
         "All denominators are valid; DivideByZeroException can't occur.");

      // ПРОВЕРКА: предоставляем Main шанс
      // проверить корректность элемента массива.
      Thread.Sleep(100);

      // Все элементы корректны, теперь выполняем работу.
      for (Int32 index = 0; index < denominators.Length; index++) {
         Console.WriteLine("100 / {0} = {1}",
            denominators[index], 100 / denominators[index]);
      }
   }
}
```

После компоновки и выполнения кода получим следующее:

```
All denominators are valid; DivideByZeroException can't occur.
Press <Enter> when you see the results.
100 / 1 = 100
100 / 2 = 50

Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at Program.Divide100By(Object o) in C:\...\ArgumentValidation.cs:line 50
   at System.Threading._ThreadPoolWaitCallback.WaitCallback_Context(Object state)
   at System.Threading.ExecutionContext.Run(ExecutionContext executionContext,
ContextCall
back callback, Object state)
   at System.Threading._ThreadPoolWaitCallback.PerformWaitCallback(Object state)
```

Как видите, элементы массива были должным образом проверены на корректность и метод приступил к обработке входных данных. Но метод *Main* изменил последний элемент массива уже после проверки, поэтому *Divide100By* сгенерировало исключение *DivideByZeroException*, хотя, казалось бы, это невозможно!

В приведенном выше исходном тексте программы есть код, решающий проблему, — достаточно просто удалить строку, определяющую символ *BADCODE*. Теперь *Divide100By* будет создавать копию массива, которая пройдет проверку и будет использоваться в остальной части метода. Теперь исключение *DivideByZeroException* не возникнет. Скомпоновав и выполнив версию кода без определенного BADCODE, получим:

```
All denominators are valid; DivideByZeroException can't occur.
Press <Enter> when you see the results.
100 / 1 = 100
100 / 2 = 50
100 / 3 = 33
```

Блоков *finally* не должно быть слишком много

По-моему, блоки *finally* — прекрасное средство! Они позволяют определять код, который будет гарантированно исполнен независимо от вида исключения, сгенерированного потоком. Блоки *finally* нужны, чтобы выполнить очистку после любой успешно начатой операции прежде, чем вернуть управление или продолжить исполнение кода, расположенного после блока *finally*. Блоки *finally* также часто используют для явного уничтожения любых объектов во избежание утечки ресурсов. Вот пример, в котором весь код, выполняющий очистку (закрывающий файл), размещен в блоке *finally*:

```
using System;
using System.IO;

public sealed class SomeType {
   private void SomeMethod() {

      // Открываем файл.
      FileStream fs = new FileStream(@"C:\Data.bin ", FileMode.Open);
      try {
         // Выводим частное 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, возникло исключение
         // (например, если первый байт файла равен 0) или нет.
         fs.Close();
      }
   }
}
```

Гарантия исполнения кода очистки при любых обстоятельствах настолько важна, что большинство языков поддерживает соответствующие конструкции, облегчающие его программирование. Например, C# поддерживает для этого операторы *lock* и *using*. Эти операторы дают разработчику простой синтаксис, заставляющий компилятор автоматически генерировать блоки *try* и *finally*, помещая код для очи-

стки в блок *finally*. Например, в следующем коде на C# используются возможности оператора *using*. Он короче предыдущего, но при обработке исходного текста из этого и из предыдущего примеров компилятор генерирует идентичный код.

```
using System;
using System.IO;

internal sealed class SomeType {
   private void SomeMethod() {

      // Открываем файл.
      using (FileStream fs =
         new FileStream(@"C:\Data.bin", FileMode.Open)) {

         // Выводим частное от деления 100 на первый байт файла.
         Console.WriteLine(100 / fs.ReadByte());
      }
   }
}
```

Подробнее об операторе *using* я расскажу в главе 20, а об операторе *lock* — в главе 24.

Не всякое исключение следует перехватывать

Распространенная ошибка — слишком частое и неверное 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. Но слишком часто приходится видеть код вроде этого:

```
try {
   // Попытка выполнить код, который, как считает программист,
   // может потерпеть сбой...
}
catch (Exception) {
   ...
}
```

Этот код объявляет, что предвидел *все* исключения *любого* типа и умеет 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.

Кроме того, причиной исключения вполне может оказаться объект в некорректном состоянии. Если библиотечный код перехватит и «проглотит» такое исключение, программа продолжит выполнение, но результаты могут оказаться непредсказуемыми, а также может быть нарушена защита. Лучше, если исключение

останется необработанным, а приложение завершится. (О необработанных исключениях — чуть попозже.) Честно говоря, большинство необрабатываемых исключений обнаруживается в процессе тестирования кода. Исправляют ситуацию, либо предусматривая в коде перехват и обработку такого исключения, либо корректируя код, устраняя причины появления исключения. Число необработанных исключений в окончательной версии программы, предназначенной для выполнения в производственной среде, должно быть минимальным (лучше вообще отсутствовать), а сама программа должна быть исключительно устойчивой.

Примечание В некоторых случаях метод, не способный выполнить задачу, обнаруживает, что состояние некоторых объектов нарушено 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 генерировать исключение, а немедленно выполнять принудительное завершение процесса вызовом метода *FailFast* типа *System.Environment*.

Кстати, вполне *нормально* перехватить исключение *System.Exception* и выполнить определенный код внутри блока *catch* при условии, что в конце этого кода исключение повторно генерируется. Перехват и «проглатывание» (без повторной генерации) исключения *System.Exception* не должно никогда выполняться, так как это приводит к сокрытию сбоев и продолжению работы приложения с непредсказуемыми результатами и нарушениями безопасности. Предоставляемая компанией Microsoft утилита FxCop позволяет обнаружить блоки *catch (Exception)*, в коде которых отсутствует оператор *throw*. Подробнее мы обсудим это далее в этой главе.

Наконец, допускается перехватывать исключение, возникшее в одном потоке, и повторно генерировать его в другом потоке. Такое поведение поддерживает модель асинхронного программирования (см. главу 23). 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 *End*Xxx, чтобы выяснить результат асинхронной операции. Метод *End*Xxx сгенерирует такое же исключение, что и поток из пула, выполнявшего заданную работу. В такой ситуации исключение игнорируется первым потоком, но повторно генерируется потоком, вызывавшим метод *End*Xxx, поэтому оно не скрывается от приложения.

Корректное восстановление после исключения

Некоторые исключения, генерируемые методом, бывают известны заранее. Поскольку это ожидаемые исключения, нужен код, обеспечивающий корректное 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. Но если ячейка со знаменателем содержит 0, то CLR сгенерирует исключение *DivideByZeroException*. Тогда метод перехватывает именно это исключение и возвращает специальную строку, которая будет показана пользователю. Аналогично содержимое ячейки может быть произведением двух других ячеек. Если полученное значение не умещается в отведенное число бит, CLR сгенерирует объект *OverflowException*, а также специальную строку для показа пользователю.

Перехватывая конкретные исключения, нужно полностью понимать вызывающие их обстоятельства и знать типы исключений, производные от перехватываемого типа. Не следует перехватывать и обрабатывать *SystemException* (без повторной генерации), так как нельзя знать все возможные исключения, которые могут быть сгенерированы внутри вашего блока *try* (особенно это касается *OutOfMemoryException* и *StackOverflowException*).

Отмена незавершенных операций при невосстановимых 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 дисковый файл. После сериализации 10 объектов генерируется исключение (скажем, из-за переполнения диска или из-за того, что следующий сериализуемый объект не помечен атрибутом *Serializable*). Теперь исключение будет отфильтровано и передано вызывающему методу, но в каком состоянии останется дисковый файл? А он будет поврежден — ведь в нем находится граф частично сериализованного объекта. Было бы здорово, если бы приложение смогло отменить незавершенные операции, чтобы вернуть файл в исходное состояние, в котором он был до записи сериализованных объектов. Вот правильный способ реализации отмены:

```
public void SerializeObjectGraph(FileStream fs,
   IFormatter formatter, Object rootObj) {

   // Сохранить текущую позицию в файле.
   Int64 beforeSerialization = fs.Position;

   try {
      // Попытаться сериализовать граф объекта и записать его в файл.
      formatter.Serialize(fs, rootObj);
   }
```

```
   catch {  // Перехватываем все и каждое исключение.
      // При ЛЮБОМ отклонении вернуть файл в нормальное состояние.
      fs.Position = beforeSerialization;

      // Усечь файл.
      fs.SetLength(fs.Position);

      // ПРИМЕЧАНИЕ: предыдущий код не помещен в блок finally,
      // так как сброс потока нужен только при сбое сериализации.

      // Уведомить вызывающий код о том, что случилось,
      // сгенерировав ТО ЖЕ САМОЕ исключение повторно.
      throw;
   }
}
```

Чтобы корректно отменить незавершенные операции, нужен код, перехватывающий все исключения. Да, здесь нужно перехватывать *все* исключения, так как тут важен не тип ошибки, а возврат структур данных в согласованное состояние. Перехватив и обработав исключение, не «проглатывайте» его — вызывающий код должен узнать о возникшем исключении. Это делается путем повторной генерации того же исключения. Фактически C# и многие другие языки позволяют без труда сделать это. Достаточно лишь указать единственное ключевое слово C# — *throw*, как показано в предыдущем коде.

Обратите внимание, что в предыдущем примере не указан тип исключения в блоке *catch*, поскольку здесь требуется перехватывать абсолютно все исключения. К счастью, на C# это легко сделать, просто не указывая тип исключения, что заставит оператор *throw* повторно генерировать любое перехваченное исключение.

Сокрытие деталей реализации для сохранения контракта

Порой после перехвата исключения одного типа полезно сгенерировать исключение другого типа, например:

```
public Int32 SomeMethod(Int32 x){
   try {
      return 100 / x;
   }
   catch (DivideByZeroException e) {
      throw new ArgumentOutOfRangeException("x can't be 0", e);
   }
}
```

При вызове методу *SomeMethod* передается значение *Int32*, а метод возвращает частное от деления 100 на переданное ему значение. После входа в метод код проверяет *x* и, если он равен 0, генерирует на этом этапе исключение *ArgumentOutOfRangeException*. Однако такую проверку придется выполнять при каждом вызове метода. Поскольку неявно предполагается, что *x* редко бывает равен 0, эта проверка снижает быстродействие. Итак, этот метод предполагает, что *x* не равен 0, и пытается поделить его на 100. Если же *x* оказался равен 0, перехватывается конкретное исключение — *DivideByZeroException*, после чего оно генерируется повтор-

но, но уже как *ArgumentOutOfRangeException*. Перехват одного исключения и генерация другого позволяют соблюсти соглашение, или контракт с вызывающим кодом. Обратите внимание, что исключение *DivideByZeroException* задано как значение свойства *InnerException* для исключения *ArgumentOutOfRangeException*. Эта дополнительная информация может быть полезной для отладки, так как позволяет программисту узнать, какое исключение случилось на самом деле.

Эта методика позволяет перехватывать исключения, описанные в разделе «Корректное восстановление после исключения». Перехватывая специфические исключения, нужно полностью понимать вызвавшие их обстоятельства и знать типы исключений, производные от перехватываемого типа.

И здесь разработчик библиотеки классов не должен перехватывать *System.Exception* и ему подобные, иначе он просто сведет все типы исключений к одному и потеряет все ценные сведения о реальном исключении. Без этой информации коду, расположенному выше в стеке вызовов, намного сложнее перехватить и обработать специфическое исключение. Дайте коду, расположенному выше в стеке вызовов, шанс перехватить *System.Exception* или иной тип исключений, являющийся базовым типом для более специфичных исключений.

Вообще, сохранение контракта, или соглашений метода, — это единственное, для чего при перехвате одного типа нужно генерировать исключение другого типа. Кроме того, новый тип исключений должен быть конкретным (то есть не должен быть базовым типом для других исключений). Представьте себе тип *PhoneBook*, определяющий метод для поиска телефонного номера по заданному имени. Вот псевдокод этого метода:

```
internal sealed class PhoneBook {
   private String m_pathname;  // Путь к файлу с телефонным справочником.

   // Здесь находятся остальные методы.

   public String GetPhoneNumber(String name) {
      String phone;
      FileStream fs = null;
      try {
         fs = new FileStream(m_pathname, FileMode.Open);
         (Code to read from fs until name is found)
         phone = /* телефонный номер найден */
      }
      catch (FileNotFoundException e) {
         // Генерируем другое исключение, содержащее имя абонента,
         // задав исходное исключение как внутреннее исключение нового.
         throw new NameNotFoundException(name, e);
      }
      catch (IOException e) {
         // Генерируем другое исключение, содержащее имя абонента,
         // задав исходное исключение как внутреннее исключение нового.
         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 увидит 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 генерирует вместо них новое исключение *NameNotFoundException*.

Повторная генерация исключения позволяет вызывающему коду узнать о том, что метод не в состоянии выполнить свою задачу, а тип *NameNotFoundException* дает ему абстрактное представление о причине сбоя. Важно установить внутреннее исключение нового исключения как *FileNotFoundException* или *IOException*, чтобы не потерять его реальную причину, знание которой может быть полезно разработчику типа *PhoneBook*.

Внимание! Используя описанный метод, мы врем вызывающему коду о двух вещах. Во-первых, сообщается неправда о том, что пошло не так. В нашем примере файл не удалось найти, но мы сообщили о невозможности найти имя. Во-вторых, мы врем о месте сбоя. Если бы *FileNotFoundException* могло пробиться в верх стека вызовов, его свойство *StackTrace* говорило бы, что ошибка произошла в конструкторе *FileStream*. Но, когда мы «проглатываем» это исключение и генерируем новое *NameNotFoundException*, след стека будет указывать, что ошибка произошла в блоке *catch*, то есть на расстоянии нескольких строк кода от места возникновения исключения. Это может сильно затруднить отладку, поэтому описанный метод следует использовать с большой 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*. Теперь пользователи будут ожидать, что путь будет предусмотрен контрактом *PhoneBook*.

Вопросы быстродействия

В сообществе программистов вопросы быстродействия, связанные с обработкой исключений, обсуждаются очень часто и активно. По собственному опыту могу

утверждать, что преимущества обработки исключений с запасом перевешивают все потери производительности. В этом разделе мы поговорим о некоторых вопросах быстродействия, связанных с обработкой исключений.

Трудно сравнивать быстродействие обработки исключений и более привычных средств уведомления об исключениях (возврата *HRESULT*, специальных кодов и т. п.). Если вы напишете код, который будет проверять значение, возвращаемое каждым вызванным методом, фильтровать и передавать его коду, вызвавшему метод, то быстродействие приложения сильно снизится. Даже если оставить быстродействие в стороне, объем дополнительного кодирования и потенциальная возможность ошибок будут невероятно велики. 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 размер кода, так и на время исполнения.

С другой стороны, 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 исполняется и во время выполнения, а быстродействие приложения растет.

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. Иногда 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 могут встраивать методы, содержащие обработчики исключений, а другие не регистрируют переменные, если в методе есть обработчик исключений.

Суть в том, что нельзя оценить величину дополнительных издержек, которые влечет 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 Microsoft, предназначенными для внутреннего использования. Я неожиданно обнаружил разительную разницу в быстродействии. 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 быстродействии при использовании обработки исключений. Как я уже сказал, выгоды от этого намного перевешивают вызванное им снижение быстродействия.

Если вам интересно, насколько обработка исключений снижает производительность вашего кода, можно задействовать PerfMon.exe или элемент управления ActiveX System Monitor, который есть в Windows. На снимке экрана видны счетчики, связанные с обработкой исключений, которые устанавливаются при установке .NET Framework (рис. 19-1).

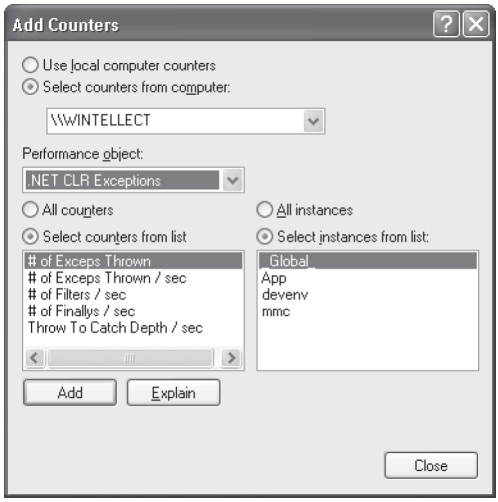


Рис. 19-1. *Счетчики исключений .NET CLR в окне PerfMon.exe*

Когда-нибудь вы столкнетесь с часто вызываемым методом, который часто генерирует исключение. В такой ситуации снижение производительности из-за обработки слишком частых исключений недопустимо велико. В частности, в Microsoft слышали от нескольких клиентов жалобы, что при вызове метода *Parse* класса *Int32* и передаче данных, введенных конечными пользователями, возникал сбой разбора. Так как *Parse* вызывался часто, генерация и перехват исключений сильно снижали общую производительность приложения.

Для решения заявленной клиентами проблемы и в соответствии с принципами, описанными в этой главе, Microsoft добавила в класс *Int32* два метода, которые называются *TryParse*:

```
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 булево значение, которое указывает, можно ли обработать все символы переданной в *Int32* строки (*String*). Эти методы также возвращают выходной параметр *result*. Если метод возвращает *true*, *result* содержит результат преобразования строки в 32-разрядное целое. Если же возвращает *false*, *result* содержит 0, но это значение вряд ли будет использоваться в коде.

Хотел бы прояснить одну вещь. Возвращенное методом *Try*Xxx булево значение *false* указывает на один и только один тип сбоя. Для других сбоев метод может генерировать исключения. Например, *TryParse* класса *Int32* генерирует *ArgumentException*, если параметр неверный, и, конечно же, может генерировать *OutOfMemoryException*, если при вызове *TryParse* произошла ошибка выделения памяти.

Также хотелось бы подчеркнуть, что объектно-ориентированное программирование повышает производительность труда программиста. И не в последнюю очередь за счет запрета на передачу кодов ошибок через члены типа. Иначе говоря, конструкторы, методы, свойства и другие создаются с тем расчетом, что в их работе сбоев не будет. И, при условии правильности определения, в большинстве случаев использования члена сбоев в нем не будет, а, значит, не будет снижения производительности, потому что нет исключений.

Определять типы и их члены надо с тем расчетом, чтобы свести к минимуму вероятность их сбоя в стандартных сценариях их использования. Если вы позже услышите от своих клиентов, что производительность неудовлетворительна из-за генерации множества исключений, то только в этом случае следует подумать о добавлении в тип методов *Try*Xxx. Иначе говоря, сначала надо создать оптимальную объектную модель, а затем, если пользователи окажутся недовольными, добавить в тип несколько методов *Try*Xxx, которые облегчат жизнь пользователям, столкнувшимся с проблемой низкой производительности. Остальные пользователи — те, которые не испытывают проблем с производительностью, могут использовать версию типа без этих методов — ведь созданная вами объектная модель отвечает реалиям их работы.

Необработанные исключения

Итак, при возникновении 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 указывает на ситуацию, которую не предвидел программист, и должно считаться признаком серьезной ошибки в приложении. На этом этапе о недостатке следует уведомить компанию, где разрабатывается приложение, чтобы авторы могли устранить неполадку и выпустить исправленную версию.

Разработчикам библиотек классов даже думать нельзя о возможности возникновения необработанных исключений. Только разработчики приложений должны заботиться о необработанных исключениях, а в приложении должна быть реализована политика, определяющая порядок действия при возникновении необработанных исключений. Microsoft рекомендует разработчикам приложений просто принять политику CLR по умолчанию: при возникновении необработанного исключения открывается диалоговое окно (рис. 19-2), предоставляющее пользователю возможность отправить информацию об ошибке в Microsoft. Этот механизм информирования об ошибках называется Windows Error Reporting (подробнее см. сайт *http://WinQual.Microsoft.com*).

Рис. 19-2. *Диалоговое окно необработанного исключения, позволяющее отправить информацию об ошибке в Microsoft*

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). Последняя нужна, чтобы результаты работы приложений и компонентов были доступны только компании-издателю.

Если вы не хотите, чтобы информация о сбое отправлялась в Microsoft, можете воспользоваться комплектом Shareware Starter Kit (*http://msdn.microsoft.com/vstudio/downloads/starterkits/*). Он позволяет создать и настроить собственный Web-сервер, на который будет поступать информация о необработанных исключениях. В сущности, этот набор позволяет установить, запустить и администрировать собственную, урезанную версию системы Windows Error Reporting.

Впрочем, вы вправе разработать собственную систему получения информации о необработанных исключениях, которая нужна для устранения недостатков программы. При инициализации приложения можно проинформировать CLR, что есть метод, который нужно вызывать каждый раз, когда в каком-либо потоке приложения происходит необработанное исключение.

Можно использовать следующие члены FCL (подробнее см. документацию):

- для всех приложений — событие *UnhandledException* класса *System.AppDomain*;
- для приложений Windows Forms — виртуальный метод *OnThreadException* класса *System.Windows.Forms.NativeWindow*, одноименный виртуальный метод класса *System.Windows.Forms.Application* и событие *ThreadException* класса *System.Windows.Forms.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*.

К сожалению, для разных типов приложений (ASP.NET Web Services, WSE Web Services, Windows Communication Foundation Web Services, Windows Presentation Foundation и т. д.) Microsoft предлагает разные способы получения информации о необработанных исключениях.

В завершение темы хотелось бы сказать несколько слов о необработанных исключениях, которые могут произойти в распределенных приложениях, таких

как Web-сайты и Web-службы. В идеальном мире серверное приложение, в котором случилось необработанное исключение, зарегистрирует 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 состояния (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 — это снижает вероятность успешной хакерской атаки.

Трассировка стека при исключениях

Как вы помните, тип *SystemException* поддерживает открытое неизменяемое свойство *StackTrace*. Блок *catch* может, прочитав это свойство, получить трассировку стека с описанием событий, имевших место непосредственно перед исключением. Эти сведения могут быть чрезвычайно полезны для выявления причин исключения с целью исправления кода. В этом разделе мы обсудим ряд не совсем очевидных моментов, связанных с трассировкой стека.

Свойство *StackTrace* типа *Exception* поистине волшебно. Обращаясь к нему, вы на самом деле вызываете код CLR, причем это свойство не просто возвращает строку. Когда вы создаете новый объект-потомок *Exception*, его свойство инициализируется значением *null*. Если в этот момент попытаться прочитать это свойство, вы получите *null* вместо трассировки стека.

При генерации исключения внутренние механизмы CLR регистрируют место, в котором была исполнена команда *throw*. Когда фильтр перехвата принимает исключение, CLR регистрирует место перехвата исключения. Теперь, если в блоке *catch* обратиться к свойству *StackTrace* сгенерированного объекта исключения, код этого свойства вызовет CLR. При этом CLR построит строку, идентифицирующую все методы, вызванные между возникновением исключения и срабатыванием фильтра, перехватившего исключение.

Внимание! При генерации исключения CLR сбрасывает его начальную точку в стеке, то есть CLR запоминает только место генерации последнего объекта исключения. Следующий код генерирует тот же объект исключения, что был перехвачен им, еще раз, заставляя CLR сбросить начальную точку исключения:

```
private void SomeMethod() {
   try { ... }
   catch (Exception e) {
      ...
      throw e;  // CLR думает, что начало исключения расположено здесь.
                // FxCop сообщает об этом, как об ошибке.
   }
}
```

А вот при повторной генерации исключения (при помощи ключевого слова *throw* без указания типа исключения) CLR не сбрасывает начальную точку в стеке. Следующий код генерирует тот же объект исключения повторно, не заставляя CLR сбрасывать начальную точку исключения:

```
private void SomeMethod() {
   try { ... }
   catch (Exception e) {
      ...
      throw;    // Это не влияет на сведения о начальной точке исключения,
                // зарегистрированные CLR.
                // FxCop НЕ считает это ошибкой.
   }
}
```

Фактически эти два фрагмента кода различаются только позицией в стеке, где, по мнению CLR, было сгенерировано исключение. К сожалению, когда вы впервые или повторно генерируете исключение, Windows не сбрасывает начальную точку в стеке. Поэтому, если исключение остается необработанным, позиция в стеке о котором сообщается по механизму Windows Error Reporting — это всего лишь место последней первичной или повторной генерации исключения, несмотря на то, что CLR знает место возникновения исходного исключения. Это очень неудобно, так как усложняет отладку приложений, потерпевших сбой в «полевых» условиях, то есть у клиента. Некоторым разработчикам это не нравится до такой степени, что они иначе реализуют свой код, чтобы трассировочный след отражал истинное место возникновения исключения:

```
private void SomeMethod() {
   Boolean trySucceeds = false;
   try {
      ...
      trySucceeds = true;
   }
   finally {
      if (!trySucceeds) { /* здесь находится код перехвата */ }
   }
}
```

В строке, возвращаемой свойством *StackTrace*, нет ни одного метода, расположенного в стеке вызовов выше точки, в которой объект исключения был принят блоком *catch*. Чтобы получить полную трассировку стека от начала потока до вызова обработчика исключения, надо воспользоваться типом *System.Diagnostics.StackTrace*, в котором определен ряд свойств и методов, позволяющих разработчику программно манипулировать трассировкой стека и составляющими ее фреймами.

Можно создавать объект *StackTrace* разными конструкторами: одни создают объект *StackTrace*, представляющий фреймы от начала потока до точки создания этого объекта, другие инициализируют фреймы объекта *StackTrace* с помощью объекта, производного от *Exception*.

Если CLR найдет отладочные символы для ваших сборок, то в строку, возвращаемую свойством *StackTrace* типа *System.Exception* или методом *ToString* типа

System.Diagnostics.StackTrace, будут включены пути к файлу с исходным текстом и номера строк, что невероятно удобно для отладки.

В трассировке стека можно обнаружить отсутствие некоторых методов из стека вызовов. Причина — в способности JIT-компилятора встраивать методы во избежание издержек, связанных с вызовом и возвратом управления отдельными методами. Многие компиляторы (включая компилятор C#) поддерживают параметр командной строки */debug*. При его наличии компиляторы добавляют в результирующую сборку информацию, запрещающую JIT-компилятору встраивать методы для этой сборки, что позволяет получать более полные трассировки, более понятные отлаживающему код разработчику.

Примечание JIT-компилятор проверяет специализированный атрибут *System.Diagnostics.DebuggableAttribute*, которым помечена сборка. Компилятор C# применяет этот атрибут автоматически. Если параметру *isJITOptimizerDisabled* конструктора этого атрибута задать true, то JIT-компилятор не будет встраивать методы сборки. Параметр командной строки */debug* компилятора C# задает этому параметру значение true. Применив к методу атрибут *System.Runtime.CompilerServices.MethodImplAttribute*, можно запретить JIT-компилятору встраивание методов как в отладочных, так и рабочих компоновках программы. Следующее определение метода иллюстрирует запрет встраивания методов:

```
using System;
using System.Runtime.CompilerServices;

internal sealed class SomeType {

   [MethodImpl(MethodImplOptions.NoInlining)]
   public void SomeMethod() {
      ...
   }
}
```

Отладка исключений

В отладчике из Microsoft Visual Studio есть специальная поддержка исключений: выберите в меню элемент **Debug.Exceptions** — появится диалоговое окно (рис. 19-3).

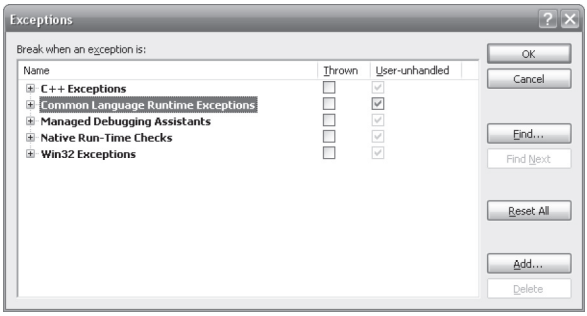


Рис. 19-3. *Различные виды исключений в диалоговом окне Exceptions из Visual Studio*

Здесь показаны типы исключений, поддерживаемые Visual Studio. Раскрыв ветвь **Common Language Runtime Exceptions**, вы увидите пространства имен, поддерживаемые отладчиком из Visual Studio (рис. 19-4).

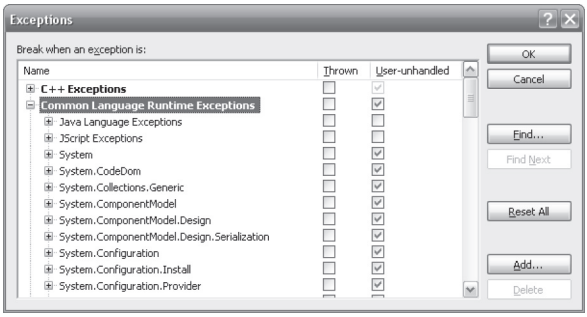


Рис. 19-4. *Исключения CLR, организованные по пространствам имен, в диалоговом окне Exceptions в Visual Studio*

Раскрыв пространство имен, вы увидите все определенные в нем типы, производные от *SystemException* (рис. 19-5).

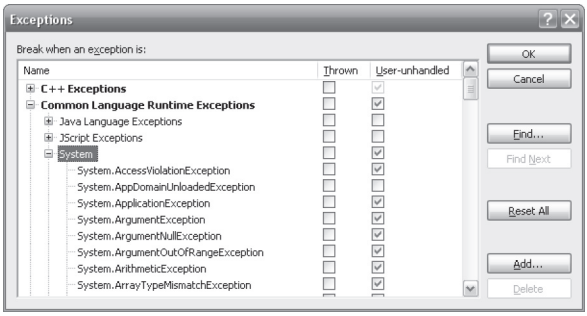


Рис. 19-5. *Диалоговое окно Exceptions в Visual Studio с CLR-исключениями, определенными в пространстве имен* System

Если для какого-либо исключения установлен флажок *Thrown*, при возникновении этого исключения отладчик остановится. На этот момент CLR еще не приступила к поиску подходящего блока *catch*. Такая возможность полезна для отладки кода, ответственного за перехват и обработку соответствующего исключения. Также она нужна в ситуации, когда вы подозреваете, что компонент или библиотека проглатывают или повторно генерируют 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 генерации 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; приложение продолжает работу, как обычно.

Вы можете определить собственные типы и добавить их в окно, щелкнув кнопку **Add**. В результате откроется такое окно (рис. 19-6):

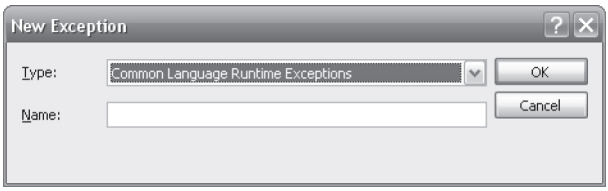


Рис. 19-6. *Передача Visual Studio сведений о собственном типе исключений*

В этом окне сначала выбирают тип исключения — **Common Language Runtime Exceptions**, а затем вводят полное имя собственного типа исключений. Заметьте: вводимый тип не обязательно должен быть потомком *SystemException*, 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 все типы, производные от *Exception*, в отладчик Visual Studio. А еще хорошо бы дополнительно идентифицировать каждый тип именем сборки, это решило бы проблему одноименных типов из разных сборок.

Кроме того, было бы здорово, если бы это диалоговое окно позволяло отдельно выбирать типы, не производные от *Exception*, чтобы добавлять любые типы исключений, не совместимые с CLS. Однако очень не рекомендуется использовать исключения, не совместимые с CLS, поэтому на реализации этой функции я не настаиваю.

ГЛАВА 20

Автоматическое управление памятью (сбор мусора)

Эта глава о том, как управляемые приложения создают новые объекты, как управляемая куча распоряжается временем жизни этих объектов и как освобождается занятая ими память. Я вкратце объясню, как работает сборщик мусора общеязыковой среды CLR, и расскажу о различных проблемах с его производительностью. Я также покажу, как разрабатывать приложения, наиболее эффективно использующие память.

Основы работы платформы, поддерживающей сбор мусора

Любая программа использует те или иные ресурсы: файлы, буферы в памяти, пространство экрана, сетевые подключения, базы данных и т. д. В объектно-ориентированной среде каждый тип идентифицирует некоторый ресурс, доступный программе. Для использования любого ресурса должна быть выделена память для представления этого типа. Для получения доступа к ресурсу нужно сделать следующее.

1. Выделить память для типа, представляющего ресурс, вызвав команду *newobj* промежуточного языка, которая генерируется при использовании оператора *new* в C#.
2. Инициализировать выделенную память, чтобы установить начальное состояние ресурса и сделать 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. Разрушить состояние ресурса, чтобы выполнить очистку (об этом чуть позже).
5. Освободить память, за что отвечает исключительно сборщик мусора.

Эта, на первый взгляд, простая парадигма была одной из основных причин ошибок при программировании. 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 последствия, ни периодичность появления. Прочие ошибки довольно просто исправить, заметив

неправильную работу приложения. Но эти две ошибки вызывают утечку ресурсов (чем увеличивают потребление памяти) и повреждение объектов (дестабилизируя систему), приводя к непредсказуемой работе приложения. Для облегчения поиска таких ошибок специально разработано множество инструментов: Task Manager (Диспетчер задач) в Microsoft Windows, ActiveX-элемент System Monitor, NuMega BoundsChecker от компании Compuware и Purify от компании Rational.

Правильно управлять ресурсами весьма сложно и утомительно. Это занятие отвлекает разработчиков от решения реальных задач. Если бы у разработчиков был механизм, упрощающий управление памятью, которое так изматывает программистов! Такой механизм есть — это сбор мусора.

Сбор мусора 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 п. 4 нашего списка — разрушить состояние ресурса с целью его очистки. Чтобы корректно очистить ресурс, разработчик должен написать код, «умеющий» правильно выполнить это действие. Этот код нужно поместить в методы *Finalize*, *Dispose* и *Close*, но об этом чуть позже. Однако, как будет видно далее, и здесь сборщик мусора может оказаться полезным, во многих случаях позволяя разработчикам опускать п. 4.

Кроме того, многие типы, такие как *String*, *Attribute*, *Delegate* и *Exception*, представляют ресурсы, не требующие никакой особой очистки. Так, чтобы полностью очистить ресурс типа *String*, достаточно уничтожить массив символов в памяти его объекта.

С другой стороны, когда нужно освободить память объекта типа, представляющего (или инкапсулировавшего) неуправляемый или «родной» ресурс, например файл, подключение к базе данных, сокет, мьютекс, битовую карту, значок и т. п., всегда требуется выполнить код очистки. Далее в этой главе я покажу, как правильно определять типы, требующие явной очистки, и как использовать типы, поддерживающие явную очистку. А теперь посмотрим, как выделяется память и инициализируются ресурсы.

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 не нужно удалять объекты из управляемой кучи — они удаляются автоматически, когда становятся не нужными приложению. Естественно, сразу возникает вопрос: «Как управляемая куча узнает, что объект больше не нужен приложению?». Об этом чуть позже.

В настоящее время используют несколько алгоритмов сбора мусора. Каждый из них оптимизирован для конкретной среды, обеспечивая максимальную производительность. В этой главе основное внимание уделяется алгоритму сбора мусора, который применяется в CLR Microsoft .NET Framework. Начнем с основных понятий.

При инициализации процесса CLR резервирует непрерывную область адресного пространства, которой изначально не соответствует никакой физической памяти. Эта область адресного пространства и есть управляемая куча. Куча также под-

держивает указатель, назовем его *NextObjPtr*. Он указывает адрес в куче, по которому будет выделена память для следующего объекта. Изначально *NextObjPtr* указывает на базовый адрес этой зарезервированной области адресного пространства.

IL-команда *newobj* создает объект. Многие языки (в том числе C#, C++/CLI и Microsoft Visual Basic) поддерживают оператор *new*, заставляющий компилятор создать команду *newobj* и поместить ее в IL-код метода. Получив команду *newobj*, CLR выполняет следующие действия.

1. Подсчитывает число байт, необходимых для размещения полей типа (и полей всех его базовых типов).
2. Прибавляет к полученному значению число байт, необходимых для размещения системных полей объекта. У каждого объекта есть пара таких полей: указатель на объект-тип и *SyncBlockIndex*. 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 доступных байт, чтобы выделить память для объекта (и при необходимости передает память). Если в управляемой куче достаточно места для объекта, ему выделяется память, начиная с адреса, на который указывает *NextObjPtr*, а занимаемые им байты обнуляются. Далее 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.

На рис. 20-1 показана управляемая куча с тремя объектами: A, B и C. Новый объект будет размещен по адресу, указанному *NextObjPtr* (сразу после объекта C).

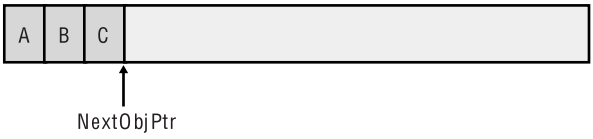


Рис. 20-1. *Только что инициализированная управляемая куча с тремя объектами*

Для сравнения посмотрим, как выделяется память в куче исполняющей среды C. Чтобы выделить в куче память для объекта, исполняющая среда C должна пройти по связному списку 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. Но в управляемой куче последовательно созданные объекты гарантированно будут расположены друг за другом.

Во многих приложениях объекты, выделяемые примерно в одно время, обычно связаны теснее, и к ним часто обращаются примерно в одно время. Так, очень часто сразу после объекта *FileStream* создается объект *BinaryWriter*. Затем приложение обращается к объекту *BinaryWriter*, внутренний код которого использует *FileStream*. В среде, поддерживающей сбор 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 кеше процессора. Приложение сможет получать доступ к этим объектам с феноменальной скоростью, так как процессор будет выполнять большинство своих операций без промахов кеша, 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 — следствие очень существенного допущения о бесконечности адресного пространства и ресурсов памяти. Конечно же, это нелепо, поэтому у управляемой кучи должен быть механизм, который сделает такое допущение возможным. Это сборщик мусора. Посмотрим, как он работает.

Когда приложение вызывает оператор *new*, чтобы создать объект, в области, где выделяется объект, может не хватать свободного адресного пространства. Куча определяет недостающую память, добавляя байты, необходимые для объекта, к адресу, на который указывает *NextObjPtr*. Если результирующее значение выходит за пределы адресного пространства, куча заполнена и надо собрать мусор.

Внимание! Это весьма упрощенное описание работы сборщика мусора. На самом деле сбор мусора начинается при заполнении поколения 0. Некоторые сборщики используют поколения — механизм, единственное назначение которого состоит в улучшении производительности. Его идея такова: недавно созданные объекты составляют новое поколение, а созданные в начале жизненного цикла приложения — старое. Объекты из поколения 0 были созданы недавно и еще не проходили проверку через алгоритм сборщика мусора. Объекты, оставшиеся после сборки мусора, переходят в другое поколение (например, в поколение 1). Деление объектов на поколения позволяет сборщику мусора ограничиться при сборе несколькими поколениями вместо всей управляемой кучи. Подробнее о поколениях рассказывается далее в этой главе, а пока для простоты считайте, что сбор мусора происходит при заполнении кучи.

Алгоритм сбора мусора

Сборщик мусора проверяет, есть ли в куче объекты, которые больше не используются приложением. Если да, можно освободить занятую ими память. (Если после сбора мусора в куче нет свободной памяти, оператор *new* генерирует исключение *OutOfMemoryException*.) Откуда сборщик знает, используется объект приложением или нет? Ясно, что этот вопрос не из легких.

У каждого приложения есть набор *корней* (root). Корень — это адрес, по которому находится указатель 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. 20-2. (Примечания справа на рисунке поясняют, как машинный код соотносится с исходным.)

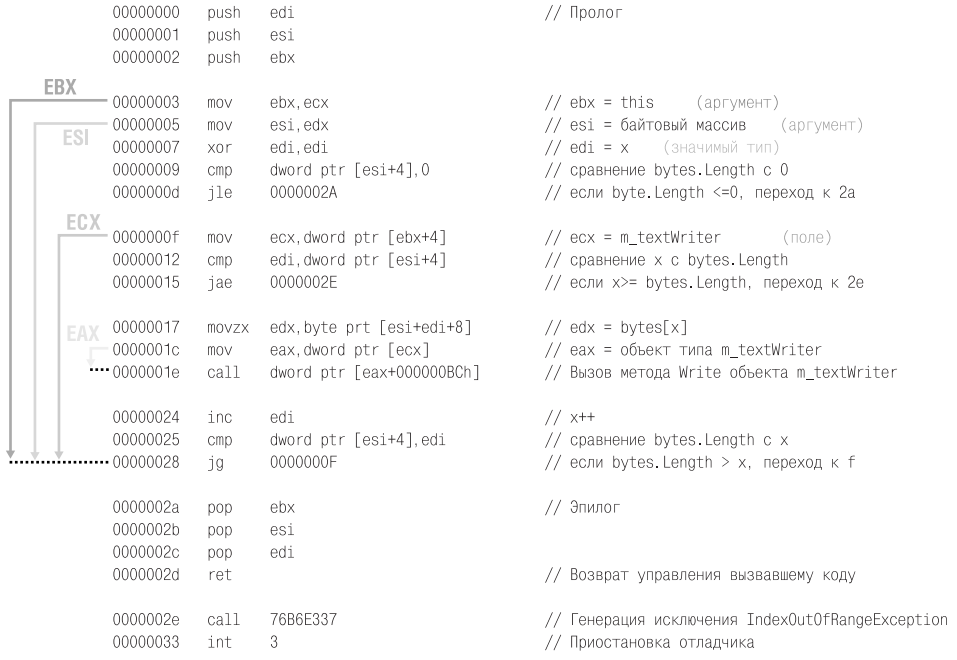


Рис. 20-2. *Машинный код, созданный JIT-компилятором, с иерархией корней*

При генерации машинного кода JIT-компилятор также создает внутреннюю таблицу. Логически каждая строка таблицы указывает диапазон смещений байт 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: регистр EAX является корнем между 0x0000001c и 0x0000001e. Регистр EDI служит для хранения величины типа *Int32*, представленной переменной *x* в исходном коде. *Int32* — это значимый тип, поэтому JIT-компилятор не считает регистр EDI корнем.

Метод *WriteBytes* довольно прост, и все используемые в нем переменные могут быть зарегистрированы. Более сложный метод может использовать 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 методов экземпляров классов 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 сбор мусора начался во время исполнения кода со смещением 0x00000017 в методе *WriteBytes*, сборщик мусора знал бы, что объекты, на которые ссылается регистр EBX (аргумент *this*), ESI (аргумент *bytes*) и ECX (поле *m_textWriter*), — это корни, 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 объекты-типы.

Начиная работу, сборщик предполагает, что все объекты в куче — мусор. Иначе говоря, он предполагает, что в стеке потока нет переменных, ссылающихся на объекты в куче, а также что на объекты в куче не ссылаются регистры процессора и статические поля. Затем сборщик переходит к этапу сбора мусора, называемому *маркировка* (marking). Он проходит по стеку потока и проверяет все корни. Если окажется, что корень ссылается на объект, в поле *SyncBlockIndex* этого объекта будет включен бит — именно так объект *маркируется*. Например, сборщик мусора может найти локальную переменную, указывающую на объект в куче. На рис. 20-3 показана куча с несколькими объектами, в которой корни приложения напрямую ссылаются на объекты A, C D и F. Все эти объекты маркируются. При маркировке объекта D сборщик мусора замечает, 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. При попытке пометить объект, уже помеченный ранее, сборщик мусора останавливается. Это нужно по двум причинам. Во-первых, заметно повышается быстродей-

ствие, так как сборщик проходит набор объектов не больше одного раза, а во-вторых, исключается возможность бесконечных циклов, возникающих из-за замкнутых связных списков объектов.

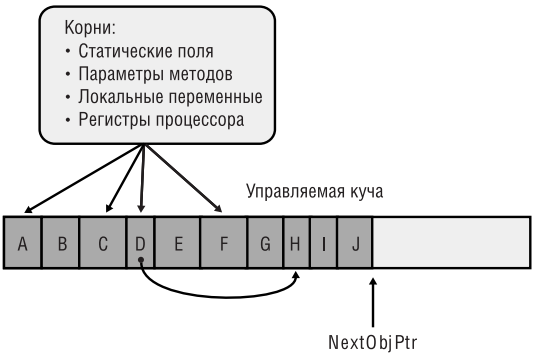


Рис. 20-3. *Управляемая куча перед сбором мусора*

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 сбора мусора, называемому *сжатие* (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 таким образом кучу.

Естественно, перемещение объектов в памяти делает все переменные и регистры процессора, содержащие указатели на объекты, недействительными. 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. 20-4 показана управляемая куча после сбора мусора.

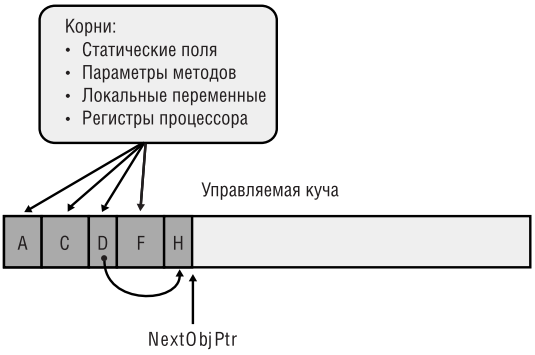


Рис. 20-4. *Управляемая куча после сбора мусора*

Как видите, сбор мусора сильно бьет по производительности — это основной недостаток управляемой кучи. Однако не следует забывать, что сбор мусора начинается только после заполнения поколения 0, а до этого управляемая куча работает намного быстрее, чем куча исполняющей среды C. Наконец, ряд оптимизаций сборщика мусора CLR существенно повышает производительность сбора мусора.

Программист должен извлечь для себя несколько важных уроков из этого обсуждения. Для начала: больше не нужен код, управляющий временем жизни объектов, используемых приложением. И заметьте: исключается возможность двух видов ошибок, описанных в начале этой главы. Во-первых, отсутствует утечка объектов, так как все объекты, недоступные от корней приложения, рано или поздно уничтожает сборщик мусора. Во-вторых, невозможно получить доступ к освобожденному объекту, поскольку доступные объекты не освобождаются, а если объект недостижим, приложение не получит к нему доступ. Также, поскольку при сборе мусора происходит сжатие памяти, управляемые объекты не фрагментируют виртуальное адресное пространство процесса. Иногда это становилось серьезным препятствием при работе с неуправляемой кучей, но в управляемой куче это уже не проблема. Есть одно исключение: при хранении больших объектов все-таки возможна фрагментация кучи (об этом см. далее в этой главе).

Примечание Если сбор мусора — такая замечательная вещь, то почему его нет в ANSI C++, спросите вы. Дело в том, что сборщику мусора необходима возможность определять корни приложения и находить все указатели на объекты. Проблема с неуправляемым C++ в том, что он допускает приведение указателей одного типа к другому, поэтому нельзя узнать, на что ссылается указатель. Управляемая куча в CLR всегда знает настоящий тип объекта и при помощи метаданных способна определить, какие члены объекта ссылаются на другие объекты.

Сбор мусора и отладка

На рис. 20-2 обратите внимание, что на аргумент *bytes* метода (хранящийся в регистре ESI) нет ссылок после команды процессора со смещением 0x00000028. Это значит, что объект-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 объект-массив). Иначе говоря, как только объект становится недостижимым, он становится кандидатом на удаление — не все объекты просуществуют до окончания времени жизни метода. Эта особенность может иметь интересные последствия для приложения. 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) {
      // Отображение 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*, а переменная *t* ссылается на этот объект. Поскольку объект-таймер существует, таймер должен срабатывать. Заметьте: в методе *TimerCallback* я принудительно вызвал сбор мусора с помощью *GC.Collect()*.

Сразу после запуска сборщик мусора предполагает, что все объекты в куче недостижимы (то есть являются мусором), и объект *Timer* в том числе. 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. Поэтому таймер останавливается, а метод *TimerCallback* вызывается всего один раз.

Допустим, вы используете отладчик для *Main*, а сбор мусора выполняется сразу после того, как переменной *t* присвоено значение адреса нового объекта *Timer*. Затем, предположим, вы попытаетесь просмотреть объект, на который ссылается *t*, с помощью окна Quick Watch отладчика. Что же произойдет? 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 Microsoft предложила свое решение.

Когда JIT-компилятор компилирует IL-код метода в машинный код, он проверяет, была ли сборка, в которой определен метод, скомпилирована без оптимизаций и выполняется ли процесс с отладчиком. Если и то и другое верно, JIT-компилятор генерирует внутреннюю таблицу корней метода так, чтобы искусственно продлить время жизни всех переменных до завершения метода. Иначе говоря, JIT-компилятор занимается самообманом, убеждая себя в том, что переменная *t* в *Main* должна жить до завершения метода. Поэтому при сборе мусора сборщик посчитает, что *t* — это корень, а объект *Timer*, на который она ссылается, доступен. Объект *Timer* не будет удален, и метод *TimerCallback* будет вызываться, *Console.ReadLine* не вернет управление и существует *Main*. В этом легко убедиться. Просто запустите тот же исполняемый файл в режиме отладки — метод *TimerCallback* будет вызываться регулярно.

Теперь перекомпилируйте программу из командной строки, указав параметр */debug+* компилятора C#. Запустив полученный исполняемый файл, вы увидите, что метод *TimerCallback* вызывается периодически — даже если 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*. Если JIT-компилятор обнаружит, что этот атрибут задан, он также скомпилирует метод, искусственно продлевая время жизни всех переменных до окончания метода. При указании параметра компилятора */debug+* компилятор C# добавляет этот атрибут в готовую сборку. Учтите, что параметр */optimize+* компилятора C# может вновь включить оптимизацию, поэтому при выполнении такого эксперимента этот параметр компилятора указывать не следует.

Таким образом, JIT-компилятор помогает своевременно выполнить отладку. Теперь можно запустить приложение в обычном режиме (без отладчика), а если метод будет вызван, JIT-компилятор искусственно увеличит время жизни переменных до окончания метода. Затем, если к процессу будет добавлен отладчик, можно добавить точку прерывания в ранее скомпилированный метод и изучить переменные.

Теперь вы знаете, как создать программу, которая работает на этапе отладки и не работает корректно 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() {
   // Создаем объект Timer, вызывающий 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
   // это выражение будет удалено).
   t = null;
}
```

Все равно после компиляции этого кода (без параметра */debug+*) и запуска полученного исполняемого файла (без отладчика) оказывается, что метод *TimerCallback* также 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. А вот как правильно изменить метод *Main*.

```
public static void Main() {
   // Создаем объект Timer, вызывающий 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 будет удалена сборщиком мусора до возврата 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, а программа работает корректно. Это объясняется тем, что объект, на который ссылается переменная *t*, не должен удаляться, чтобы можно было вызвать экземплярный метод *Dispose* (а значение *t* нужно передать *Dispose* как аргумент *this*).

Использование завершения для освобождения машинных ресурсов

Итак, мы познакомились с азами механизма сбора мусора и управляемой кучи, а также способов освобождения памяти объекта. На наше счастье, большинство типов требует для работы только память. Так, типы *String*, *Attribute*, *Delegate* и *Exception* всего лишь манипулируют байтами в памяти. Но есть типы посложнее — чтобы выполнять полезную работу, им помимо памяти необходимы машинные ресурсы.

Например, типу *System.IO.FileStream* нужно открыть файл (машинный ресурс) и сохранить его описатель. Затем его методы *Read* и *Write* работают с файлом при помощи этого описателя. Аналогично тип *System.Threading.Mutex* открывает мьютекс, объект ядра Windows (машинный ресурс), сохраняет его описатель и использует его при вызове методов объекта *Mutex*.

Завершение (finalization) — это механизм, поддерживаемый CLR, который позволяет 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 других, должен поддерживать завершение. Для этого в типе реализуют метод *Finalize*. Определив, что объект стал мусором, сборщик вызывает метод *Finalize* объекта (если он есть). Иначе говоря, если в типе реализован метод *Finalize*, это означает, что все его объекты имеют право на исполнение «последнего желания перед экзекуцией».

Группа разработчиков C# из Microsoft посчитала, что метод *Finalize* отличается от остальных и для него нужен специальный синтаксис в языке программирования (аналогично, в C# специальный синтаксис используется для определения конструктора). Так, для определения метода *Finalize* в C# перед именем класса нужно добавить тильду (~), как в следующем примере.

```
internal sealed class SomeType {
   // Это метод Finalize.
   ~SomeType() {
      // Здесь помещается код метода Finalize.
   }
}
```

Скомпилировав этот код и проверив полученную сборку с помощью 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 *Finalize*, похож на синтаксис деструктора C++. Действительно, в предыдущих версиях спецификации C# этот метод назывался *деструктором* (destructor). Однако метод *Finalize* 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 C# приведет к детерминированному уничтожению объектов типа, как это происходит в C++. Но CLR не поддерживает детерминированное уничтожение, поэтому C# не может предоставить этот механизм.

Во второй версии спецификации C# метод с таким синтаксисом официально назван *завершителем* (finalizer). Группа разработчиков C# также рассматривала возможность изменения синтаксиса этого метода, чтобы отказаться от применения тильды (~), но это сделало бы непригодными имеющиеся программы. Поэтому изменилось лишь название, а синтаксис оставлен прежним.

Метод *Finalize* обычно вызывает Win32-функцию *CloseHandle*, передавая ей описатель машинного ресурса. В типе *FileStream* определено поле описателя файла, указывающее на этот машинный ресурс. В типе *FileStream* также определен метод *Finalize*, внутренний код которого вызывает *CloseHandle*, передавая последнему поле описателя файла. Это гарантирует, что собственный описатель файла будет закрыт, когда управляемый объект *FileStream* станет мусором. Если у типа, служащего оболочкой для машинного ресурса, нет метода *Finalize*, машинный ресурс не будет закрыт, и возникнет утечка ресурса, продолжающаяся до завершения процесса, при котором ОС освобождает машинные ресурсы.

Гарантированное завершение с использованием типов *CriticalFinalizerObject*

Для удобства разработчиков в пространстве имен *System.Runtime.ConstrainedExecution* определен класс *CriticalFinalizerObject* следующего вида.

```
public abstract class CriticalFinalizerObject {
   protected CriticalFinalizerObject() { /* здесь нет никакого кода */ }

   // Далее следует метод Finalize.
   ~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 CLR наделяет этот класс тремя замечательными возможностями.

- При первом создании любого объекта, производного от типа *CriticalFinalizerObject*, CLR автоматически запускает JIT-компилятор, компилирующий все методы *Finalize* в иерархии наследования. Компиляция этих методов после создания объекта гарантирует, что машинные ресурсы освободятся, как только объект станет мусором. Без компиляции метода *Finalize* возможно лишь выделение и использование ресурсов, но не их освобождение. При недостатке памяти CLR не сможет найти достаточно памяти для компиляции метода *Finalize* — тогда метод не будет исполнен, что приведет к утечке машинных ресурсов. Или же ресурсы не будут освобождены, если код в методе *Finalize* содержит ссылку на тип в другой сборке, которая не была обнаружена CLR.
- CLR вызывает метод *Finalize* типов, производных от *CriticalFinalizerObject*, после вызова методов *Finalize* типов, непроизводных от *CriticalFinalizerObject*. Благодаря этому классы управляемых ресурсов, имеющие метод *Finalize*, могут успешно обращаться к объектам, производным от *CriticalFinalizerObject*, в их методах *Finalize*. Так, метод *Finalize* класса *FileStream* может сбросить данные из буфера памяти на диск в полной уверенности, что дисковый файл все еще открыт.
- CLR вызывает метод *Finalize* типов, производных от *CriticalFinalizerObject*, если домен приложения (AppDomain) был аварийно завершен хост-приложением (например, Microsoft SQL Server или приложением Microsoft ASP.NET). Это также гарантирует, что машинные ресурсы будут освобождены даже в том случае, когда хост-приложение больше не доверяет работающему внутри него управляемому коду.

Тип *SafeHandle* и его производные

В Microsoft понимают, что из всех машинных ресурсов чаще всего используются ресурсы Windows, а также то, что работа с большинством ресурсов Windows выполняется через описатели (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 в методах поясняют, что они делают.)

```
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable {
   // Это описатель машинного ресурса.
   protected IntPtr handle;

   protected SafeHandle(IntPtr invalidHandleValue, Boolean ownsHandle) {
      this.handle = invalidHandleValue;
      // Если 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 вполне 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 ownsHandle равно false, возвратить управление.
      // Установка флага, означающего, что этот ресурс был освобожден.
      // Вызов виртуального метода ReleaseHandle.
      // Вызов GC.SuppressFinalize(this), отменяющий вызов Finalize.
      // Если ReleaseHandle равно true, управление возвращается коду.
      // Запуск ReleaseHandleFailed Managed Debugging Assistant (MDA).
   }

   // Здесь вполне подходит стандартная реализация метода Finalize.
   // 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 Finalize.
   }

   public Boolean IsClosed {
      get {
         // Возврат флага, показывающего, был ли ресурс освобожден.
      }
   }

   public abstract Boolean IsInvalid {
      get {
         // Производный класс переопределяет это свойство.
         // Реализация должна вернуть значение true, если значение описателя
         // не представляет ресурс (обычно это значит,
         // что описатель равен 0 или -1).
      }
   }

   // Эти три метода имеют отношение к безопасности и подсчету ссылок.
   // Подробнее о них в конце этого раздела.
   public void     DangerousAddRef(ref Boolean success) {...}
   public IntPtrDangerousGetHandle() {...}
   public void     DangerousRelease() {...}
}
```

Говоря о классе *SafeHandle*, прежде всего нужно отметить, что он наследует классу *CriticalFinalizerObject* — гарантия того, что CLR будет обращаться с ним не так, как с другими классами. Во-вторых, это абстрактный класс: предполагается, что будет создан еще один производный от *SafeHandle* класс, который переопределит защищенный конструктор, абстрактный метод *ReleaseHandle* и абстрактное свойство *IsInvalid* метода-аксессора *get*.

В Windows большинство описателей являются недействительными, если их значение равно 0 или -1. Пространство имен *Microsoft.Win32.SafeHandle* содержит еще один вспомогательный класс *SafeHandleZeroOrMinusOneIsInvald* следующего вида.

```
public abstract class SafeHandleZeroOrMinusOneIsInvalid : SafeHandle {
   protected SafeHandleZeroOrMinusOneIsInvalid(Boolean ownsHandle)
      : base(IntPtr.Zero, ownsHandle) {
   }

   public override Boolean IsInvalid {
      get {
         if (base.handle == IntPtr.Zero) return true;
         if (base.handle == (IntPtr) (-1)) return true;
      return false;
      }
   }
}
```

Заметьте: класс *SafeHandleZeroOrMinusOneIsInvalid* — абстрактный, поэтому надо создать дочерний класс, который переопределит защищенный конструктор и абстрактный метод *ReleaseHandle*. В платформе Microsoft .NET Framework есть два открытых класса, производных от *SafeHandleZeroOrMunusOneIsInvalid* — *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. Единственная причина наличия у различных классов похожих реализаций — обеспечение безопасности типов: компилятор не позволит использовать файловый описатель в качестве аргумента метода, принимающего описатель блокировки, и наоборот.

Жаль, что в платформе .NET Framework нет дополнительных классов, служащих оболочкой различных машинных ресурсов, например таких, как *SafeProcessHandle*, *SafeThreadHandle*, *SafeTokenHandle*, *SafeFileMappingHandle*, *SafeFileMapViewHandle* (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 так далее.

Все эти классы (а также некоторые другие) есть в библиотеке FCL. Но широкой аудитории известны лишь *SafeFileHandle* и *SafeWaitHandle*. Все остальные являются внутренними классами MSCorLib.dll или System.dll. Microsoft не афишировала эти классы, чтобы не выполнять их полное тестирование и не тратить время на их документирование. Если же вам в работе потребуется любой из этих классов, рекомендую воспользоваться инструментом 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 помощью типов *SafeHandle*

Как уже было показано, классы, производные от *SafeHandle*, необычайно полезны, поскольку они гарантируют освобождение машинного ресурса в процессе сбора мусора. Стоит добавить, что у *SafeHandle* есть еще две функциональные особенности. Во-первых, когда типы, производные от *SafeHandle*, 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);

   // 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, что этот код был неустойчив к сбоям. После вызова метода *CreateEventBad* (создающего ресурс машинного события), возможна ситуация, когда исключение *ThreadAbortException* генерируется до присвоения переменной *handle* описателя. В таких редких случаях в управляемом коде образуется утечка машинного ресурса. И событие можно закрыть одним способом — завершив процесс.

С выходом версии 2.0 .NET Framework стало возможным использовать класс *SafeHandle* для устранения такой возможной утечки ресурсов. Обратите внимание, что прототип метода *CreateEventGood* возвращает *SafeWaitHandle*, а не *IntPtr*. При вызове *CreateEventGood* CLR вызывает Win32-функцию *CreateEvent*. Когда функция *CreateEvent* возвращает управление управляемому коду, CLR знает, что *SafeWaitHandle* является производным от *SafeHandle*. Поэтому CLR автоматически создает экземпляр класса *SafeWaitHandle*, передавая ему значение описателя, полученное от *CreateEvent*. Обновление объекта *SafeWaitHandle* и присвоение описателя происходят в неуправляемом коде, который не может быть прерван исключением *ThreadAbortException*. Теперь в управляемом коде не может возникнуть утечка этого машинного ресурса. В итоге, объект *SafeWaitHandle* удаляется сборщиком мусора и вызывается его метод *Finalize*, обеспечивающий освобождение ресурса.

И, наконец, классы, производные от *SafeHandle*, гарантируют, что никто не может воспользоваться возможными брешами в защите. Проблема в том, что один из потоков может пытаться использовать машинный ресурс, освобождаемый другим потоком. Это называется атакой с повторным использованием описателей. Класс *SafeHandle* предотвращает нарушение безопасности за счет механизма подсчета ссылок. В классе *SafeHandle* определено закрытое поле, выполняющее роль счетчика. Когда производному от *SafeHandle* объекту присваивается корректный описатель, 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, так 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 увеличит значение счетчика перед вызовом и уменьшит значение счетчика сразу после вызова. Конечно, работа счетчика выполняется 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 будет освобожден.

При написании или вызове кода, работающего с описателем, 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 описателю выполняется через метод *DangerousGetHandle*.

И, конечно, я должен упомянуть о классе *CriticalHandle*, также определенном в пространстве имен *System.RuntimeInteropServices*. Он работает точно так же, как и *SafeHandl*e, 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 если высокая производительность крайне необходима и оправдывает некоторое ослабление защиты.

Применение завершения к управляемым ресурсам

Внимание! Некоторые полагают, что к управляемым ресурсам не следует применять завершение. В принципе, я с ними согласен. Поэтому этот раздел можно и не читать. Применение завершения к управляемым ресурсам — это высший пилотаж в разработке кода, прибегать к которому нужно лишь в крайних случаях. Необходимо досконально знать код, вызываемый из метода *Finalize*. Более того, должна быть уверенность в том, что поведение этого кода не изменится с выходом новых версий и что код, вызываемый из метода *Finalize*, не использует другой объект, который мог быть завершен ранее.

Завершение обычно используется исключительно для освобождения машинного ресурса, но иногда оно бывает полезным и для управляемых ресурсов. Вот класс, который заставляет компьютер давать звуковой сигнал каждый раз, когда сборщик мусора начинает свою работу.

```
internal sealed class GCBeep {
   // Это метод Finalize.
   ~GCBeep() {
      // Идет завершение – дать сигнал.
      Console.Beep();

      // Если домен приложения не выгружается, а процесс не завершается,
      // создать новый объект, завершение которого произойдет
      // во время следующего сбора мусора.
      if (!AppDomain.CurrentDomain.IsFinalizingForUnload() &&
      !Environment.HasShutdownStarted)
      new GCBeep();
   }
}
```

Для использования этого класса достаточно создать один его экземпляр. Затем при каждом сборе мусора будет вызываться метод *Finalize* объекта, который вызывает *Beep* и создает новый объект *GCBeep*. Метод *Finalize* этого объекта *GCBeep*

будет вызван при следующем сборе мусора. Вот пример программы, в которой использован класс *GCBeep*.

```
public static class Program {
   public static void Main() {
      // После создания объекта GCBeep каждый раз, когда
      // начинается сбор 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];
      }
   }
}
```

Также учтите, что метод *Finalize* будет вызван, даже если конструктор экземпляра этого типа сгенерирует исключение. Таким образом, в методе *Finalize* не стоит 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 сгенерировать исключение.
      m_fs = new FileStream(filename, FileMode.Create);

      // Сохранить имя этого файла.
      m_filename = filename;
   }

   // Это метод Finalize.
   ~TempFile() {
      // 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 сгенерировать 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 Finalize.
            // О SuppressFinalize см. далее в этой главе.
            GC.SuppressFinalize(this);

            // Уведомить вызывающий код об ошибке.
            throw;
        }
    }

    // Метод Finalize.
    ~TempFile() {
        // Условный оператор теперь не нужен, поскольку этот код
        // выполняется только после успешного исполнения конструктора.
        File.Delete(m_filename);
    }
}
```

При конструировании типа лучше избегать использования метода *Finalize* по ряду причин.

- Выделение памяти для объектов, поддерживающих завершение, занимает больше времени, так как указатели на них должны размещаться в списке завершения.
- Объекты, поддерживающие завершение, переходят в старшие поколения, что увеличивает нагрузку на память и не позволяет освободить память объекта в тот момент, когда сборщик мусора определил его как мусор. 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 завершение, замедляют работу приложения, потому что каждое удаление объекта при сборке мусора требует дополнительного расхода ресурсов.

Также учтите невозможность контролировать момент исполнения метода *Finalize*. Методы *Finalize* запускаются при сборе мусора, который может произойти в тот момент, когда приложению нужно больше памяти. Также CLR не гарантирует определенного порядка вызова методов *Finalize*. Поэтому лучше не создавать метод *Finalize*, который обращается к другим объектам, в типе которых определен метод *Finalize*, — эти объекты могли уже быть завершены. Обращаться к экземплярам значимого или ссылочного типа, в которых не определен метод *Finalize*, вполне допустимо. Также следует быть осторожным при вызове статических методов, поскольку их внутренний код может обращаться к завершенным объектам, что сделает их работу непредсказуемой.

Когда вызываются методы Finalize

Методы *Finalize* вызываются при завершении сбора мусора, который происходит в результате одного из пяти следующих событий.

- **Заполнение поколения 0** приводит к запуску сборщика мусора. Это событие намного чаще остальных приводит к вызову метода *Finalize*, так как является естественным следствием создания новых объектов во время работы кода приложения.
- **Явный вызов статического метода *Collect* объекта *System.GC*** Код может явно запросить сбор мусора у CLR. Хотя Microsoft настоятельно не рекомендует так поступать, порой принудительный сбор мусора имеет смысл.
- **Windows сообщает о нехватке памяти** Для общего мониторинга системной памяти CLR использует Win32-функции *CreateMemoryResourceNotification* и *QueryMemoryResourceNotification*. Если Windows сообщает о нехватке памяти, CLR запускает сбор мусора, чтобы освободить нерабочие объекты и уменьшить рабочий набор процесса.
- **Выгрузка домена приложения CLR** Выгружая домен приложения, CLR считает, что ни один объект в нем не является корнем, и выполняет сбор мусора всех поколений. (О доменах приложения см. главу 21.)
- **Закрытие CLR** CLR завершает работу после нормального завершения работы процесса (в отличие от внешнего завершения, например диспетчера задач). При этом CLR считает, что в процессе нет корней, и вызывает метод *Finalize* для всех объектов в управляемой куче. Учтите, что CLR не пытается сжать или освободить память, потому что процесс завершается, а Windows освобождает всю занятую им память.

CLR использует особый выделенный поток для вызова методов *Finalize*. В случае первых четырех событий, если метод *Finalize* зацикливается, выделенный поток блокируется и вызов методов *Finalize* прекращается. Это очень плохо, потому что приложение не сможет освободить память, занятую объектами, у которых есть метод *Finalize*, и в итоге будет испытывать нехватку памяти.

В случае пятого события каждому методу *Finalize* дается примерно 2 секунды на то, чтобы вернуть управление программе. Если он не успевает, CLR просто завершает процесс и методы *Finalize* больше не вызываются. А если для вызова методов *Finalize* всех объектов требуется более 40 секунд, CLR также просто завершит процесс.

Примечание Указанные значения тайм-аута верны на момент написания этой книги, но ничто не мешает Microsoft изменить их в будущем. Код метода *Finalize* может создавать новые объекты. Если это происходит при закрытии CLR, CLR продолжает уничтожение объектов и вызов их методов *Finalize*, пока не кончатся объекты или не пройдет 40 секунд.

Вспомните тип *GCBeep*, показанный выше. Если объект *GCBeep* завершится в результате первого, второго или третьего события (см. список выше), будет создан новый объект *GCBeep*. Это нормально, поскольку приложение продолжает работать, предполагая, что в дальнейшем еще будет выполняться сбор мусора. Но,

если объект *GCBeep* завершится в результате четвертого или пятого события, новый объект *GCBeep* не должен создаваться, так как это произойдет при выгрузке домена приложения или закрытии CLR. Если такие новые объекты все же будут созданы, CLR придется сделать много бесполезной работы, так как она будет продолжать вызывать методы *Finalize*.

Чтобы предотвратить создание новых объектов *GCBeep*, метод *Finalize* объекта *GCBeep* вызывает метод *IsFinalizingForUnload* объекта *AppDomain* и запрашивает свойство *HasShutdownStarted* объекта *System.Environment*. Метод *IsFinalizingForUnload* возвращает *true*, если метод *Finalize* объекта вызван в процессе выгрузки домена приложения. Свойство *HasShutdownStarted* возвращает *true*, если метод *Finalize* объекта вызван в процессе завершения приложения.

Внутренний механизм завершения

На первый взгляд, в завершении нет ничего сложного: вы создаете объект, а когда его подбирает сборщик мусора, вызывается метод *Finalize* этого объекта. Но на самом деле завершение сложнее, чем кажется.

Когда приложение создает новый объект, оператор *new* выделяет для него память из кучи. Если в типе объекта определен метод *Finalize*, прямо перед вызовом конструктора экземпляра типа указатель на объект помещается в *список завершения* (finalization list) — внутреннюю структуру данных, управляемую сборщиком мусора. Каждая запись этого списка указывает на объект, для которого нужно вызвать метод *Finalize*, прежде чем освободить занятую им память.

На рис. 20-5 показана куча с несколькими объектами. Одни достижимы из корней приложения, другие — нет. При создании объектов C, E, F, I и J система обнаружила в их типах методы *Finalize* и добавила указатели на эти объекты в список завершения.

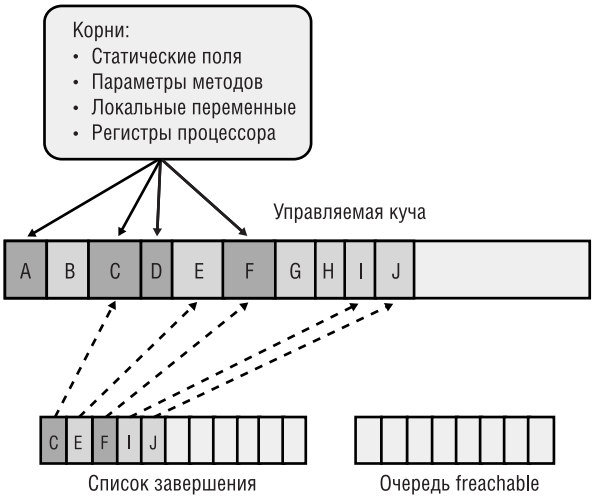


Рис. 20-5. *Управляемая куча с указателями в списке завершения*

Примечание Хотя в *System.Object* определен метод *Finalize*, CLR его игнорирует, то есть, если при создании экземпляра типа метод *Finalize* этого типа унаследован от *System.Object*, созданный объект не считается подлежащим завершению. Метод *Finalize* объекта *Object* должен переопределяться в одном из типов-потомков.

Сначала сборщик мусора определяет, что объекты B, E, G, G, I и J — это мусор. Сборщик сканирует список завершения в поисках указателей на эти объекты. Обнаружив указатель, он удаляет его из списка завершения и добавляет в конец очереди freachable — еще одной внутренней структуры данных сборщика мусора. Каждый указатель в этой очереди идентифицирует объект, готовый к вызову своего метода *Finalize*. После сбора мусора управляемая куча выглядит, как показано на рис. 20-6.

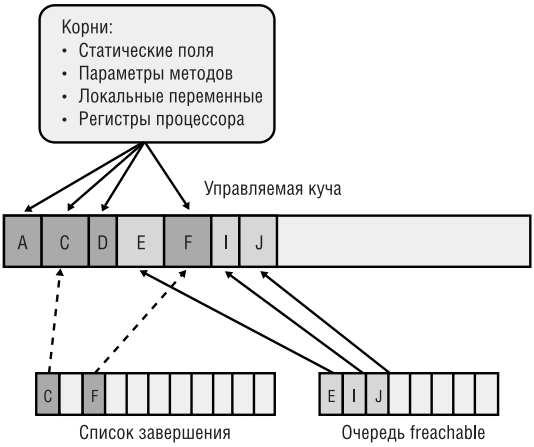


Рис. 20-6. *Управляемая куча с указателями, перемещенными из списка завершения в очередь freachable*

На рис. 20-6 видно, что занятая объектами B, G и H память была освобождена, поскольку у них нет метода *Finalize*. Однако память, занятую объектами E, I и J, освободить нельзя, так как их методы *Finalize* еще не были вызваны.

В CLR есть особый высокоприоритетный поток, выделенный для вызова методов *Finalize*. Отдельный поток нужен для предотвращения возможных проблем с синхронизацией потоков, которые могли бы возникнуть при использовании вместо него одного из потоков приложения с обычным приоритетом. Когда очередь freachable пуста (это ее обычное состояние), этот поток бездействует. Но как только в ней появляются элементы, он активизируется, последовательно удаляет элементы из очереди, вызывая соответствующий метод *Finalize*. Особенности работы данного потока запрещают исполнять в методе *Finalize* любой код, имеющий какие-либо допущения о потоке, исполняющем код. Например, в методе *Finalize* следует избегать обращения к локальной памяти потока.

Возможно, в будущем, CLR будет поддерживать несколько завершающих потоков. Поэтому следует избегать создания кода, содержащего допущение, что методы *Finalize* вызываются последовательно. Иначе говоря, если код в методе *Finalize*

затрагивает общее состояние, придется использовать блокировку синхронизации потоков. При наличии всего лишь одного завершающего потока могут возникнуть проблемы с производительностью и масштабируемостью в ситуации, когда завершаемые объекты распределяются между несколькими процессорами и лишь один поток исполняет методы *Finalize* — он может просто не успеть.

Взаимодействие списка завершения и очереди freachable само по себе замечательно, но сначала я расскажу, как эта очередь получила свое название. Очевидно, буква «f» означает «finalization», то есть завершение: каждая запись в очереди freachable — это ссылка на объект в управляемой куче, для которого должен быть вызван метод *Finalize*. Вторая часть имени, «reachable», означает, что эти объекты достижимы. Можно рассматривать очередь freachable и просто как корень, подобно статическим полям, которые являются корнями. Таким образом, если объект находится в очереди freachable, он достижим и не является мусором.

Короче говоря, если объект недостижим, сборщик считает его мусором. Далее, когда сборщик перемещает ссылку на объект из списка завершения в очередь freachable, объект более не считается мусором, и занятую им память нельзя освободить. По мере маркировки объектов из очереди freachable объекты, на которые ссылаются их поля ссылочного типа, также рекурсивно помечаются — все эти объекты должны пережить сбор мусора. На этом этапе сборщик завершил поиск мусора, и некоторые объекты, идентифицированные как мусор, перестали считаться таковым: они как бы воскресли. Сборщик мусора сжимает освобожденную память, а особый поток CLR очищает очередь freachable, выполняя метод *Finalize* для каждого объекта.

Вызванный снова, сборщик обнаруживает, что завершенные объекты стали мусором, так как ни корни приложения, ни очередь freachable больше на них не указывают. Память, занятая этими объектами, попросту освобождается. Важно понять, что для освобождения памяти, занятой объектами, требующими завершения, сбор мусора нужно выполнить дважды. На самом деле может понадобиться и больше сборов мусора, поскольку объекты переходят в следующее поколение (но об этом чуть позже). На рис. 20-7 показан вид управляемой кучи после второго сбора мусора.

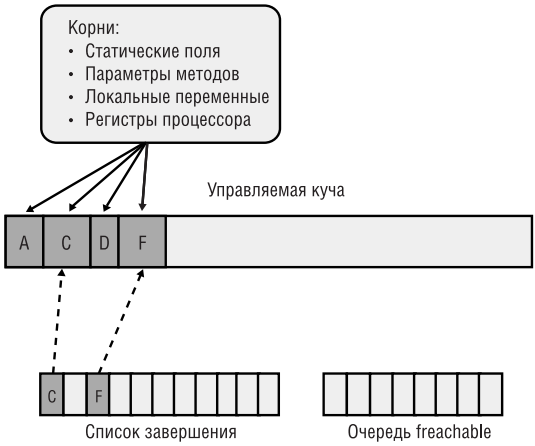


Рис. 20-7. *Состояние управляемой кучи после второго сбора мусора*

Модель освобождения ресурсов: принудительная очистка объекта

Метод *Finalize* невероятно 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, и, 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. Так, нужно открыть соединение с базой данных, извлечь из нее ряд записей и закрыть соединение. Нежелательно оставлять соединение открытым до следующего сбора мусора, особенно потому, что он может произойти через несколько часов, а то и дней после извлечения записей из базы данных.

В типах, поддерживающих возможность детерминированного уничтожения или закрытия, реализована *модель освобождения ресурсов* (dispose pattern). Она определяет соглашения, которым должен следовать разработчик, определяющий тип, поддерживающий явную очистку. Кроме того, если тип поддерживает модель освобождения ресурсов, разработчик-пользователь типа будет точно знать, как явно уничтожить объект, ставший ненужным.

Примечание Каждый тип, в котором определен метод *Finalize*, должен поддерживать и модель освобождения ресурсов, описанную в этом разделе, чтобы у пользователей типа было больше возможностей управлять временем жизни ресурса. Однако существуют типы, у которых есть только модель освобождения ресурсов и нет метода *Finalize*. В эту категорию попадает, например, класс *System.IO.BinaryWriter*. В разделе «Интересная проблема с зависимостью» объясняется, почему сделано такое исключение.

Выше я показал класс *SafeHandle*. Он реализует метод *Finalize*, который гарантирует, что машинный ресурс, оболочкой которого является некий объект, закрывается (или освобождается), когда этот объект удаляется сборщиком мусора. Однако разработчик, использующий объект *SafeHandle*, может явно закрыть машинный ресурс, потому что класс *SafeHandle* реализует интерфейс *IDisposable*.

Рассмотрим еще раз класс *SafeHandle*, но для краткости сосредоточимся на его фрагментах, связанных с моделью освобождения ресурсов.

```
// Реализация интерфейса IDisposable сигнализирует пользователям
// этого класса о том, что у него есть модель освобождения ресурсов.
public abstract class SafeHandle : CriticalFinalizerObject, IDisposable {

   // Этот открытый метод можно вызвать для детерминированного
   // уничтожения ресурса.
   // Этот метод реализует 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 сборе мусора этот метод Finalize вызывается,
   // чтобы закрыть ресурс.
   ~SafeHandle() {
      // Вызов метода, реально выполняющего очистку.
      Dispose(false);
   }

   // Общий метод, реально выполняющий очистку.
   // Его вызывают методы Finalize, 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, а не завершается.
         // Поэтому в этом условном операторе обращение к полям,
         // ссылающимся на другие объекты, безопасно для кода,
         // так как метод Finalize этих объектов еще не вызван.

         // Классу SafeHandle здесь делать ничего не нужно.
      }

      // Выполняется уничтожение/закрытие или завершение объекта.
      // Происходит вот что.
      // Если ресурс уже освобожден, просто возвращается управление.
      // Если ownsHandle равно false, возвращается управление.
      // Устанавливается флаг, указывающий, что данный ресурс был освобожден.
      // Вызов виртуального метода ReleaseHandle.
      // Вызов GC.SuppressFinalize(this), чтобы запретить вызов Finalize.
   }
}
```

Реализацию модели освобождения ресурсов нельзя назвать тривиальной. А теперь я объясню, что весь этот код делает. 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 модель освобождения ресурсов. Проще говоря, этот тип поддерживает открытый метод *Dispose*, не принимающий параметров, который можно явно вызвать для освобождения ресурса, оболочкой которого является объект. Учтите, что память, занятая самим объектом, при этом не освобождается в управляемой куче. Сборщик мусора по-прежнему отвечает за освобождение памяти объекта, и нельзя сказать на-

верняка, когда он это сделает. Оба метода *Dispose* и *Close*, не принимающие параметров, должны быть открытыми и невиртуальными.

Примечание Возможно, вы заметили, что этот класс *SafeHandle* также поддерживает открытый метод *Close*, просто вызывающий *Dispose*. Для удобства некоторые классы, поддерживающие модель освобождения, заодно поддерживают метод *Close*, но для модели освобождения он необязателен. Скажем, класс *System.IO.FileStream* поддерживает как модель 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 модель освобождения ресурсов.

Внимание! Если в классе определено поле, тип которого реализует модель освобождения ресурсов, она также должна быть реализована в этом классе. Метод *Dispose* должен уничтожать объекты, на которые ссылается это поле. Это позволяет при использовании этого класса вызывать для него *Dispose*, который, в свою очередь, освобождает ресурсы, занятые самим объектом. На самом деле это одна из главных причин, по которым в типах должна быть реализована модель освобождения ресурсов, а не метод *Finalize*.

Например, в классе *BinaryWriter* реализована модель освобождения ресурсов. При вызове *Dispose* для объекта *BinaryWriter*, он (*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*: путем вызова метода *Dispose* либо *Close* или же с помощью сборщика мусора, вызывающего метод *Finalize* объекта. Код очистки помещается в отдельный защищенный виртуальный метод, который также называют *Dispose*, но он принимает булев 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 модель 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 генерации исключений).

Примечание Возможна ситуация, когда несколько потоков одновременно вызывают методы *Dispose/Close* для одного объекта. Однако модель 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, вы-

зывать *Dispose/Close* не следует. Лучше подождать, пока сборщик мусора не определит, что объект больше не нужен, а затем освободить ресурс.

Во время вызова метода *Finalize* параметр *disposing* метода *Dispose* приравнивается к *false*. Это запрещает методу *Dispose* исполнять любой код, ссылающийся на другие управляемые объекты, в классах которых реализован метод *Finalize*. Представьте, что во время завершения работы CLR вы пытаетесь выполнить запись в объект *FileStream* в методе *Finalize*. Это может не сработать, так как у *FileStream* может уже быть вызван метод *Finalize*, который закроет соответствующий дисковый файл.

С другой стороны, при вызове в коде метода *Dispose* или *Close* параметр *disposing* метода *Dispose* должен приравниваться к *true*. Это указывает на выполнение явного закрытия, а не завершения объекта. 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 булев 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 булев метод *Dispose*, чтобы переопределить код очистки. В производном классе не будет реализован не принимающий параметров метод *Dispose* или *Close*, и он не переопределит метод *Finalize*. Этот производный класс просто унаследует реализацию всех этих методов. Учтите, что переопределенный производным классом метод *Dispose*, принимающий булев параметр, должен вызывать булев метод *Dispose* базового класса, что позволит базовому классу выполнить всю необходимую очистку. То же можно сказать и о типе *FileStream*, использованном в примере, — он является производным от *Stream*, в котором реализованы метод *Close* и метод *IDisposable.Dispose*, не принимающий параметров. *FileStream* просто переопределяет метод *Dispose*, принимающий булев параметр, чтобы очистить поле *SafeHandle*, являющееся оболочкой для неуправляемого файлового ресурса.

Внимание! Здесь вы должны быть в курсе некоторых проблем с управлением версиями. Если в версии 1 в базовом типе 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 будет знать, как вызывать методы базового типа, а у базового типа не будет шансов выполнить свою очистку корректно. С другой стороны, если в версии 1 интерфейс *IDisposable* реализован в базовом типе, его нельзя будет убрать из последующих версий, потому что иначе производный тип будет пытаться вызвать методы, более не существующие в базовом типе.

Обратите также внимание на ту часть кода, где внутри принимающего булево значение метода *Dispose* вызывается статический метод *SuppressFinalize* типа *GC*. Если код, где используется объект *SafeHandle*, явно вызывает *Dispose* или *Close*, не нужно выполнять его метод *Finalize*, потому что его исполнение было бы бесполезной попыткой повторно освободить ресурс. Вызов *SuppressFinalize* устанавли-

вает битовый флаг, связанный с объектом, на который ссылается его единственный параметр *this*. Когда этот флаг установлен, CLR запрещено перемещать указатель на объект из списка завершения в очередь freachable, что не дает вызвать метод *Finalize* этого объекта и гарантирует, что объект не доживет до следующего сбора мусора. Учтите, что класс *SafeHandle* вызывает *SuppressFinalize*, даже когда объект находится в процессе завершения. Ничего плохого в этом нет — ведь объект уже в процессе завершения.

Использование типов, поддерживающих модель освобождения ресурсов

Итак, вы познакомились с реализацией модели освобождения ресурсов в типах — посмотрим, как разработчики используют подобные типы. Оставим *SafeHandle* и поговорим теперь о более распространенном классе *System.IO.FileStream*. Класс *FileStream* позволяет открыть файл, прочитать и записать в него байты и закрыть его. При создании объекта *FileStream* вызывается Win32-функция *CreateFile*, возвращаемый описатель 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, нужно написать код, который создает временный файл, записывает в него байты, после чего удаляет файл. Можно начать с такого кода.

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Создать байты для записи во временный файл.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Создать временный файл.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Записать байты во временный файл.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Удалить временный файл.
      File.Delete("Temp.dat");   // Генерируется исключение IOException.
   }
}
```

К сожалению, если скомпоновать и запустить этот код, он, скорее всего, не будет работать. Дело в том, что вызов статического метода *Delete* объекта *File* приказывает Windows удалить открытый файл, поэтому *Delete* генерирует исключение *System.IO.IOException* с таким сообщением: «The process cannot access the file «Temp.dat» because it is being used by another process» («Процесс не может обратиться к файлу Temp.dat, потому что он используется другим процессом»).

Знайте, что иногда файл все же удаляется! Если другой поток инициировал сбор мусора между вызовами *Write* и *Delete*, поле *SafeFileHandle* объекта *FileStream* вызывает свой метод *Finalize*, который закроет файл и разрешит работу *Delete*. Вероятность такой ситуации очень мала, поэтому в 99 случаях из 100 приведенный выше код завершится неудачей.

К счастью, в классе *FileStream* реализована модель освобождения ресурсов, что позволяет изменить исходный текст программы так, чтобы она явно закрывала файл. Вот исправленный текст.

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Создать байты для записи во временный файл.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Создать временный файл.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Записать байты во временный файл.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Явно закрыть файл после записи.
      fs.Dispose();

      // Удалить временный файл.
      File.Delete("Temp.dat");  // Такой оператор будет работать всегда.
   }
}
```

Единственное отличие здесь в том, что я добавил вызов метода *Dispose* объекта *FileStream*. Метод *Dispose* вызывает метод *Dispose*, принимающий как параметр *Boolean*, который, в свою очередь, вызывает *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* также поддерживает 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() {
      // Создать байты для записи во временный файл.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Создать временный файл.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);
```

```
      // Записать байты во временный файл.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Явно закрыть файл после записи.
      fs.Close();

      // Удалить временный файл.
      File.Delete("Temp.dat");  // Теперь этот метод работает всегда.
   }
}
```

Примечание Не забывайте, что метод *Close* официально не входит в модель освобождения ресурсов — одни типы поддерживают его, а другие — нет.

Учтите, что вызов *Dispose* или *Close* — это просто способ заставить объект выполнить самоочистку в определенное время. Эти методы не управляют памятью, занятой объектом в управляемой куче. Это значит, что можно вызвать методы объекта даже после его очистки. Следующий код вызывает метод *Write* после закрытия файла и пытается дописать в файл несколько байт. Ясно, что это невозможно, и при исполнении кода второй вызов метода *Write* сгенерирует исключение *System.ObjectDisposedException* с сообщением: «Cannot access a closed file» («Нет доступа к закрытому файлу»).

```
using System;

using System.IO;

public static class Program {
   public static void Main() {
      // Создать байты для записи во временный файл.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Создать временный файл.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // Записать байты во временный файл.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Явно закрыть файл после записи.
      fs.Close();

      // Попытка записать в файл после его закрытия.
      // Следующая строка генерирует исключение ObjectDisposedException.
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // Удалить временный файл.
      File.Delete("Temp.dat");
   }
}
```

Здесь 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 модель освобождения ресурсов, обязательно сделайте так, чтобы все методы и свойства генерировали исключение *System.ObjectDisposedException*, если объект был явно очищен. При повторных вызовах методы *Dispose* и *Close* никогда не должны генерировать исключение *ObjectDisposedException* — они должны просто вернуть управление.

Внимание! Настоятельно рекомендую в общем случае отказаться от методов *Dispose* или *Close*. Сборщик мусора из CLR достаточно хорошо написан, и пусть он делает свою работу сам. Он знает, когда объект больше недостижим из кода приложения, и только тогда уничтожает объект. Вызывая *Dispose* или *Close*, код приложения в сущности заявляет, что знает, когда объект становится не нужен приложению. Но во многих приложениях нельзя с полной уверенностью сказать, что данный объект больше не нужен.

Допустим, у вас есть код, создающий 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 *Dispose* или *Close*, но, если какой-то код попытается обратиться к этому объекту, будет сгенерировано исключение *ObjectDisposedException*.

Рекомендую вызывать *Dispose* или *Close* 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 завершения и тем самым не позволив ему перейти в следующее поколение.

Оператор *using* языка C#

Приведенные примеры кода демонстрируют методику явного вызова методов *Dispose* и *Close*. Если вы решили вызывать один из них явно, настоятельно рекомендую поместить вызовы в блок обработки исключений *finally*, что гарантирует исполнение кода очистки. Так что лучше написать код из предыдущего примера так.

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Создать байты для записи во временный файл.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };
```

```
      // Создать временный файл.
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);
      try {
         // Записать байты во временный файл.
         fs.Write(bytesToWrite, 0, bytesToWrite.Length);
      }
      finally {
         // Явно закрыть файл после записи.
         if (fs != null)
            fs.Dispose();
      }

      // Удалить временный файл.
      File.Delete("Temp.dat");
   }
}
```

Здесь верным решением было бы добавить код для обработки исключений; не поленитесь это сделать. К счастью, в C# есть оператор *using*, предлагающий упрощенный синтаксис, позволяющий сгенерировать код, идентичный только что показанному. Вот как можно переписать предыдущий код с использованием оператора *using*.

```
using System;
using System.IO;

public static class Program {
   public static void Main() {
      // Создать байты для записи во временный файл.
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // Создать временный файл.
      using (FileStream fs = new FileStream("Temp.dat", FileMode.Create)) {
         // Записать байты во временный файл.
         fs.Write(bytesToWrite, 0, bytesToWrite.Length);
      }

      // Удалить временный файл.
      File.Delete("Temp.dat");
   }
}
```

Оператор *using* инициализирует объект и сохраняет ссылку на него в переменной. После этого к этой переменной можно обращаться из кода, расположенного в фигурных скобках в операторе *using*. При компиляции этого кода компилятор автоматически генерирует блоки *try* и *finally*. Внутри блока *finally* компилятор помещает код, выполняющий приведение объекта к *IDisposable*, и вызывает метод *Dispose*. Ясно, что компилятор позволяет использовать оператор *using* только с типами, в которых реализован интерфейс *IDisposable*.

Примечание Оператор языка C# *using* позволяет инициализировать несколько переменных одного типа или использовать переменную, инициализированную ранее.

Оператор *using* также работает со значимыми типами, реализующими интерфейс *IDisposable*. Это позволяет создать чрезвычайно эффективный и полезный механизм для инкапсуляции кода, необходимого для начала и завершения операции. Скажем, вам нужно заблокировать блок кода с помощью мьютекса. В классе *Mutex* не реализован интерфейс *IDisposable*, но вызов *Dispose* для этого объекта освобождает машинный ресурс. Это не имеет никакого отношения к блокировке. Для эффективной блокировки и разблокировки мьютекса можно определить значимый тип, который инкапсулирует блокировку и разблокировку объекта *Mutex*. Для примера рассмотрим структуру *MutexLock*, а следующий за ней метод *Main* демонстрирует эффективное использование *MutexLock*.

```
using System;
using System.Threading;

// Этот значимый тип инкапсулирует блокировку и разблокировку мьютекса.
// Учтите, что эта неоткрытая структура,созданная в помощь программе;
// этот тип не является частью библиотеки.
internal struct MutexLock : IDisposable {
   private Mutex m_mutex;

   // Этот конструктор получает блокировку мьютекса.
   public MutexLock(Mutex m) {
      m_mutex = m;
      m_mutex.WaitOne();
   }

   // Этот метод Dispose снимает блокировку мьютекса.
   public void Dispose() {
      m_mutex.ReleaseMutex();
   }
}

public static class Program {
   // Этот метод показывает, как эффективно использовать MutexLock.
   public static void Main() {
      // Создаем объект-мьютекс.
      Mutex m = new Mutex();

      // Блокируем мьютекс, выполняем задачу и разблокируем мьютекс.
      using (new MutexLock(m)) {
         // Выполняем операции в безопасном режиме.
      }
   }
}
```

Внимание! Наличие значимого типа, в котором реализован интерфейс *IDisposable*, как в примере с *MutexLock*, очень эффективно. Однако такие значимые типы должны определяться и использоваться только в закрытом коде. Не следует определять открытые значимые типы, реализующие интерфейс *IDisposable*, в повторно используемом коде библиотеки классов. Это объясняется тем, что значимые типы копируются при упаковке и при передаче в качестве аргументов методам. Это значит, что есть несколько экземпляров, ссылающихся на один ресурс. Пользователь экземпляра может завершить операцию, но другой код может использовать другой экземпляр, который, в свою очередь, может дважды использоваться до завершения операции, что отрицательно скажется на стабильности приложения. Если в коде библиотеки нужно представить вспомогательный класс (например, тип *MutexLock*), следует определить тип как ссылочный (то есть класс) и реализовать дополнительную логику, чтобы обеспечить однократное исполнение операции завершения, даже если *Dispose* вызывается несколько раз.

Интересная проблема с зависимостью

Тип *System.IO.FileStream* позволяет пользователю открывать файл для чтения и записи. Для повышения быстродействия реализация типа использует буфер памяти. Тип сбрасывает содержимое буфера в файл только после его заполнения. Тип *FileStream* поддерживает только запись байт — для записи символов или строк используйте 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 Close обязателен.
sw.Close();
// ПРИМЕЧАНИЕ: метод StreamWriter.Close закрывает объект FileStream.
// Явно закрывать объект FileStream не нужно.
```

Заметьте: конструктор *StreamWriter* принимает в качестве параметра ссылку на объект *Stream*, тем самым позволяя передавать ссылку на объект *FileStream* в качестве параметра. Внутренний код объекта *StreamWriter* сохраняет ссылку на объект *Stream*. При записи в объект *StreamWriter* он выполняет внутреннюю буферизацию данных в свой буфер в памяти. При заполнении буфера *StreamWriter* записывает данные в *Stream*.

После записи данных через объект *BinaryWriter* следует вызывать метод *Dispose* или *Close* (так как в типе *StreamWriter* реализована модель освобождения ресурсов, его можно использовать с оператором *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 Явно вызывать методы *Dispose* или *Close* для объекта *FileStream* необязательно: *BinaryWriter* сделает это сам. Если же один из этих методов все-таки был вызван явно, *FileStream* увидит, что очистка объекта уже выполнена, и вызванный метод просто вернет управление.

Как вы думаете, что было бы, не будь кода, явно вызывающего *Dispose* или *Close*? Сборщик мусора однажды правильно определил бы, что эти объекты стали мусором, и завершил их. Но он не гарантирует вызов методов *Finalize* в определенном порядке. Поэтому если объект *FileStream* завершится первым, он закроет файл. Затем после завершения объекта *StreamWriter* он попытается записать данные в закрытый файл, что вызовет исключение. С другой стороны, если *StreamWriter* завершается первым, данные будут благополучно записаны в файл.

Как с этой проблемой справились в Microsoft? Заставить сборщик мусора завершить объекты гарантированно в «правильном» порядке нельзя, так как в объектах могут быть ссылки друг на друга, и тогда сборщик не сможет определить правильный порядок их завершения. В Microsoft нашли выход: в типе *StreamWriter* не реализован метод *Finalize*, поэтому он не может сбросить данные из своего буфера в базовый объект *FileStream*. Это значит, что, если вы забыли явно закрыть объект *StreamWriter*, данные гарантированно будут потеряны. В Microsoft ожидают, что разработчики заметят эту повторяющуюся потерю данных и исправят код, вставив явный вызов *Close* или *Dispose*.

Примечание В .NET 2.0 поддерживаются управляемые отладчики — Managed Debugging Assistants (MDA). Когда они включены, .NET Framework выполняет поиск некоторых распространенных ошибок в программах и запускает соответствующий MDA. В отладчике все это выглядит как генерация исключения. Соответствующий MDA обнаружит, когда объект *StreamWriter* был удален сборщиком мусора, прежде чем он был явно закрыт. Чтобы включить данный MDA в Visual Studio, откройте проект и выберите в меню Debug/Exceptions. В диалоговом окне **Exceptions** раскройте узел **Managed Debugging Assistants**, прокрутите страницу вниз до **StreamWriterBufferedDataLost** и отметьте этот элемент флажком **Thrown**, чтобы включить остановку отладчика Visual Studio при каждой потере данных объекта *StreamWriter*.

Ручной мониторинг и управление временем жизни объектов

Для каждого домена приложения CLR поддерживает *таблицу описателей GC* (GC handle table), с помощью которой приложение отслеживает время жизни объекта или позволяет управлять им вручную. При создании домена приложения таблица пустая. Каждый элемент таблицы состоит из указателя на объект в управляемой куче и флага, указывающего способ мониторинга или управления объектом. Приложение добавляет и удаляет элементы из таблицы с помощью типа *System.Runtime.InteropServices.GCHandle*, показанного далее. Поскольку таблица описателей GC чаще всего используется в сценариях взаимодействия с неуправляемым кодом, большин-

ство членов *GCHandle* должны иметь ссылку на *SecurityPermission* с флагом *UnmanagedCode*.

```
// Этот тип определен в пространстве имен System.Runtime.InteropServices.
public struct GCHandle {
   // Статические методы, создающие элементы в таблице.
   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 GCHandle.
   public static Boolean operator ==(GCHandle a, GCHandle b);
   public static Boolean operator !=(GCHandle a, GCHandle b);

   // Метод экземпляра, освобождающий элемент в таблице
   // (индекс приравнивается к 0).
   public void Free();

   // Свойство экземпляра, извлекающее/назначающее ссылку на объект элемента.
   public object Target { get; set; }

   // Свойство экземпляра, возвращающее true, если индекс не равен 0.
   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 управления или мониторинга времени жизни объекта вызывается статический метод *Alloc* объекта *GCHandle*, передающий ссылку на этот объект, и тип *GCHandleType* — флаг, указывающий способ мониторинга/управления объектом. Перечислимый тип *GCHandleType* определяется так.

```
public enum GCHandleType {
   Weak = 0, // Используется для мониторинга объекта.
   WeakTrackResurrection = 1 // Используется для мониторинга объекта.
   Normal = 2, // Используется для управления временем жизни объекта.
   Pinned = 3  // Используется для управления временем жизни объекта.
}
```

Вот, что означают эти флаги.

- ***Weak*** позволяет *выполнять мониторинг* времени жизни объекта, а именно: можно узнать, когда сборщик мусора обнаружит, что объект недостижим из кода приложения. Учтите, что метод *Finalize* объекта мог как выполниться, так и не выполниться, поэтому объект может по-прежнему оставаться в памяти.
- ***WeakTrackResurrection*** позволяет *выполнять мониторинг* времени жизни объекта, а именно: можно узнать, когда сборщик мусора обнаружит, что объект недостижим из кода приложения. Учтите, что метод *Finalize* объекта (если таковой имеется) уже точно выполнился и память, занятая объектом, была освобождена.
- ***Normal*** позволяет *управлять* временем жизни объекта, а именно заставляет сборщик мусора оставить объект в памяти, даже если в приложении нет переменных (корней), ссылающихся на него. В ходе сбора мусора память, занятая этим объектом, может быть сжата (перемещена). Метод *Alloc*, не принимающий флаг *GCHandleType*, предполагает, что *GCHandleTypeNormal* определен.
- ***Pinned*** позволяет *управлять* временем жизни объекта, а именно заставляет сборщик мусора оставить объект в памяти, даже если в приложении нет переменных (корней), ссылающихся не него. В ходе сбора мусора память, занятая этим объектом, не может быть сжата (перемещена). Это обычно бывает полезно, когда нужно передать адрес в памяти в неуправляемый код. Неуправляемый код может выполнять запись по этому адресу в управляемой куче, зная, что расположение управляемого объекта не изменится после сбора мусора.

При вызове статический метод *Alloc* объекта *GCHandle* сканирует таблицу описателей GC домена приложения в поисках элемента, в котором хранится адрес объекта, переданного *Alloc*. При этом устанавливается флаг, переданный в качестве параметра *GCHandleType*. Затем *Alloc* возвращает экземпляр *GCHandle*. *GCHandle* — это «облегченный» значимый тип, содержащий одно поле экземпляра, *IntPtr*, ссылающееся на индекс элемента в таблице. При необходимости освободить этот элемент в таблице описателей GC 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 описателей GC.

1. Сборщик мусора маркирует все достижимые объекты (как описано в начале этой главы). Затем он сканирует таблицу описателей GC, все объекты с флагами *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 описателей GC в поисках всех записей с флагом *Weak*. Если такая запись ссылается на немаркированный объект, указатель относится к недостижимому объекту (мусору) и указатель этого элемента приравнивается к *null*.
3. Сборщик мусора сканирует список завершения. Если указатель из списка ссылается на немаркированный объект, он относится к недостижимому объекту и перемещается из списка завершения в очередь freachable. В этот момент объект маркируется, потому что сейчас он считается достижимым.
4. Сборщик мусора сканирует таблицу описателей GC в поисках всех элементов с флагом *WeakTrackResurrection*. Если такой элемент ссылается на немаркированный объект (теперь это объект, на который указывает элемент из очереди

freachable), указатель считается относящимся к недостижимому объекту (мусором), а указатель этого элемента приравнивается к *null*.

5. Сборщик мусора сжимает память, убирая свободные места, оставшиеся на месте недостижимых объектов. Учтите, что сборщик иногда предпочитает не сжимать память, если посчитает, что фрагментация низкая 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, узнаем, как использовать это знание на практике. Наиболее понятны флаги *Normal* и *Pinned*, начнем с них. Обычно они используются при взаимодействии с неуправляемым кодом.

Флаг *Normal* применяется, когда нужно передать ссылку на управляемый объект неуправляемому коду, потому 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 сборе 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 *IntPtr*, а *IntPtr* передать в неуправляемый код. Когда неуправляемый код выполнит обратный вызов управляемого кода, последний приведет передаваемый *IntPtr* обратно к *GCHandle*, после чего запросит свойство *Target*, чтобы получить ссылку (или текущий адрес) управляемого объекта. Когда неуправляемому коду больше не нужна эта ссылка, нужно вызывать метод *Free* объекта *GCHandle*, который позволит очистить объект при следующем сборе 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 ссылаться на него. Но есть ситуации, когда неуправляемому коду нужно использовать управляемый объект. Тогда управляемый объект надо отметить флагом *Pinned* — это запретит сборщику мусора перемещать и сжимать объект. Распространенный пример — передача 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, когда существует возможность перемещения этого объекта сборщиком мусора. Если бы объект *String* был перемещен, неуправляемый код выполнял бы чтение или запись в память, более не содержащую символы объекта *String*, — это сделало бы работу приложения непредсказуемой.

При использовании механизма CLR P/Invoke для вызова метода CLR автоматически устанавливает флаг *Pinned* параметров и снимает его, когда неуправляемый метод возвращает управление. Поэтому чаще всего в типе *GCHandle* не приходится самостоятельно явно устанавливать флаг *Pinned* 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 выполнении асинхронных операций 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 объект-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 вернет 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 при следующем сборе мусора перемещать буфер. В управляемом коде должна по-прежнему быть ссылка на этот буфер, чтобы у разработчика был доступ к данным. Эта ссылка не позволит сборщику полностью убрать буфер из памяти.

Поговорим о двух других флагах: *Weak* и *WeakTrackResurrection*. Они могут использоваться как в сценариях взаимодействия с неуправляемым кодом, так и в случаях применения только управляемого кода. Флаг *Weak* встречается в ситуациях, когда один объект не должен препятствовать сборщику мусора удалять другой объект. Например, нужно создать объект *Fax*, который будет оповещаться всякий раз, когда объект *MailManager* получает новое сообщение по электронной почте. Если же объект *Fax* становится не нужен, сборщик мусора должен его удалить. Естественно, если *Fax* удалить, объект *MailManager* не будет уведомлять его о получении новых писем.

Работа сборщика мусора в этом сценарии достаточно запутанна. Дело в том, что объект, отправляющий уведомления, должен иметь ссылку на объект, их получающий. Именно из-за этой ссылки объект-получатель не уничтожается сборщиком. Здесь на помощь придут флаги *Weak* и *WeakTrackResurrection*.

Вот что нужно сделать. При конструировании объекта *Fax* он регистрируется у объекта *MailManager* и передает ему делегат, который указывает на метод обратного вызова, определенный в классе *Fax*. Объект *MailManager* вызывает метод *Alloc* объекта *GCHandle* и передает ему ссылку на объект-делегат и флаг *Weak*. Объект *MailManager* сохраняет возвращенный экземпляр *GCHandle* в наборе вместо того, чтобы сохранить ссылку на объект-делегат. Поскольку у объекта *MailManager* нет ссылки на объект-делегат, сборщик мусора может удалить этот объект-делегат и объект *Fax*, на который он ссылается, в случае отсутствия других ссылок на объект *Fax*.

Когда объекту *MailManager* нужно уведомить *Fax* о поступлении новой почты, он берет сохраненный экземпляр *GCHandle* и использует его для запроса свойства *Target*. Если *Target* возвратит *null*, значит объект *Fax* был удален сборщиком, а *MailManager* теперь должен воспользоваться сохраненным экземпляром *GCHandle*, чтобы вызвать *Free* и удалить его из своего внутреннего списка приемников. Если свойство *Target* возвращает другое значение, значит объект *Fax* не удален, а значение свойства *Target* является ссылкой на делегат, выполняющий роль оболочки экземпляра метода объекта *Fax*; из-за этой ссылки объект *Fax* не удаляется сборщиком мусора. Объект *MailManager* теперь может воспользоваться этой переменной, чтобы уведомить объект *Fax* о поступлении новой почты. Когда эта переменная выйдет из области видимости, объект *Fax* вновь станет кандидатом на уничтожение сборщиком мусора.

Как упоминалось ранее, единственное отличие между флагами *Weak* и *WeakTrackResurrection* в том, что CLR указывает, что объект был удален сборщиком. На самом деле мне никогда не встречался разработчик, использующий флаг *WeakTrackResurrection*.

Примечание Было бы хорошо, если бы .NET Framework поддерживала механизм делегирования с помощью *мягких ссылок* (weak reference delegate), но пока этого нет. Все же этот вопрос обсуждается в команде разработчиков CLR в Microsoft, и, вполне возможно, такая возможность появится в следующей версии.

Поскольку работать с типом *GCHanlde* не очень удобно, в помощь разработчику в пространстве имен *System* предлагается класс *WeakReference*. Этот класс является объектно-ориентированной оболочкой экземпляра *GCHandle*: логически его конструктор вызывает метод *Alloc* объекта *GCHandle*, его свойство *Target* вызывает свойство *Target* объекта *GCHandle*, а его метод *Finalize* вызывает метод *Free* объекта *GCHandle*. Кроме того, коду не нужны специальные разрешения для использования класса *WeakReference*, потому что этот класс поддерживает только мягкие ссылки. Он не поддерживает работу экземпляров *GCHandle*, размещенных в памяти с помощью типа *GCHandleType* с флагом *Normal* или *Pinned*.

Недостаток класса *WeakReferences* в том, что он класс и, как следствие, является более «тяжеловесным» объектом, чем экземпляр *GCHandle*. Также, в классе *WeakReference* не реализована модель освобождения ресурсов (и это недочет), поэтому явно освободить элемент в таблице *GCHandle* никак нельзя — нужно ждать, пока начнет работу сборщик мусора и будет вызван его метод *Finalize*.

Следующий код демонстрирует применение класса *WeakReference* для реализации вышеописанного сценария.

```
using System;
using System.Collections.Generic;

public static class Program {
   public static void Main() {
      // Создание объекта MailManager.
      MailManager mm = new MailManager();

      // Создание объекта Fax.
      Fax f = new Fax(mm);

      // Поскольку сборщик мусора ни разу не запускался, объект Fax
      // получит уведомление о событии.
      mm.SimulateNewMail();

      // Принудительный вызов сборщика мусора.
      GC.Collect();

      // Мусор удален, поэтому объект Fax не получит уведомление о событии.
      mm.SimulateNewMail();
   }
}

internal sealed class MailManager {
   // Набор содержит ряд мягких ссылок на делегаты.
   private List<WeakReference> m_NewMailCallbacks = new List<WeakReference>();
```

```
// Это событие вызывается, когда поступает новая почта.
public event EventHandler NewMail {
   add {
      lock (m_NewMailCallbacks) {
         // Сконструировать мягкую ссылку (WeakReference)
         // для передаваемого делегата и добавить эту ссылку в набор.
         m_NewMailCallbacks.Add(new WeakReference(value));
      }
   }

   remove {
      lock (m_NewMailCallbacks) {

         // Просканировать набор в поисках соответствующего делегата
         // и удалить его.
         for (Int32 n = 0; n < m_NewMailCallbacks.Count; n++) {

            // Убрать мягкую ссылку (WeakReference) из набора.
            WeakReference wr = m_NewMailCallbacks[n];

            // Попытаться превратить мягкую ссылку в жесткую.
            EventHandler eh = (EventHandler) wr.Target;

            if (eh == null) {
               // Объект был удален сборщиком мусора.
               // Удалить этот элемент из набора.
               m_NewMailCallbacks.RemoveAt(n);
               n--;       // Вернуться на один элемент назад.
               continue; // Проверить следующий элемент.
            }

            // Объект не был удален сборщиком.
            // Соответствует ли делегат набора переданному делегату?
            if ((eh.Target == value.Target) && (eh.Method == value.Method)) {
               // Да, они совпадают. Удалить его и вернуть управление.
               m_NewMailCallbacks.RemoveAt(n);
               break;
            }
         }
      }
   }
}

// Вызов этого метода, чтобы смоделировать новое поступление почты.
public void SimulateNewMail() {
   Console.WriteLine ("About to raise the NewMail event");
   OnNewMail(EventArgs.Empty);
}

// Этот метод вызывает событие NewMail.
private void OnNewMail(EventArgs e) {
   lock (m_NewMailCallbacks) {
```

```
            // Сканирование набора с обратным вызовом каждого делегата.
            for (Int32 n = 0; n < m_NewMailCallbacks.Count; n++) {

               // Извлечение WeakReference из набора.
               WeakReference wr = m_NewMailCallbacks[n];

               // Попытка превратить мягкую ссылку в жесткую.
               EventHandler eh = (EventHandler)wr.Target;

               if (eh == null) {
                  // Объект был удален сборщиком мусора.
                  // Удалить этот элемент из набора.
                  m_NewMailCallbacks.RemoveAt(n);
                  n--;0      // Вернуться на один элемент назад.
               } else {
                  // Объект не был удален сборщиком мусора.
                  // Вызов делегата.
                  eh(this, e);
               }
            }
         }
      }
}

internal sealed class Fax {
   // При конструировании регистрируемся для получения
   // уведомлений о событии NewMail объекта MailManager.
   public Fax(MailManager mm) {
      mm.NewMail += GotMail;
   }

   // Метод вызывается при появлении NewMail.
   public void GotMail(Object sender, EventArgs e) {
      // Просто, чтобы подтвердить, что мы дошли до этой точки.
      Console.WriteLine("In Fax.GotMail");
   }
}
```

После компиляции и запуска этой программы получим следующее.

```
About to raise the NewMail event
In Fax.GotMail
About to raise the NewMail event
```

Внимание! Познакомившись с мягкими ссылками, разработчики сразу же считают, что они хорошо подходят для сценариев кеширования. 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, если

он рядом, она воспользуется нужными данными. Это обеспечит высокую производительность приложения». Однако после сбора мусора объекты, содержащие данные, будут уничтожены, и когда программе придется заново создавать данные, ее производительность упадет.

Недостаток такого подхода в том, что сбор мусора не происходит, когда память переполнена или почти переполнена. Напротив, он происходит, когда поколение 0 заполнено, что бывает примерно после выделения очередных 256 байт памяти. Поэтому объекты удаляются из памяти гораздо чаще, чем нужно, что сильно ухудшает производительность приложения.

Мягкие ссылки могут быть очень эффективными при кешировании, но очень сложно создать хороший алгоритм кеширования с правильным равновесием между расходованием памяти и скоростью. В сущности, необходимо, чтобы в кеше 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. Но некоторые разработчики приспособились периодически вызывать Win32-функцию *GlobalMemoryStatusEx* и проверять член *dwMemoryLoad* возвращенной структуры *MEMORYSTATUSEX*. Если его значение больше 80, память на исходе и настало время преобразовывать жесткие ссылки в мягкие по выбранному алгоритму — по давности, частоте, времени использования объектов или по другим алгоритмам.

Воскрешение

Вспомните: рассматривая завершение, мы говорили о том, что, когда требующий завершения объект считается мертвым, сборщик мусора возвращает его к жизни, чтобы вызвать его метод *Finalize*, а после вызова метода *Finalize* объект умирает навсегда. Подытоживая, можно сказать: объект, требующий завершения, сначала умирает, затем воскресает и умирает снова. Оживление мертвого объекта называется *воскрешением* (resurrection).

Акт подготовки к вызову метода *Finalize* объекта — одна из форм воскрешения. Когда сборщик мусора помещает ссылку на объект в очередь freachable, объект становится достижимым от корня и возвращается к жизни. Это нужно, чтобы код в методе *Finalize* мог обратиться к полям объекта. В конце концов метод *Finalize* этого объекта возвращает управление, после чего на объект больше не указывает ни один корень, потому что он удален из очереди freachable, после этого объект уничтожается навсегда.

Но что, если метод *Finalize* объекта исполняет код, который заносит указатель на объект в статическое поле, как показано в следующем коде?

```
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 *Finalize* объекта *SomeType* ссылка на него помещается в корень и объект становится достижимым из кода приложения. Теперь объект воскресает, а сборщик мусора не примет его за мусор. Приложение вольно задействовать этот объект, но помните, что он уже *завершен* и его использование может дать непредсказуемые результаты. Также учтите, что если в *SomeType* есть поля, ссылающиеся на другие объекты, они тоже будут воскрешены, так как они достижимы от корней приложения. Однако знайте, что у некоторых из этих объектов метод *Finalize* может быть уже вызван.

Как и в реальной жизни (или смерти), в воскрешении нет ничего хорошего, и его лучше избегать при написании программ. Воскрешение может быть полезно в тех немногих ситуациях, когда архитектура приложения требует многократного использования одного объекта. Когда объект больше не нужен, выполняется сбор мусора. В своем методе *Finalize* объект присваивает своему указателю *this* другой корень, что запрещает сборщику удалять объект. Но надо сообщить сборщику мусора о необходимости вызова метода *Finalize* после следующего использования объекта. Чтобы реализовать эту возможность, тип *GC* поддерживает статический метод *ReRegisterForFinalize*, который принимает один параметр — ссылку на объект. Следующий код показывает, как изменить метод *Finalize* объекта *SomeType*, чтобы он вызывался после каждого использования объекта.

```
internal sealed class SomeType {
   ~SomeType() {
      Program.s_ObjHolder = this;
      GC.ReRegisterForFinalize(this);
   }
}
```

Вызов метода *Finalize* воскрешает объект, заставляя корень вновь ссылаться на него. Далее этот метод вызывает *ReRegisterForFinalize*, который добавляет адрес заданного объекта (*this*) в конец списка завершения. Обнаружив, что этот объект недостижим (когда статическое поле приравнивается к *null*), сборщик мусора переместит указатель объекта из списка завершения в очередь freachable, и метод *Finalize* 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*, и очень часто это 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 *Finalize* 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 *Finalize* для объекта будет вызван несколько раз, так как при каждом вызове *ReRegisterForFinalize* к списку завершения добавляется новый элемент. Когда объект определяется как мусор, все эти элементы перемещаются из списка завершения в очередь freachable, что приводит к многократному вызову метода *Finalize*.

Поколения

Как говорилось в начале этой главы, поколения — это механизм сборщика мусора в CLR, единственное назначение которого — повышение производительности приложения. *Сборщик мусора с поддержкой поколений* (generational garbage collector) [его также называют *эфемерным сборщиком мусора* (ephemeral garbage collector), хотя я не использую такой термин в этой книге] работает на основе следующих предположений:

- чем младше объект, тем короче его время жизни;
- чем старше объект, тем длиннее его время жизни;
- сбор мусора в части кучи выполняется 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, объекты поколения 0 — это только что созданные объекты, которых не касался сборщик мусора. На рис. 20-8 показано только что запущенное приложение, разместившее в памяти пять объектов (A–E). Через некоторое время объекты C и E становятся недостижимыми.

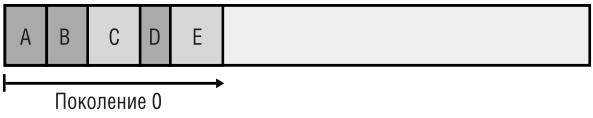


Рис. 20-8. *Вид кучи сразу после инициализации: все объекты в ней относятся к поколению 0, сбора мусора еще не было*

При инициализации CLR выбирает пороговый размер для поколения 0, например 256 Кб (конкретный размер может изменяться). Если в результате выделения памяти для нового объекта размер поколения 0 превысил пороговое значение, должен начаться сбор мусора. Допустим, объекты A–E занимают 256 Кб. При размещении объекта F должен начаться сбор мусора. Сборщик мусора выяснит, что объекты C и E — это мусор, и выполнит сжатие объекта D, переместив его вплотную к объекту B. Между прочим, пороговый размер для поколения 0 в 256 Кб был выбран из-за того, что обычно все эти объекты целиком умещаются в кеш второго уровня (L2) процессора, что значительно повысит скорость дефрагментации

памяти. Объекты, пережившие сбор мусора (A, B и D), становятся поколением 1. Объекты из поколения 1 были просмотрены сборщиком мусора один раз. Теперь куча выглядит так (рис. 20-9).

Рис. 20-9. *Вид кучи после одного сбора мусора: выжившие объекты из поколения 0 переходят в поколение 1, поколение 0 пусто*

После сбора мусора в поколении 0 не остается объектов. Как всегда, новые объекты размещаются в поколении 0. На рис. 20-10 показано, что приложение продолжает работу и размещает объекты F–K. Также во время работы приложения объекты B, H и J стали недостижимыми, и занятая ими память должна рано или поздно освободиться.

Рис. 20-10. *В поколении 0 созданы новые объекты, в поколении 1 появился мусор*

А теперь представьте, что при попытке разместить объект L размер поколения 0 превысил пороговое значение 256 Кб, и поэтому должен начаться сбор мусора. При этом сборщик мусора должен решить, какие поколения следует обработать. Я уже говорил, что при инициализации CLR выбирает пороговый размер поколения 0. CLR также выбирает пороговый размер для поколения 1, скажем 2 Мб.

Начиная сбор мусора, сборщик определяет, сколько памяти занято поколением 1. Пока поколение 1 занимает намного меньше 2 Мб, поэтому сборщик проверяет только объекты поколения 0. Первое допущение сборщика гласит, что у новых объектов время жизни будет короче. Поэтому в поколении 0, скорее всего, окажется много мусора, и очистка этого поколения освободит много памяти. Сборщик проигнорирует объекты поколения 1, что значительно ускорит сбор мусора.

Ясно, что игнорирование объектов поколения 1 повышает скорость работы сборщика. Однако его производительность улучшается еще больше за счет выборочной проверки объектов в управляемой куче. Если корень или объект ссылается на объект из старшего поколения, сборщик игнорирует все внутренние ссылки старшего объекта, уменьшая время построения графа достижимых 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 поля ссылок объекта. Он позволяет сборщику выяснить, какие из старых объектов (если они есть) были изменены с момента последнего сбора мусора. Следует изучать только старые объекты с измененными полями, чтобы выяснить, не ссылаются ли они на новые объекты из поколения 0.

Примечание Тесты быстродействия, проведенные Microsoft, показали, что сбор мусора в поколении 0 занимает меньше 1 мс. Microsoft стремится к тому, чтобы сбор мусора занимал не больше времени, чем обслуживание обычной страничной ошибки.

Сборщик мусора с поддержкой поколений также предполагает, что объекты, прожившие достаточно долго, будут жить и дальше. Так что велика вероятность, что объекты поколения 1 останутся достижимыми из приложения и впредь. Поэтому, проверив объекты поколения 1, сборщик нашел бы мало мусора и не смог бы освободить много памяти. Следовательно, сбор мусора в поколении 1 скорее всего окажется пустой тратой времени. Если в поколении 1 появляется мусор, он просто остается там. Сейчас куча выглядит так (рис. 20-11).

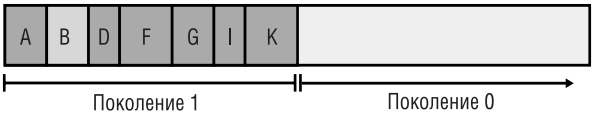


Рис. 20-11. *Вид кучи после двух сборов мусора: выжившие объекты из поколения 0 переходят в поколение 1 (увеличивая его размер), поколение 0 пусто*

Как видите, все объекты из поколения 0, пережившие сбор мусора, перешли в поколение 1. Так как сборщик не проверяет поколение 1, память, занятая объектом B, не освобождается, даже если этот объект недостижим на момент сбора мусора. И в этот раз после сбора мусора поколение 0 опустело, в это поколение попадут новые объекты. Допустим, приложение работает дальше и выделяет объекты L–O. Во время работы приложение прекращает использовать объекты G, L и M, и они становятся недостижимыми. Теперь куча примет такой вид (рис. 20-12).

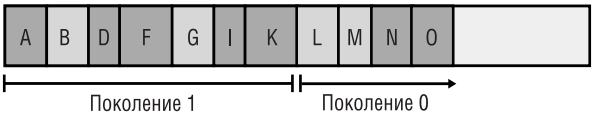


Рис. 20-12. *В поколении 0 созданы новые объекты, количество мусора в поколении 1 увеличилось*

Допустим, в результате размещения объекта P размер поколения 0 превысил пороговое значение, что вызвало запуск сбора мусора. Поскольку все объекты поколения 1 занимают в совокупности меньше 2 Мб памяти, сборщик вновь решает собрать мусор только в поколении 0, игнорируя недостижимые объекты в поколении 1 (B и G). После сбора куча выглядит так (рис. 20-13).

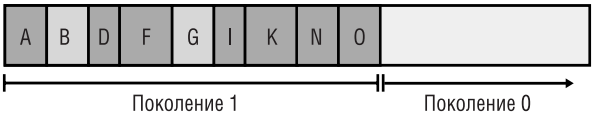


Рис. 20-13. *Вид кучи после трех сборов мусора: выжившие объекты из поколения 0 переходят в поколение 1 (увеличивая его размер); поколение 0 пусто*

На рис. 20-13 видно, что поколение 1 медленно, но верно растет. Допустим, поколение 1 выросло до таких размеров, что все его объекты в совокупности занимают 2 Мб памяти. В этот момент приложение продолжает работу (потому что сбор мусора только что завершился) и начинает размещение объектов P–S, которые заполнят поколение 0 до его порогового значения (рис. 20-14).

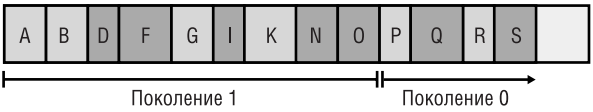


Рис. 20-14. *Новые объекты размещены в поколении 0, в поколении 1 появилось больше мусора*

Когда приложение пытается разместить объект T, поколение 0 заполняется и запускается сбор мусора. Однако на этот раз сборщик мусора обнаруживает, что место, занятое объектами, превысило пороговое значение. После нескольких сборов мусора в поколении 0 велика вероятность, что несколько объектов в поколении 1 стали недостижимыми (как в нашем примере). Поэтому теперь сборщик мусора проверит все объекты поколений 1 и 0. После сбора мусора в обоих поколениях куча выглядит, как показано на рис. 20-15.

Рис. 20-15. *Вид кучи после четырех сборов мусора: выжившие объекты из поколения 1 переходят в поколение 2, выжившие объекты из поколения 0 переходят в поколение 1, а поколение снова 0 пусто*

Все выжившие объекты поколения 0 теперь находятся в поколении 1, а все выжившие объекты поколения 1 — в поколении 2. Как всегда, сразу после сбора мусора поколение 0 пусто: в нем будут размещаться новые объекты. В поколении 2 находятся объекты, проверенные сборщиком мусора не меньше двух раз. Сборов мусора может быть много, но объекты поколения 1 проверяются, только когда их суммарный размер достигает порогового значения — до этого обычно проходит несколько сборов мусора в поколении 0.

Управляемая куча поддерживает только три поколения: 0, 1 и 2. Поколения 3 не бывает. При инициализации в CLR устанавливается пороговое значение для всех трех поколений. Как говорилось раньше, для поколения 0 оно равно примерно 256 Кб, для поколения 1 — около 2 Мб. Пороговое значение для поколения 2 равно около 10 Мб. Пороговые значения выбираются так, чтобы повысить производительность. Чем больше пороговое значение, тем реже выполняется сбор мусора. Опять же, повышение производительности достигается благодаря исходным предположениям: 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 сбора мусора он анализирует работу приложения и адаптируется. Например, если приложение создает множество объектов и пользуется ими очень недолго, сбор

мусора в поколении 0 может освободить много памяти. На самом деле можно освободить память всех объектов в поколении 0.

Если сборщик видит, что после сбора мусора в поколении 0 остается очень мало выживших объектов, он может снизить порог для поколения 0 с 256 до 128 Кб. В этом случае сбор мусора будет выполняться чаще, но это потребует меньше работы сборщика, поэтому рабочий набор процесса будет оставаться небольшим. В сущности, если все объекты поколения 0 станут мусором, сборщику не придется даже дефрагментировать память — достаточно будет вернуть *NextObjPtr* в начало поколения 0, и сбор мусора можно считать законченным. Замечательный способ освобождения памяти!

Примечание Сборщик мусора отлично работает с приложениями, потоки которых большую часть времени бездействуют, находясь в верху стека. Когда у потока появляется задача, 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 и Web-сервисах 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 недостижимыми, каждый сбор мусора освобождает много памяти. Это позволяет поддерживать очень небольшой рабочий набор и достичь исключительного быстродействия сборщика.

В сущности, работа параметров и локальных переменных большинства корней приложения зависит от стека потока. Если стек потока небольшой, сборщик мусора быстро проверяет корни и маркирует достижимые объекты. Иначе говоря, чтобы добиться быстрого сбора мусора, нужно избегать глубоких стеков, например, отказавшись от использования рекурсивных методов.

С другой стороны, если после обработки поколения 0 сборщик мусора видит множество выживших объектов, это значит, что удалось освободить мало памяти. В этом случае сборщик мусора может поднять порог для поколения 0, например до 512 Кб. Теперь сбор мусора будет выполняться реже, но каждый раз будет освобождаться значительный объем памяти. Кстати, если сборщик освободил недостаточно памяти, он выполнит полный сбор мусора, прежде чем сгенерировать исключение *OutOfMemoryException*.

Я привел пример, как сборщик динамически может изменять порог поколения 0, но сходным образом могут меняться пороги для поколений 1 и 2. При сборе 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 этих поколений, чтобы повысить производительность приложения. В итоге сборщик мусора автоматически адаптируется к загрузке памяти, необходимой для конкретного приложения, и это замечательно!

Другие возможности сборщика мусора по работе 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 мегабайт машинной памяти, а управляемый объект очень небольшой, так как он содержит только *HBITMAP* (4- или 8-байтовое значение). С точки зрения CLR до сбора мусора процесс может выделять сотни битовых карт (которые займут мало управляемой памяти). Но, если процесс манипулирует множеством битовых карт, расходование памяти процессом будет 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 объемных машинных ресурсов, чтобы сообщать сборщику мусора о реальном объеме занятой памяти. Сам сборщик следит за этим показателем, и когда он становится большим, начинается сбор мусора.

Количество некоторых машинных ресурсов ограничено. Раньше в Windows разрешалось создавать всего пять контекстов устройства. Также ограничивалось число файлов, открываемых приложением. Опять же, с точки зрения CLR до сбора мусора процесс может выделить сотни объектов (использующих мало памяти). Но, если число этих машинных ресурсов ограничено, попытка задействовать большее их количество, чем разрешено, обычно приводит к генерации исключений. 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 используется экземпляров этого ресурса. Внутренний код этого объекта-класса поддерживает счетчик используемых экземпляров, и когда значение счетчика становится большим, происходит сбор мусора.

Примечание Код методов *GCAddMemoryPressure* и *HandleCollectorAdd* 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, приложение потерпит сбой. Для большинства приложений лучше работать медленнее, чем не работать вообще.

В следующем коде показаны работа и результат методов сжатия памяти и класса *HandleCollector*.

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

public static class Program {
   public static void Main() {
      // Передача 0 приводит к нечастым сборам мусора.
      MemoryPressureDemo(0);

      // Передача 10 Мб приводит к частым сборам мусора.
      MemoryPressureDemo(10 * 1024 * 1024);

      // Сборщик описателей.
      HandleCollectorDemo();
   }

   private static void MemoryPressureDemo(Int32 size) {
      Console.WriteLine();
      Console.WriteLine("MemoryPressureDemo, size={0}", size);
      // Создание нескольких объектов с указанием их логического размера.
      for (Int32 count = 0; count < 15; count++) {
         new BigNativeResource(size);
      }

      // В целях демонстрации принуждаем сборщик
      // выполнить полную очистку.
      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 физически 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 целях демонстрации принуждаем сборщик
        // выполнить полную очистку.
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }

    private sealed class LimitedResource {
        // Создаем HandleCollector и передаем ему указание
        // выполнять очистку, когда в куче появится
        // более двух таких объектов.

    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
            // был удален из кучи.
            s_hc.Remove();
            Console.WriteLine("LimitedResource destroy. Count={0}", s_hc.Count);
        }
    }
}
```

После компиляции и запуска этого кода получим 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.
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 create.
BigNativeResource create.
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 окажется недостаточно, CLR сгенерирует исключение *OutOfMemoryException*. В этом случае вся работа окажется сделанной впустую. К тому же придется перехватить это исключение и корректно восстановить программу.

В пространстве имен *System.Runtime* есть класс *MemoryFailPoint*, позволяющий проверять наличие достаточного объема памяти до начала запуска алгоритма, потребляющего много памяти:

```
public sealed class MemoryFailPoint : CriticalFinalizerObject, IDisposable {
   public MemoryFailPoint(Int32 sizeInMegabytes);
   ~MemoryFailPoint();
```

```
    public void Dispose();
}
```

Этот класс довольно просто использовать. 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, выполняется попытка увеличения страничного файла. Если страничный файл не может быть увеличен до нужного размера, генерируется исключение *InsufficientMemoryException*.
4. Если непрерывного виртуального адресного пространства по-прежнему недостаточно, генерируется исключение *InsufficientMemoryException*.
5. Если найдено достаточно места в страничном файле и в виртуальном адресном пространстве, запрошенное число мегабайт резервируется путем добавления этого числа в закрытое статическое поле, определенное в классе *MemoryFailPoint*. Добавление выполняется в безопасном режиме, чтобы несколько потоков могли одновременно создать экземпляр этого класса и быть уверенными, что для них логически зарезервирована запрошенная память (при условии, что в конструкторе не было сгенерировано никакое исключение).

Если конструктор *MemoryFailPoint* сгенерирует исключение *InsufficientMemoryException*, в приложении можно освободить немного используемых ресурсов или снизить производительность (меньше кешировать данные), чтобы снизить вероятность появления исключения *OutOfMemoryException* в будущем. Между прочим, исключение *InsufficientMemoryException* является производным от *OutOfMemoryException*.

Внимание! Если конструктор *MemoryFailPoint* не генерирует исключение, запрошенная память была логически зарезервирована и можно выполнять алгоритм, требующий много памяти. Однако учтите, что физически память не была выделена. Это значит, что лишь *немного* повысилась вероятность успешного выполнения алгоритма и получения необходимой памяти. Класс *MemoryFailPoint* не может гарантировать, что алгоритм получит необходимую память, даже если конструктор не сгенерировал исключение. Этот класс призван лишь *помочь* разработчику сделать приложение более надежным.

По окончании выполнения алгоритма нужно вызвать метод *Dispose* для созданного объекта *MemoryFailPoint*. Внутренний код *Dispose* просто вычтет (в безопасном режиме) зарезервированное число мегабайт 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 Гб памяти.
         using (MemoryFailPoint mfp = new MemoryFailPoint(1500)) {
            // Здесь выполняется алгоритм, требующий много памяти.

         } // Dispose логически освободит 1,5 Гб памяти.
      }
      catch (InsufficientMemoryException e) {
         // Память не может быть зарезервирована.
         Console.WriteLine(e);
      }
   }
}
```

Управление сборщиком мусора из программ

Тип *System.GC* позволяет приложению напрямую управлять сборщиком мусора. Для начала замечу, что можно узнать максимальное поколение, поддерживаемое управляемой кучей, прочитав значение свойства *GC.MaxGeneration*. Это свойство всегда возвращает 2.

Сборщик также можно заставить собрать мусор, вызвав один из двух статических методов:

```
void GC.Collect(Int32 Generation)
void GC.Collect()
```

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 выполнен сбор 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*, не принимающая параметров, принуждает выполнить полный сбор мусора во всех поколениях, его вызов эквивалентен вызову:

```
GC.Collect(GC.MaxGeneration);
```

Обычно следует избегать вызова любых методов *Collect*: лучше не вмешиваться в работу сборщика мусора и позволить ему настраивать пороговые значения для поколений, основываясь на реальном поведении приложения. Но при написании приложения с консольным или графическим интерфейсом код приложения «владеет» процессом и CLR в этом процессе. В подобных приложениях порой следует собирать мусор принудительно во вполне определенное время.

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, потому что прогнозы сборщика мусора, основанные на прошлом, скорее всего, будут неточными для разовых событий.

Например, в приложении имеет смысл выполнить принудительный сбор мусора во всех поколениях после инициализации приложения или когда пользователь сохранил файл с данными. Когда на Web-странице размещен элемент управления Windows Form, полный сбор мусора выполняется при каждой выгрузке страницы. Не нужно явно вызывать *Collect*, чтобы сократить время отклика приложения, — вызывайте *Collect*, чтобы уменьшить рабочий набор процесса.

Тип *GC* также поддерживает метод *WaitForPendingFinalizers*, который просто приостанавливает вызывающий поток, пока поток, обрабатывающий очередь freachable, не опустошит ее, вызывая метод *Finalize* для каждого объекта. В большинстве приложений, скорее всего, этот метод вызывать не придется, но иногда я встречал такой код.

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

Этот код принудительно собирает мусор, освобождая память объектов, не требующих завершения. Но память, занятую объектами, которые требуют завершения, пока освободить нельзя. Когда первый вызов *Collect* возвращает управление, выделенный завершающий поток асинхронно вызывает методы *Finalize*. Вызов *WaitForPendingFinalizers* переводит поток приложения в состояние ожидания до вызова всех методов *Finalize*. Когда *WaitForPendingFinalizers* вернет управление, все завершенные объекты действительно станут мусором. Теперь второй вызов *Collect* снова выполняет принудительный сбор мусора, освобождающий память, занятую только что завершенными объектами.

Наконец, у класса *GC* есть пара статических методов, позволяющих определить, 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, а 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 *GC*.

```
using System;

internal sealed class GenObj {
   ~GenObj() {
      Console.WriteLine("In Finalize method");
   }
}

public static class Program {
   public static void Main() {
      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.

      // Сбор 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);      // Сбор мусора в поколении 0.
      GC.WaitForPendingFinalizers();   // Finalize НЕ вызывается.

      Console.WriteLine("Collecting Gens 0, and 1");
      GC.Collect(1);      // Сбор мусора в поколениях 0 и 1.
      GC.WaitForPendingFinalizers();   // Finalize НЕ вызывается.

      Console.WriteLine("Collecting Gens 0, 1, and 2");
      GC.Collect(2);                   // То же, что и Collect().
      GC.WaitForPendingFinalizers();   // Finalize вызывается.
   }
}
```

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
```

Другие вопросы производительности сборщика мусора

Ранее я объяснил, как устроен алгоритм сборщика мусора, но мое объяснение рассчитано на работу только одного потока. В реальном мире весьма вероятно, что несколько потоков будет обращаться к управляемой куче или хотя бы рабо-

тать с размещенными в ней объектами. Когда в результате работы одного из потоков инициируется сбор мусора, остальные не должны обращаться ни к каким 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 сбор 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 сбор мусора. Microsoft продолжит совершенствовать эти механизмы, чтобы обеспечить эффективный сбор мусора в дальнейшем.

Собираясь начать сбор 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 выполняет поток.

Если указатель команды потока находится по смещению, указанному в таблице, говорят, что поток достиг *безопасной точки* (safe point), то есть такого места, где можно приостановить поток до завершения сбора мусора. В противном случае поток не достиг безопасной точки, и CLR не может собрать мусор. В этом случае CLR перехватывает поток и изменяет его стек, чтобы адрес возврата указывал на специальную внутреннюю функцию, реализованную в CLR. После этого исполнение потока возобновляется. После возврата 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 перехвата потока около 250 мс, после чего вновь приостанавливает поток и проверяет его указатель команд. Если поток достиг безопасной точки, можно начинать сбор мусора. В противном случае CLR проверяет, не вызван ли другой метод. Если да, CLR вновь изменяет стек, чтобы поток был перехвачен после возврата из последнего исполняемого им метода. Затем CLR возобновляет поток и ждет еще несколько миллисекунд, прежде чем сделать следующую попытку.

Когда все потоки достигли безопасной точки или перехвачены, можно начинать сбор мусора. По его завершении все потоки возобновляются, а приложение продолжает работу. Перехваченные потоки возвращаются к методу, который изначально их вызвал.

В этом алгоритме есть маленькая хитрость. Когда CLR хочет запустить сборщик мусора, она приостанавливает все потоки, исполняющие управляемый код, но не приостанавливает потоки, исполняющие неуправляемый код. Когда все потоки, исполняющие управляемый код, окажутся в безопасной точке или будут перехвачены, сборщик может приступать 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 сбора мусора.

Оказывается, CLR использует перехват гораздо чаще, чем таблицы, сгенерированные JIT-компилятором, которые позволяют узнать, находится ли объект в безопасной точке. Причина в том, что сгенерированные JIT-компилятором таблицы требуют много памяти и увеличивают рабочий набор, что существенно ухудшает производительность. Поэтому таблицы, сгенерированные JIT-компилятором, 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 перехват.

Помимо описанных (поколений, безопасных точек и перехвата потоков) сборщик мусора поддерживает ряд дополнительных механизмов, ускоряющих размещение объектов и сбор мусора.

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.

Масштабируемый параллельный сбор мусора

Хост-приложение (например, ASP.NET или Microsoft SQL Server) может загрузить CLR и заставить ее отрегулировать сборщик мусора в соответствии с потребностями типичного серверного приложения. В результате этой настройки управляемая куча разбивается на несколько разделов — по одному на каждый процессор. Когда инициируется сбор мусора, у сборщика работает по одному потоку на процессор. Каждый поток собирает мусор в собственном разделе параллельно с другими. Параллельный сбор хорошо работает в серверных приложениях, где рабочие потоки склонны вести себя однообразно. Для этой функции необходимо, чтобы приложение выполнялось на многопроцессорном компьютере: так потоки будут на самом деле работать одновременно, и именно так достигается повышение производительности.

Чтобы дать указание CLR использовать сборщик мусора для сервера, можно создать файл конфигурации (об этом см. главы 2 и 3), содержащий элемент *gcServer* для этого приложения, например такой.

```
<configuration>
   <runtime>
      <gcServer enabled="true"/>
   </runtime>
</configuration>
```

Работающее приложение может узнать у CLR, работает ли оно с масштабируемым параллельным сборщиком, запросив свойство *IsServerGC* класса *GCSettings*.

```
using System;
using System.Runtime; // GCSettings определен в этом пространстве имен.

public static class Program {
   public static void Main() {
```

```
      Console.WriteLine("Application is running with server GC=" +
         GCSettings.IsServerGC);
   }
}
```

Параллельный сбор мусора

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 объект, вызывающий превышение порога для поколения 0, сборщик сначала приостанавливает все потоки, а затем определяет поколения, в которых нужно выполнить сбор 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 недостижимые 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 — такой сбор мусора проходит обычно быстрее, так как уже создан перечень недостижимых объектов. 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. Используя параллельный сборщик, приложение обычно расходует больше памяти, чем при работе с непараллельным сборщиком.

Подводя итог, можно сказать, что параллельный сборщик улучшает впечатление пользователей от работы с программой и поэтому лучше всего подходит для интерактивных приложений с консольным или графическим интерфейсом. Но в некоторых приложениях параллельный сборщик только снижает быстродействие и увеличивает потребление памяти. Тестируя приложение, поэкспериментируйте, используя как вариант с параллельным сборщиком мусора, так и без него, чтобы выяснить, какой подход обеспечивает максимальную производительность и минимальное использование памяти для приложения.

Можно запретить CLR использовать параллельный сборщик, создав файл конфигурации приложения (об этом см. главы 2 и 3), содержащий элемент *gcConcurrent*. Вот пример такого файла.

```
<configuration>
   <runtime>
      gcConcurrent enabled="false"/>
   </runtime>
</configuration>
```

Большие объекты

Есть и еще один механизм повышения быстродействия, о котором стоит сказать. Любые объекты размером 85 тыс. байт и более считаются большими. Большие объекты выделяются в специальной куче для больших объектов. Объекты в этой куче завершаются 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 тыс. байт ушло бы слишком много процессорного времени. Однако 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 короткоживущих больших объектов приведет к частому сбору мусора в поколении 2, что ухудшит 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)); // Выводит 2, а не 0.
   }
}
```

Все эти механизмы работают прозрачно для кода приложения. Для разработчика они выглядят просто как дополнительная управляемая куча. Эти механизмы существуют лишь для повышения производительности приложения.

Мониторинг сбора мусора

Есть немного методов, которые можно вызвать для мониторинга сборщика мусора в процессе. Так, класс *GC* поддерживает следующие статические методы, вызываемые для выяснения числа сборов 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 профилировать конкретный блок кода, я часто пишу код, вызывающий эти методы до и после этого блока, а затем вычисляю разницу. Так я могу судить о том, как этот блок кода повлиял на рабочий набор процесса, и узнать, сколько сборов мусора произошло при исполнении этого блока кода. Если показатели высоки, значит, нужно поработать над оптимизацией алгоритма в блоке кода.

Во время установки .NET Framework устанавливается набор счетчиков производительности, которые позволяют собрать в реальном времени самые разнообразные статистические данные о работе CLR. Эти данные можно просматривать с помощью инструмента PerfMon.exe или элемента управления ActiveX System

Monitor, входящего в ОС Windows. Проще всего получить доступ к элементу управления System Monitor, запустив PerfMon.exe и щелкнув кнопку + на панели инструментов, в результате откроется диалоговое окно **Add Counters** (рис. 20-16).

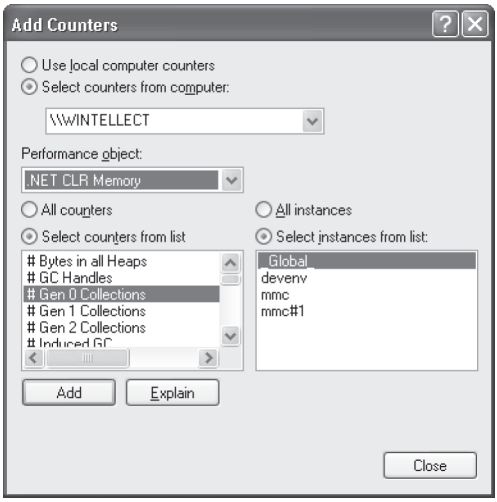


Рис. 20-16. *Счетчики памяти .NET CLR в окне PerfMon.exe*

Для мониторинга сборщика мусора CLR выберите объект производительности .NET CLR Memory, затем выберите из списка нужное приложение. В завершение выберите нужный набор счетчиков для мониторинга, щелкните **Add**, а затем **Close**. Теперь System Monitor будет в реальном времени строить график выбранного статистического показателя. Чтобы узнать, что означает счетчик, выберите его и щелкните **Explain**.

Еще один замечательный инструмент для мониторинга размещения объектов приложением — CLR Profiler, позволяющий выполнять профилирование вызовов, снимки кучи и графики использования памяти. Есть даже API-интерфейс, который можно использовать из тестового кода для запуска и остановки профилирования и добавления комментариев в журналы. Также имеется исходный текст для этого инструмента, поэтому разработчики могут изменять его по своему усмотрению. Чтобы получить этот инструмент, выполните поиск в Интернете, указав в строке поиска «CLR profiler». Это исключительно полезный инструмент — настоятельно вам его рекомендую!

ГЛАВА 21

Хостинг CLR и домены приложения (AppDomains)

В этой главе мы обсудим два предмета, которые позволяют по-настоящему раскрыть ценность Microsoft .NET Framework, — хостинг CLR и домены приложения (AppDomains). Хостинг позволяет любому приложению использовать возможности общеязыковой среды CLR, в частности существующие приложения можно по крайней мере частично переписать для использования управляемого кода. Более того, хостинг позволяет модифицировать и расширять приложения с использованием программных средств.

Поддержка расширения означает, что в вашем процессе может выполняться код сторонних разработчиков. В Microsoft Windows загрузка DLL-библиотек сторонних разработчиков в процесс была исключительно рискованным мероприятием. В такой библиотеке очень легко мог оказаться код, разрушающий структуры данных и код приложения. Библиотека также могла использовать контекст безопасности приложения для получения доступа к ресурсам, к которым в обычных условиях доступа у нее нет. Все эти проблемы решаются за счет использования доменов приложений, которые позволяют не пользующийся доверием код сторонних разработчиков выполнять в существующем процессе, а CLR гарантирует безопасность и целостность структур данных и кода и невозможность использовать в неблаговидных целях контекст безопасности.

Обычно хостинг и домены приложений используют вместе с загрузкой сборок и отражением. Наличие и возможность совместного использования этих четырех технологий превращает CLR в невероятно богатую и мощную платформу.

В этой главе я сосредоточусь на хостинге и доменах приложений, а в следующей расскажу о загрузке сборок и отражении. Изучив и освоив эти технологии, вы узнаете, как нынешние инвестиции в .NET Framework вернутся сторицей в будущем.

Хостинг CLR

.NET Framework работает поверх Microsoft Windows. Это значит, что .NET Framework построена на основе технологий, «понятных Windows». Для начала замечу, что все файлы управляемых модулей или сборок должны быть в формате *Windows*

PE (portable executable) и являться исполняемыми файлами Windows или динамически подключаемыми библиотеками (DLL).

В Microsoft сконструировали CLR в виде COM-сервера, содержащегося в DLL, то есть разработчики Microsoft 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++ MSCorEE.h из .NET Framework SDK — этот файл определяет все GUID и неуправляемый интерфейс *ICorRuntimeHost*.

Любое приложение Windows может быть хостом CLR. Однако не следует создавать экземпляры COM-сервера CLR, вызывая *CoCreateInstance*, вместо этого неуправляемый хост должен вызывать функцию *CorBindToRuntimeEx* или ее аналог (ее прототип хранится в MSCorEE.h). Эта функция реализована в библиотеке MSCorEE.dll, которая обычно расположена в каталоге *C:\Windows\System32*. Эта библиотека служит *согласователем* (shim) — она не содержит COM-сервер CLR, а только определяет, какую версию CLR следует создать.

На одной машине может устанавливаться несколько версий CLR, но может быть только одна версия файла MSCorEE.dll (об исключениях из этого правила говорится в нижеследующем примечании). Версия установленной на машине библиотеки MSCorEE.dll совпадает с версией самой последней установленной CLR, поэтому эта версия MSCorEE.dll «знает», как найти любые более ранние версии CLR, которые устанавливались на машине.

Примечание В 64-разрядной версии Windows с установленной версией 2.0 платформы .NET Framework в действительности установлены две версии файла MSCorEE.dll. Первая — 32-разрядная версия x86, а вторая — 64-разрядная версия x64 или IA64 (в зависимости от архитектуры процессора).

Сама CLR реализована в файле MSCorWks.dll, а не в библиотеке MSCorEE.dll. Если на машине установлены версии 1.0, 1.1 и 2.0 среды CLR, версии MSCorWks.dll размещаются в следующих каталогах:

- 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*.

При вызове функции *CorBindToRuntimeEx* ее параметры позволяют хосту указать нужную ему версию CLR, а также другие параметры. На основе заданных сведений о версии и кое-какой дополнительной информации, собранной ею самостоятельно (например, число процессоров и установленные версии CLR), *CorBindToRuntimeEx* определяет версию CLR для загрузки — согласователь может и не загрузить версию CLR, запрошенную хостом.

По умолчанию согласователь анализирует управляемый исполняемый файл и извлекает из него версию CLR, с которой было скомпоновано и протестировано приложение. Но приложение может переопределить сведения по умолчанию, записав элементы *requiredRuntime* и *supportedRuntime* в конфигурационный XML-файл (см. главы 2 и 3).

Функция *CorBindToRuntimeEx* возвращает указатель на неуправляемый интерфейс *ICorRuntimeHost*. Вызывая методы этого интерфейса, хост-приложение может:

- **определять диспетчеры хоста**, то есть сообщать CLR, что хост должен принимать участие в принятии решений касательно выделения памяти, планирования и синхронизации потоков, загрузки сборок и т. п. Хосту также могут понадобиться уведомления о начале и окончании сборки мусора, а также о тайм-ауте определенных операций;
- **получать информацию о диспетчерах CLR**, то есть запрещать CLR использовать определенные классы или члены. Кроме того, хост может указать, какой код можно, а какой нельзя отлаживать, а также какие методы вызывать при наступлении определенных событий, таких как выгрузка домена приложения, остановка CLR или исключение переполнения стека;
- **инициализировать и запускать CLR;**
- **загружать сборку и выполнять ее код;**
- **останавливать CLR**, предотвращая дальнейшее выполнение управляемого кода в процессе Windows.

Есть очень много причин в пользу применения хостинга в CLR. Хостинг позволяет любому приложению обеспечивать возможности и программную модель CLR, а также быть хотя бы частично написанным на управляемом коде. Многие приложения, служащие хостом исполняющей среды, предлагают массу возможностей разработчикам, стремящимся расширить функциональности приложения. Вот всего лишь несколько из этих возможностей:

- возможность программировать на любом языке;
- код может JIT-компилироваться (а не интерпретироваться) для обеспечения максимальной скорости работы;
- поддержка сборки мусора, предотвращающей утечку и повреждение памяти;
- выполнение кода в безопасной изоляции;
- хосту не нужно заботиться об обеспечении многофункциональной среды разработки. Хост использует имеющиеся технологии: языки, компиляторы, редакторы, отладчики, средства профилирования и др.

Интересующимся подробностями использования хостинга CLR я настоятельно рекомендую отличную книгу Стивена Претчнера (Steven Pratschner) «Customizing the Microsoft .NET Framework Common Language Runtime» (Microsoft Press, 2005).

Примечание Конечно, процессу Windows не обязательно загружать CLR — эта среда нужна только для выполнения в процессе управляемого кода. Процесс Windows может загрузить только одну версию CLR. При многократном вызове *CorBindToRuntimeEx* каждый раз возвращается один и тот же указатель *ICLRRuntimeHost*.

После загрузки CLR в процесс Windows ее нельзя выгрузить — вызов методов *AddRef* и *Release* по отношению к интерфейсу *ICLRRuntimeHost* ни к чему не приводит. Единственный метод выгрузить CLR из процесса — завершить процесс, что вынудит Windows очистить все ресурсы, используемые в процессе.

Следующий неуправляемый код на C++ демонстрирует, как просто создать неуправляемый хост, который загружает среду CLR и выполняет в ней некоторый код.

```
#include <Windows.h>
#include <MSCorEE.h>
#include <stdio.h>

void main(int argc, WCHAR **argv) {
    // Загружаем CLR.
    ICLRRuntimeHost *pClrHost;
    HRESULT hr = CorBindToRuntimeEx(
        NULL,                   // желаемая версия CLR (NULL означает "самая последняя")
        NULL,                   // требуемый вид сборщика мусора (NULL означает
                                // "рабочая станция")
        0,                      // требуемые флаги запуска
        CLSID_CLRRuntimeHost,   // CLSID среды CLR
        IID_ICLRRuntimeHost,    // IID интерфейса ICLRRuntimeHost
        (PVOID*) &pClrHost);    // возвращенный COM-интерфейс

   // (Здесь определяются диспетчеры хоста)
   // (Здесь получают информацию о диспетчерах CLR)

   // Инициализируем и запускаем CLR.
   pClrHost->Start();

   // Загружаем сборку и вызываем метод,
   // принимающий String и возвращающий Int32.
   DWORD retVal;
   hr = pClrHost->ExecuteInDefaultAppDomain(
           L"SomeMgdAssem.dll",
           L"Wintellect.SomeType", L"SomeMethod", L"Jeff", &retVal);

   // Показываем результат, возвращенный управляемым кодом.
   wprintf(L"Managed code returned %d", retVal);

   // Завершаем процесс (вместе с загруженной в него CLR).
}
```

В этом коде содержится некоторая жестко прописанная информация о загружаемой управляемой сборке. В частности, управляемая сборка должна компилироваться в файл SomeMgdAssem.dll, который находится в рабочем каталоге процесса. В этой сборке должен быть определен тип *Wintellect.SomeType*, в котором, в свою очередь, должен быть определен метод *SomeMethod*. Далее, метод требует, чтобы *SomeMethod* принимал параметр типа *String*, а возвращал значение типа *Int32*. Вот код на C# простой библиотеки, которую может вызывать этот неуправляемый хост:

```
// Скомпилируйте этот код на C# сборку-библиотеку с именем SomeMgdAssem.dll.
using System;

namespace Wintellect {
   public sealed class SomeType {
```

```
      public static Int32 SomeMethod(String s) {
         Console.WriteLine("Managed assembly: {0}", s);
         return s.Length;
      }
   }
}
```

Домены приложения

При загрузке COM-сервера CLR он создает *домен приложения*, или AppDomain, — логический контейнер набора сборок. Первый AppDomain, созданный при инициализации, называют *основным* (default AppDomain), он уничтожается только при завершении процесса Windows.

Помимо основного, хост, использующий неуправляемый COM-интерфейс либо методы управляемого типа, может заставить CLR создать дополнительные домены приложения. Основная задача домена приложения — обеспечить изоляцию. Домены приложения удобны благодаря следующим свойствам.

- **Объекты, созданные одним AppDomain, не видят другие домены приложения** Когда код домена приложения создает объект, AppDomain «владеет» этим объектом. Иначе говоря, время жизни объекта ограничивается временем существования соответствующего домена. Код другого домена может получить доступ к объекту, только используя семантику маршалинга по ссылке или по значению. Это позволяет «герметично» изолировать код в домене приложения, так как код в одном AppDomain не может напрямую ссылаться на объект, созданный в другом. Такая изоляция позволяет легко выгружать домены приложения из процесса, не влияя на работу других доменов.
- **Домены приложения можно выгружать** CLR не поддерживает выгрузку отдельных сборок. Однако можно приказать CLR выгрузить AppDomain со всеми сборками, которые в данный момент в нем находятся.
- **Домены приложения можно защищать по отдельности** При создании домену приложения можно назначить разрешение, определяющее максимальные права сборок, работающих в AppDomain. Это позволяет хосту загружать код и быть уверенным, что этот код не испортит или не считает важные структуры данных, используемые самим доменом.
- **Домены приложения можно конфигурировать по отдельности** Созданный домен характеризуется многими конфигурационными параметрами. Они в основном определяют, как CLR должна загружать сборки в AppDomain. Есть также параметры, определяющие пути поиска, перенаправление привязки к версиям и оптимизацию загрузчика.

Внимание! Замечательная возможность Windows — работа каждого приложения в собственном адресном пространстве. Это гарантирует, что код одного приложения не может получить доступ к коду и данным другого. Изоляция процессов предотвращает возникновение брешей в защите, повреждение данных и другие неприятности, обеспечивая надежность Windows и работающих в ней приложений. К сожалению, создание про-

цесса в Windows — операция очень ресурсоемкая. Win32-функция *CreateProcess* выполняется очень медленно, а для виртуализации адресного пространства процесса требуется много памяти.

Однако, если приложение полностью состоит из управляемого кода, который гарантировано безопасен и не вызывает неуправляемого кода, нет никаких проблем с выполнением нескольких управляемых приложений в одном процессе Windows. А домены приложений обеспечивают изоляцию, необходимую для защиты, конфигурирования и завершения отдельных приложений.

На рис. 21-1 показан отдельный процесс Windows, в котором работает один COM-сервер CLR, управляющий двумя доменами приложения (кстати, нет никаких жестких ограничений на количество доменов приложений, которые могут выполняться в одном процессе Windows). У каждого такого домена собственная куча загрузчика, ведущая учет обращений к типам с момента создания AppDomain (см. главу 4). У каждого типа в куче загрузчика есть таблица методов, каждая строка которой указывает на код метода (если этот метод был хоть раз исполнен, его код уже скомпилирован JIT-компилятором).

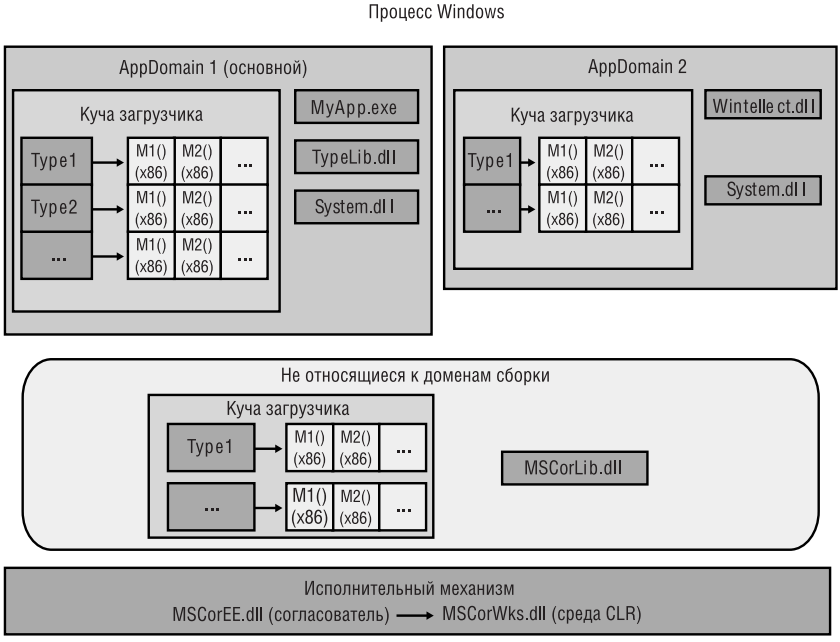


Рис. 21-1. *Процесс Windows, являющийся хостом для CLR и двух доменов приложения*

Кроме того, в каждый AppDomain загружены сборки. В первый AppDomain (он является основным) загружены три сборки: MyApp.exe, TypeLib.dll и System.dll, а во второй — две сборки: Wintellect.dll и System.dll.

Заметьте: сборка System.dll загружается в оба AppDomain. Если в обеих сборках используется один тип из System.dll, в кучах загрузчика обоих доменов приложения будет размещен одинаковый объект-тип; 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.

Отсутствие совместного использования памяти для хранения объектов-типов или машинного кода расточительно. Но ведь вся затея с доменами приложений ориентирована на изоляцию; у CLR должна быть возможность выгрузить AppDomain и освободить все его ресурсы, никак не затронув остальные домены приложения. Для этого и дублируются структуры данных.

Некоторые сборки предназначены для совместного использования доменами приложения. Лучший пример — сборка MSCorLib.dll, созданная Microsoft. В ней находятся *System.Object*, *System.Int32* и другие типы, неотделимые от .NET Framework. Эта сборка автоматически загружается при инициализации CLR, и домены приложения совместно используют ее типы. Для экономии ресурсов MSCorLib.dll загружается как сборка, не связанная с конкретным AppDomain. Все объекты-типы в этой куче загрузчика и весь машинный код методов этих типов совместно используются всеми доменами процесса. К сожалению, для достижения всех преимуществ, достигнутых за счет совместного использования ресурсов, пришлось кое-чем пожертвовать: сборки, загруженные без привязки к доменам, не могут быть выгружены до завершения процесса. Единственный способ вернуть ресурсы — завершить процесс.

Доступ к объектам из другого AppDomain

Код, расположенный в одном AppDomain, способен взаимодействовать с типами и объектами из другого AppDomain. Однако доступ к этим типам и объектам возможен только через определенные механизмы. Приведенное ниже приложение-пример AppDomainMarshalling демонстрирует создание нового домена приложения, загрузку в него сборки и конструирование экземпляра типа, определенного в этой сборке. Пример показывает различное поведение при конструировании типа, передаваемого по механизму маршалинга по ссылке и по значению, а также типа, который вообще не поддается маршалингу. Также AppDomainMarshalling демонстрирует, как объекты, подвергшиеся различным методам маршалинга, ведут себя, когда создавший их домен приложения выгружается. В этом примере мало кода, но очень много комментариев. После кода программы я подробно объясню, что в ней происходит.

```
using System;
using System.Reflection;
using System.Threading;
using System.Runtime.Remoting;

public static class Program {
   public static void Main() {
      // Получаем ссылку на AppDomain, в котором выполняется вызывающий поток.
      AppDomain adCallingThreadDomain = Thread.GetDomain();

      // Каждому AppDomain присваивается информативное имя,
      // которое облегчает отладку. Получаем имя этого AppDomain
```

```
    // и отображаем его.
    String callingDomainName = adCallingThreadDomain.FriendlyName;

Console.WriteLine("Default AppDomain's friendly name={0}", callingDomainName);

    // Получаем и отображаем сборку в нашем AppDomain,
    // который содержит метод Main.
    String exeAssembly = Assembly.GetEntryAssembly().FullName;
    Console.WriteLine("Main assembly={0}", exeAssembly);

    // Определяем локальную переменную, которая может ссылаться на AppDomain.
    AppDomain ad2 = null;

    // ПРИМЕР 1: доступ к объектам другого домена AppDomain
    // с использованием маршалинга по ссылке.
    Console.WriteLine("{0}Demo #1: Marshal-by-Reference", Environment.NewLine);

    // Создаем новый AppDomain (с такими же параметрами безопасности и
    // конфигурации, как в текущем AppDomain).
    ad2 = AppDomain.CreateDomain("AD #2", null, null);

    // Загружаем нашу сборку в новый AppDomain, конструируем объект,
    // выполняем его маршалинг обратно в наш домен
    // (в действительности мы получаем ссылку на прокси).
    MarshalByRefType mbrt = (MarshalByRefType)
       ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByRefType");
    Type t = mbrt.GetType();

    // Убеждаемся, что получили ссылку на объект-прокси.
    Console.WriteLine("Is proxy={0}", RemotingServices.IsTransparentProxy(mbrt));

    // Кажется, что мы вызываем метод экземпляра MarshalByRefType, но это не так.
    // Мы вызываем метод экземпляра типа объекта-прокси.
    // Прокси переносит поток на AppDomain, в котором располагается объект,
    // и вызывает метод реального объекта.
    mbrt.SomeMethod(callingDomainName);

    // Выгружаем новый AppDomain.
    AppDomain.Unload(ad2);
    // mbrt ссылается на действительный объект-прокси;
    // прокси теперь ссылается на недействительный AppDomain.

    try {
       // Вызываем метод на объекте-типе прокси.
       // Домен недействительный, поэтому генерируется исключение.
       mbrt.SomeMethod(callingDomainName);
       Console.WriteLine("Successful call.");
    }
    catch (AppDomainUnloadedException) {
       Console.WriteLine("Failed call.");
    }
```

```
      // ПРИМЕР 2: доступ к объектам другого домена AppDomain
      // с использованием маршалинга по значению.
      Console.WriteLine("{0}Demo #2: Marshal-by-Value", Environment.NewLine);

      // Создаем новый AppDomain (с такими же параметрами безопасности и
      // конфигурации, как в текущем AppDomain).
      ad2 = AppDomain.CreateDomain("AD #2", null, null);

      // Загружаем нашу сборку в новый AppDomain, конструируем объект,
      // выполняем его маршалинг обратно в наш домен
      // (в действительности мы получаем ссылку на объект с таким же состоянием).
      MarshalByValType mbvt = (MarshalByValType)
         ad2.CreateInstanceAndUnwrap(exeAssembly, "MarshalByValType");

      // Убеждаемся, что НЕ получили ссылку на объект-прокси.
      Console.WriteLine("Is proxy={0}", RemotingServices.IsTransparentProxy(mbvt));

      // Кажется, что мы вызываем метод экземпляра MarshalByRefType,
      // и это так на самом деле.
      mbvt.SomeMethod(callingDomainName);

      // Выгружаем новый AppDomain.
      AppDomain.Unload(ad2);
      // mbrt ссылается на действительный объект;
      // выгрузка AppDomain не оказывает никакого влияния.

      try {
         // Вызываем метод объекта; исключение не генерируется.
         mbvt.SomeMethod(callingDomainName);
         Console.WriteLine("Successful call.");
      }
      catch (AppDomainUnloadedException) {
         Console.WriteLine("Failed call.");
      }

      // ПРИМЕР 3: доступ к объектам другого домена AppDomain
      // с использованием типа, не поддающегося маршалингу.
      Console.WriteLine("{0}Demo #3: Non-Marshalable Type", Environment.NewLine);

      // Создаем новый AppDomain (с такими же параметрами безопасности и
      // конфигурации, как в текущем AppDomain).
      ad2 = AppDomain.CreateDomain("AD #2", null, null);

      // Загружаем нашу сборку в новый AppDomain, конструируем объект,
      // пытаемся выполнить его маршалинг обратно в наш домен -
      // генерируется исключение.
      NotMarshalableType nmt = (NotMarshalableType)
         ad2.CreateInstanceAndUnwrap(exeAssembly, "NotMarshalableType");
      // До этого места выполнение кода не дойдет...
   }
}
```

```
// Можно выполнять маршалинг по ссылке между доменами.
public class MarshalByRefType : MarshalByRefObject {
   DateTime creation = DateTime.Now;

   public MarshalByRefType() {
      Console.WriteLine("{0} ctor running in {1}",
         this.GetType().ToString(), Thread.GetDomain().FriendlyName);
   }

   public void SomeMethod(String callingDomainName) {
      Console.WriteLine("Calling from '{0}' to '{1}'.",
         callingDomainName, Thread.GetDomain().FriendlyName);
   }
}

// Можно выполнять маршалинг по значению между доменами.
[Serializable]
public class MarshalByValType : Object {
   DateTime creation = DateTime.Now;

   public MarshalByValType() {
      Console.WriteLine("{0} ctor running in {1}",
         this.GetType().ToString(), Thread.GetDomain().FriendlyName);
   }

   public void SomeMethod(String callingDomainName) {
      Console.WriteLine("Calling from '{0}' to '{1}'.",
         callingDomainName, Thread.GetDomain().FriendlyName);
   }
}

// Нельзя выполнять маршалинг по ссылке между доменами.
// [Serializable]
public class NotMarshalableType : Object {
   DateTime creation = DateTime.Now;

   public NotMarshalableType() {
      Console.WriteLine("{0} ctor running in {1}",
         this.GetType().ToString(), Thread.GetDomain().FriendlyName);
   }

   public void SomeMethod(String callingDomainName) {
      Console.WriteLine("Calling from '{0}' to '{1}'.",
         callingDomainName, Thread.GetDomain().FriendlyName);
   }
}
```

Скомпоновав и выполнив приложение AppDomainMarshalling, получим следующее:

```
Default AppDomain's friendly name=AppDomainMarshalling.exe
Main assembly=AppDomainMarshalling, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
```

```
Demo #1: Marshal-by-Reference
MarshalByRefType ctor running in AD #2
Is proxy=True
Calling from 'AppDomainMarshalling.exe' to 'AD #2'.
Failed call.

Demo #2: Marshal-by-Value
MarshalByValType ctor running in AD #2
Is proxy=False
Calling from 'AppDomainMarshalling.exe' to 'AppDomainMarshalling.exe'.
Calling from 'AppDomainMarshalling.exe' to 'AppDomainMarshalling.exe'.
Successful call.

Demo #3: Non-Marshalable Type
NotMarshalableType ctor running in AD #2

Unhandled Exception: System.Runtime.Serialization.SerializationException:
Type 'NotMarshalableType' in assembly 'AppDomainMarshalling, Version=0.0.0.0,
Culture=neutral, PublicKeyToken=null' is not marked as serializable.
   at System.AppDomain.CreateInstanceAndUnwrap(String assemblyName, String typeName)
   at Program.Main() in C:\AppDomainMarshalling.cs:line 97
```

А теперь я объясню, что делает этот код и как работает CLR.

В методе *Main* я первым делом получаю ссылку на объект *AppDomain*, который идентифицирует домен приложения, в котором сейчас выполняется вызывающий поток. В Windows поток всегда создается в контексте одного процесса и проводит в нем всю свою жизнь. Однако между потоками и доменами приложения нет взаимно-однозначного соответствия. Домены приложения — это возможность, существующая в CLR; Windows ничего не знает о них. Так как в одном процессе Windows может быть много доменов AppDomain, 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 AppDomain в нем выполняется в текущий момент, вызвав статический метод *GetDomain* класса *System.Threading.Thread*. Ту же информацию поток может получить, запросив статическое неизменяемое свойство *CurrentDomain* класса *SystemAppDomain*.

При создании AppDomain ему можно назначить *информативное имя* (friendly name) — строку типа *String*, которую можно использовать для идентификации домена приложения. Обычно это оказывается полезным при отладке. Так как среда CLR создает основной домен AppDomain до выполнения какого-либо кода, она использует в качестве информативного имени по умолчанию имя исполняемого файла домена. Мой метод *Main* запрашивает информативное имя AppDomain по умолчанию, обращаясь к неизменяемому свойству *FriendlyName* класса *System.AppDomain*.

Далее, метод *Main* запрашивает строгое имя сборки (загруженной в основной AppDomain), которое определяет метод *Main* точки входа. В этой сборке определено несколько типов: *Program*, *MarshalByRefType*, *MarshalByValType* и *NonMarshalableType*. Теперь мы готовы познакомиться с тремя примерами, которые довольно похожи друг на друга.

Пример 1: доступ к объектам другого домена AppDomain с использованием маршалинга по ссылке

Статический метод *CreateDomain* типа *SystemAppDomain* вызывается, чтобы заставить CLR создать новый AppDomain в том же процессе Windows. Тип *AppDomain* поддерживает несколько перегруженных версий метода *CreateDomain*; я рекомендую вам изучить их и выбрать версию, которая больше всего подходит при написании кода создания нового AppDomain. Версия *CreateDomain*, которую использую я, принимает три параметра:

- строку *String*, содержащую информативное имя, которое я хочу назначить новому AppDomain, — «AD #2»;
- *System.Security.Policy.Evidence*, содержащий политику, которую должна использовать CLR для вычисления набора разрешений AppDomain. В этом аргументе я передаю null, чтобы новый домен приложения наследовал тот же набор разрешений, что и «родительский» AppDomain. Обычно, если нужно создать границу безопасности вокруг кода домена приложения, конструируют объект *System.Security.PermissionSet*, создают в нем необходимые объекты разрешений (экземпляры типов, которые реализуют интерфейс *IPermission*), а затем передают ссылку результирующего объекта *PermissionSet* на перегруженную версию метода *CreateDomain*, принимающего *PermissionSet*;
- *SystemAppDomainSetup*, указывающий на параметры конфигурации, которые CLR должна применить к новому AppDomain. И здесь я передаю null, чтобы новый домен приложения наследовал конфигурацию у родительского AppDomain. Если нужно, чтобы AppDomain получил особую конфигурацию, надо создать объект *AppDomainSetup*, задать его свойства в соответствии с потребностями, а затем передать ссылку на метод *CreateDomain* результирующего объекта *AppDomainSetup*.

Код метода *CreateDomain* создает в процессе новый AppDomain. Этому домену присваивается заданное информативное имя, параметры безопасности и конфигурации. У нового AppDomain есть собственная куча загрузчика, которая пуста, потому что нет сборок, загруженных в новый AppDomain. При создании AppDomain среда CLR не создаст никаких потоков в этом AppDomain; никакой код выполняться в AppDomain не будет, если только явно не заставить поток вызвать код домена приложения.

Теперь, чтобы создать экземпляр объекта в новом AppDomain, надо сначала загрузить сборку в новый домен, а затем создать экземпляр типа, определенного в этой сборке. Именно это и делает вызов открытого экземплярного метода *CreateInstanceAndUnwrap* класса *AppDomain*. Вызывая *CreateInstanceAndUnwrap*, я передаю два параметра: строку *String*, идентифицирующую сборку, которую я хочу загрузить в новый AppDomain (на нее ссылается переменная *ad2*), и вторую строку *String* — имя типа, экземпляр которого надо создать. Код *CreateInstanceAndUnwrap* заставляет вызывающий поток перейти от текущего AppDomain в новый. Теперь поток (который выполняет вызов *CreateInstanceAndUnwrap*) загружает указанную сборку в новый AppDomain, а затем просматривает таблицу метаданных с определениями типов сборки в поисках указанного типа (*MarshalByRefType*). Найдя нужный тип, поток вызывает конструктор без параметров *MarshalByRefType*. Теперь поток переходит обратно в основной домен, чтобы *CreateInstanceAndUnwrap* мог вернуть ссылку на новый объект *MarshalByRefType*.

Примечание Есть перегруженные версии *CreateInstanceAndUnwrap*, которые позволяют вызывать конструктор типа, передавая параметры.

Но все на так радужно, как могло бы показаться, — проблема в том, что CLR не позволяет переменной (корню), находящейся в одном AppDomain, ссылаться на объект, созданный в другом домене. Если бы *CreateInstanceAndUnwrap* просто вернул ссылку на объект, изоляция была бы нарушена, а ведь изоляция — главная причина создания AppDomains! Поэтому, непосредственно перед тем как возвратить ссылку на объект, *CreateInstanceAndUnwrap* выполняет некоторые дополнительные операции.

Заметьте: тип *MarshalByRefType* наследует очень специальному базовому классу — *System.MarshalByRefObject*. Обнаружив, что *CreateInstanceAndUnwrap* выполняет маршалинг объекта, тип которого производный от *MarshalByRefObject*, CLR выполнит маршалинг объекта в другой домен по ссылке. Далее объясняется, что означает маршалинг объекта по ссылке из одного AppDomain (домен-источник, где реально создается объект) в другой AppDomain (домен-адресат, где вызывается *CreateInstanceAndUnwrap*).

Когда домену-источнику нужно переслать или вернуть ссылку на объект в целевом AppDomain, CLR определяет тип-прокси куче загрузчика целевого домена. Этот тип определяется с использованием метаданных исходного типа, поэтому выглядит в точности как исходный тип — у него в точности совпадают все экземплярные члены (свойства, события и методы). Экземплярные поля не относятся к типу, но об этом чуть позже. В этом новом типе действительно определены 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. (Внутрение механизмы объекта-прокси используют экземпляр *GCHandle*, который ссылается на реальный объект. Тип *GCHandle* обсуждается в главе 20.)

После определения этого типа в целевом AppDomain метод *CreateInstanceAndUnwrap* создает экземпляр этого типа-прокси и инициализирует его поля так, чтобы они указывали на домен-источник и реальный объект, и возвращает ссылку на этот объект-прокси в целевом домене. В нашем приложении AppDomainMarshalling на этот прокси будет ссылаться переменная *mbrt*. Заметьте: объект, возвращенный *CreateInstanceAndUnwrap*, не является экземпляром типа *MarshalByRefType*. Обычно CLR не разрешает приводить объект определенного типа к несовместимому типу. Однако в этой ситуации CLR разрешает такое приведение, потому что у нового типа те же экземплярные члены, что и у исходного. В сущности, при вызове метода *GetType* объекта-прокси он лжет, представляясь объектом *MarshalByRefType*.

Однако можно узнать, что объект, возвращенный методом *CreateInstanceAndUnwrap*, в реальности является ссылкой на объект-прокси. Для этого в приложении AppDomainMarshalling вызывается открытый статический метод *IsTransparentProxy* типа *System.Runtime.Remoting.RemotingService*, которому в качестве параметра передается ссылка, возвращенная *CreateInstanceAndUnwrap*. *IsTransparentProxy* возвратит true, указывая, что объект является прокси.

Теперь приложение AppDomainMarshalling использует прокси для вызова метода *SomeMethod*. Так как переменная *mbrt* ссылается на прокси, вызывается реа-

лизация этого метода в прокси. В ней используются информационные поля объекта-прокси для перенаправления вызывающего потока из основного домена в новый AppDomain. Теперь любые действия этого потока выполняются в контексте безопасности и конфигурации нового домена. Далее поток использует поле *GCHandle* объекта-прокси, чтобы найти реальный объект в новом AppDomain, после чего вызывает метод *SomeMethod* реального объекта.

Есть два способа убедиться, что запрашивающий поток перешел от основного к новому домену. Во-первых, в методе *SomeMethod* я вызываю *Thread.GetDomain().FriendlyName*. В результате возвращается «AD #2» (что подтверждается выходными данными), так как поток теперь выполняется в новом AppDomain, созданном методом *AppDomain.CreateDomain* с параметром «AD #2» в качестве информативного имени. Во-вторых, если пошагово выполнить код в отладчике с открытым окном Call Stack, строка [AppDomain Transition] отметит переход потока через границу между доменами (рис. 21-2).

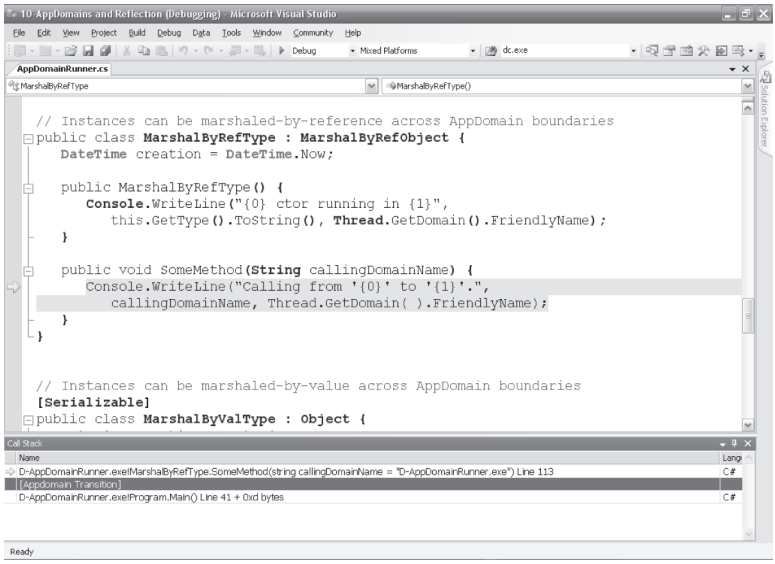


Рис. 21-2. *Переход между доменами в окне Call Stack отладчика*

Когда реальный метод *SomeMethod* вернет управление, оно возвратится методу *SomeMethod* прокси, который переведет поток обратно в основной домен, после чего поток продолжит выполнение кода в этом AppDomain.

Примечание Когда поток в одном домене вызывает метод другого домена приложения, 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 AppDomain. Если нужно выполнять код во многих доменах приложения одновременно, придется создавать дополнительные потоки и заставлять их выполнять нужный код в нужном AppDomains.

Далее приложение AppDomainMarshalling вызывает открытый статический метод *Unload* типа *AppDomain*, чтобы заставить CLR выгрузить указанный AppDomain, в том числе все загруженные в него сборки, одновременно инициируется сборка мусора для освобождения всех объектов выгружаемого AppDomain. На этом этапе переменная *mbrt* основного AppDomain все еще ссылается на действительный прокси; однако объект-прокси больше не ссылается на действительный домен приложения (потому что тот выгружается).

Когда основной AppDomain пытается использовать объект-прокси, чтобы вызвать метод *SomeMethod*, вызывается имеющаяся в прокси реализация этого метода. В процессе работы этой реализации обнаруживается, что AppDomain, который содержал реальный объект, выгружен, и метод *SomeMethod* объекта-прокси генерирует исключение *AppDomainUnloadedException*, информируя вызывающий код о невозможности выполнить операцию.

Вот как! Команда разработчиков CLR в Microsoft проделала огромную работу, чтобы обеспечить изоляцию доменов приложений, и это исключительно важная работа, потому что эта функциональность все активнее используется разработчиками в повседневной работе. Ясно, что доступ к объектам через границы между доменами AppDomain с использованием семантики маршалинга по ссылке связан с некоторой потерей производительности, поэтому использование таких операций надо сводить к минимуму.

Я обещал, что расскажу побольше об экземплярных полях. В типе, производном от *MarshalByRefObject*, можно определять экземплярные поля. Однако они не определяются как часть типа-прокси и отсутствуют в объекте-прокси. Создавая код, который считывает и изменяет значения экземплярных полей типа, производного от *MarshalByRefObject*, JIT-компилятор генерирует код, использующий объект-прокси (чтобы найти реальный домен приложения и объект), вызывая соответственно метод *FieldGetter* или *FieldSetter* класса *System.Object*. Это закрытые и незадокументированные методы; в сущности это методы, в которых для считывания и записи значений в поле используется отражение. Хотя и есть возможность получить доступ к полям типа, производного от *MarshalByRefObject*, производительность от этого особенно сильно страдает, потому что в действительности для получения такого доступа к полям среде CLR приходится вызывать методы. Производительность сильно снижается, даже если объект, доступ к которому осуществляется, находится в локальном домене приложения.

Наконец, с точки зрения удобства использования в производном от *MarshalByRefObject* типе не следует определять какие-либо статические члены по той простой причине, что к статическим членам всегда обращаются в контексте обращающегося домена приложения. Никакой переход между доменами невозможен, потому что информация о целевом домене содержится в объекте-прокси, но такой объект попросту отсутствует при вызове статического члена. Модель программирования, предусматривающая выполнение статических членов типа в одном AppDomain, в то время как экземплярные члены выполняются в другом, была бы очень неудачной.

Пример 2: доступ к объектам другого домена AppDomain с использованием маршалинга по значению

Этот пример похож на предыдущий. Точно так же создается второй домен AppDomain. Затем вызывается *CreateInstanceAndUnwrap* для загрузки той же сборки в новый AppDomain и создания в нем экземпляра типа. На сей раз создается экземпляр типа *MarshalByValType*, который не наследует типу *System.MarshalByRefObject*. Как и раньше, *CreateInstanceAndUnwrap* должен обеспечить изоляцию и поэтому не может просто вернуть ссылку на объект основному домену приложения. Так как *MarshalByValType* не является производным от *System.MarshalByRefObject, CreateInstanceAndUnwrap* может определить тип-прокси и создать его экземпляр; нельзя выполнять маршалинг объекта по ссылке.

Однако, так как *MarshalByValType* отмечен нестандартным атрибутом *[Serializable]*, метод *CreateInstanceAndUnwrap* может выполнять маршалинг по значению. Далее объясняется, что означает маршалинг объекта по значению из одного AppDomain (домен-источник) в другой AppDomain (целевой домен).

Когда домену-источнику нужно передать или возвратить ссылку на объект в целевом домене AppDomain, CLR сериализует экземплярные поля объекта в байтовый массив, который затем копируется из домена-источника в целевой домен. Затем 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 AppDomain. Затем *CreateInstanceAndUnwrap* возвращает ссылку на эту копию; объект переправлен по методу маршалинга через границу между доменами.

Внимание! При загрузке сборки CLR использует политики и конфигурацию целевого AppDomain (в частности, у домена приложения может быть другой каталог AppBase или информация о перенаправлении версий). Эти различия политик могут не позволить CLR определить местонахождение сборки. Если сборку загрузить не удастся, будет сгенерировано исключение, и целевой домен приложения не получит ссылку на объект.

На этом этапе объекты в домене-источнике и целевом AppDomain существуют независимо друг от друга, и их состояния также могут меняться независимо. В домене-источнике нет корней, предотвращающих исходный объект от уничтожения сборщиком мусора (как в приложении *AppDomainMarshalling*), его память будет освобождена при следующей сборке.

Чтобы убедиться, что объект, возвращенный методом CreateInstanceAndUnwrap, не является ссылкой на объект-прокси, приложение AppDomainMarshalling вызывает открытый статический метод *IsTransparentProxy* типа *System.Runtime.Remoting.RemotingService*, передав ему в качестве параметра ссылку, возвращенную методом *CreateInstanceAndUnwrap*. *IsTransparentProxy* возвращает значение false, означающее, что объект является реальным, не прокси.

Теперь программа использует реальный объект для вызова метода *SomeMethod*. Так как переменная *mbvt* ссылается на реальный объект, вызывается реальная реализация этого метода и не происходит никакого перехода между доменами приложения. Это легко проверить, проанализировав выходные данные: вызов *Thread.GetDomain().FriendlyName* возвращает «AppDomainMarshalling». Кроме того, в окне Call Stack отладчика не появляется строка [AppDomain Transition].

Чтобы предоставить дополнительное доказательство того, что прокси не задействуется, приложение AppDomainMarshalling выгружает новый AppDomain, после чего пытается снова вызвать метод *SomeMethod*. В отличие от примера 1, на сей раз запрос успешно выполняется, потому что выгрузка нового AppDomain никак не повлияла на объекты, «принадлежащие» основному домену приложения, в том числе на объект, переправленный по методу маршалинга по значению.

А сейчас у вас может возникнуть законный вопрос: а зачем вообще нужно создавать и выполнять маршалинг объекта в другой домен, используя семантику маршалинга по значению. В моем сценарии конструктор *MarshalByValType* фактически работает в новом AppDomain, то есть в рамках разрешений и конфигурации этого домена, а это полезно в некоторых ситуациях. Но сразу после маршалинга объекта через границу между доменами, все последующие его вызовы происходят в рамках разрешений и параметров конфигурации основного AppDomain. В разделе «Маршалинг аргументов и возвращаемых значений через границы между доменами» я расскажу об основной причине, по которой выгодно выполнять маршалинг объектов по значению между доменами приложения. По существу, большинство типов определено так, чтобы обеспечить возможность маршалинга через границы между доменами AppDomain.

Пример 3: обмен между доменам приложения с использованием не поддерживающих маршалинг типов

Пример 3 очень похож на описанные ранее примеры 1 и 2. Точно так же создается новый домен AppDomain, после чего вызывается *CreateInstanceAndUnwrap* для загрузки той же сборки в новый домен приложения и создания в нем экземпляра типа. На сей раз создается экземпляр типа *NonMarshalableType*. Как прежде, *CreateInstanceAndUnwrap* должен обеспечить изоляцию и не может просто вернуть ссылку на объект основному домену. Так как *NonMarshalableType* не наследует классу *System.MarshalByRefObject* и не отмечен нестандартным атрибутом *[Serializable]*, метод *CreateInstanceAndUnwrap* не может переслать объект по методу маршалинга — ни по ссылке, ни по значению — маршалинг объекта через границы между доменами вообще невозможен! Чтобы сообщить об этом вызывающему коду, *CreateInstanceAndUnwrap* генерирует исключение *SerializationException* в основном домене AppDomain. Так как в нашей программе не предусмотрен перехват этого исключения, программа просто тихо «умирает».

Маршалинг аргументов и возвращаемых значений через границы между доменами

В программе-примере метод *SomeMethod* принимает параметр *String*. Поэтому в примере 1 строка *String*, переданная методу *SomeMethod* прокси, должна в конечном счете попасть в метод *SomeMethod* реального объекта. Но, если CLR просто передаст ссылку на объект из одного домена приложения другому, изоляция бу-

дет нарушена. Поэтому, собираясь переправить параметр в другой домен, CLR должна проверить, является ли он производным от *MarshalByRefObject*; при положительном результате проверки выполняется маршалинг объекта по ссылке. Если нет, CLR проверяет, отмечен ли тип нестандартным атрибутом *[Serializable]*. Если да, выполняется маршалинг объекта по значению. Если тип объекта не является производным от *MarshalByRefObject* и не отмечен нестандартным атрибутом *[Serializable]*, CLR вообще не в состоянии выполнить маршалинг такого объекта и генерирует исключение.

Если метод принимает более одного параметра, CLR должна выполнить полный маршалинг для каждого параметра. Точно так же, когда метод пытается возвратить объект в домен-источник, CLR должна выполнить его маршалинг по ссылке или по значению или сгенерировать исключение. Кстати, если объект относится к сложному типу, состоящему из нескольких полей, CLR должна пройти весь граф объекта и выполнить маршалинг всех полей по ссылке или по значению — в зависимости от типа конкретного поля.

Примечание В приложении AppDomainMarshalling я передаю *String* в другой домен приложения. Так как тип *System.String* не является производным от *MarshalByRefObject*, нельзя выполнять его маршалинг по ссылке. К счастью, *System.String* отмечен атрибутом *[Serializable]*, поэтому может передаваться по методу маршалинга по значению. CLR выполняет специальную оптимизацию объектов *String*. При маршалинге объекта *String* в другой домен AppDomain среда CLR просто передает ссылку на объект *String* через границу, не создавая его копию. Такая оптимизация возможна из-за того, что объекты *String* неизменны, поэтому код одного домена не может «испортить» символы объекта *String* в другом. Подробнее о неизменности строк см. главу 11.

Выгрузка доменов AppDomain

Одна из замечательных особенностей CLR — возможность выгрузки доменов приложений. При выгрузке AppDomain выгружаются и все загруженные в него сборки, а CLR также освобождает кучу загрузчика домена. Выгрузить AppDomain легко: нужно просто вызывать статический метод *Unload* класса *AppDomain* (как показано в приложении AppDomainMarshalling). Этот вызов заставляет CLR выполнить массу операций по корректной выгрузке указанного AppDomain.

1. CLR приостанавливает все потоки в процессе, которые когда-либо выполняли управляемый код.
2. CLR проверяет стеки всех потоков на наличие потоков, которые в текущий момент выполняют код выгружаемого AppDomain или которые могут рано или поздно вернуться к выполнению кода выгружаемого AppDomain. CLR вынуждает все потоки, в стеке которых находится выгружаемый AppDomain, сгенерировать исключение *ThreadAbortException* (при этом выполнение потока возобновляется). Это заставляет потоки выполнить все блоки *finally*, то есть корректно завершить свою работу. Если не обнаружится кода, перехватывающего *ThreadAbortException*, такое ставшее необотанным исключение CLR «проглаты-

вает»; поток завершается, но процессу разрешено 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. Если бы CLR уничтожилала такие потоки, невозможно было бы выполнить код очистки, восстановления после ошибок, инициализации типа, критический код или любой код, который CLR не знает, как обрабатывать, а это приводило бы к непредсказуемому поведению приложения и появлению брешей в защите. Уничтожаемому потоку разрешается закончить выполнение таких блоков, и только после этого CLR вынуждает поток сгенерировать исключение *ThreadAbortException*.

3. После выгрузки из AppDomain всех потоков, обнаруженных в п. 2 CLR выполняет проход по куче и устанавливает флаг в каждом объекте-прокси, который ссылается на объект, созданный в выгружаемом AppDomain. Так объекты-прокси «узнают», что реальный объект, на который они ссылаются, уничтожен. Теперь, если вызвать метод недействительного объекта-прокси, этот метод сгенерирует исключение *AppDomainUnloadedException*.
4. CLR инициирует принудительную сборку мусора, чтобы освободить память, занятую объектами выгружаемого AppDomain. Вызываются методы *Finalize* этих объектов, предоставляя им возможность выполнить корректную очистку.
5. CLR возобновляет работу всех оставшихся потоков. Поток, вызвавший *AppDomain.Unload*, продолжает работу; вызовы *AppDomain.Unload* выполняются синхронно.

В приложении AppDomainMarshalling всю работу выполняет один поток. Всякий раз, когда код приложения вызывает *AppDomain.Unload*, в выгружаемом домене AppDomain нет потоков, поэтому CLR не приходится генерировать исключение *ThreadAbortException* (подробнее о *ThreadAbortException* я расскажу чуть позже).

Кстати, когда поток вызывает *AppDomain.Unload*, CLR ждет 10 секунд (по умолчанию), чтобы потоки выгружаемого AppDomain могли покинуть его. Если по истечении 10 секунд, поток, вызвавший *AppDomain.Unload*, не возвращает управление, он генерирует исключение *CannotUnloadAppDomainException*, и AppDomain может быть (или нет) выгруженным в будущем.

Примечание Если поток, вызвавший *AppDomain.Unload*, находится в выгружаемом AppDomain, CLR создаст другой поток, который попытается выгрузить AppDomain. Первый поток принудительно сгенерирует *ThreadAbortException* и выполнит раскрутку — поиск и очистку всех операций, начатых ниже по стеку вызовов. Новый поток дождется выгрузки AppDomain, а затем завершится. В случае сбоя выгрузки AppDomain новый поток обработает исключение *CannotUnloadAppDomainException*, но так как нет кода, выполняемого этим новым потоком, перехватить это исключение нельзя.

Как хосты используют домены AppDomain

Я уже рассказывал о хостах: и как они загружают CLR, и как хост может заставить CLR создать или выгрузить AppDomain. Теперь, чтобы перевести разговор в более конкретное русло, я опишу несколько обычных сценариев с хостингом CLR и доменами приложений. В частности, я расскажу, как разные типы приложений осуществляют хостинг CLR и управляют доменами приложения.

Консольные приложения и приложения Windows Forms

Вызывая управляемое консольное приложение или приложение Windows Forms, согласователь (shim) анализирует сведения заголовка CLR из сборки приложения. Заголовок содержит сведения о версии CLR, с которой было создано и протестировано приложение. На основе этих данных согласователь определяет нужную версию CLR. После загрузки и инициализации CLR еще раз анализирует CLR-заголовок сборки, чтобы определить метод, являющийся точкой входа приложения (*Main*). После того как CLR вызвала его, можно считать, что приложение уже работает.

По мере выполнения код обращается к новым типам. При ссылке на тип, расположенный в другой сборке, CLR находит нужную сборку и загружает ее в тот же AppDomain, в него загружаются и другие сборки, на которые код будет ссылаться в ходе исполнения. Когда метод *Main* возвращает управление, процесс Windows завершается (что вызывает уничтожение основного и остальных его доменов).

Примечание Кстати, чтобы завершить процесс Windows и выгрузить все домены приложения, можно вызвать статический метод *Exit* типа *SystemEnvironment*. Вызов *Exit* — наиболее корректный метод завершения процесса, так как он сначала вызывает методы *Finalize* для всех объектов в управляемой куче, затем освобождает все неуправляемые COM-объекты, удерживаемые CLR, и в завершение вызывает Win32-функцию *ExitProcess*.

Консольное приложение или приложение Windows Forms может заставить CLR создать дополнительные домены приложения в адресном пространстве процесса, как показано в приложении AppDomainMarshalling.

Microsoft Internet Explorer

При установке .NET Framework устанавливает MIME-фильтр (MSCorIE.dll), который подключается к Internet Explorer версии 5.01 и выше. Этот фильтр обрабатывает загружаемое содержимое, помеченное типом MIME «application/octet-stream» и «application/x-msdownload». Обнаружив, что загружается управляемая сборка, MIME-фильтр вызывает функцию *CorBindToRuntimeEx* для загрузки CLR, и процесс Internet Explorer становится хостом CLR.

Фильтр MIME контролирует CLR и следит, чтобы все сборки с одного Web-узла загружались в один AppDomain. Поскольку у каждого домена приложения собственный контекст безопасности администратор может по-разному обращаться со сборками, доверяя сборкам, загруженным с одних Web-сайтов, и не доверяя другим, а также выгружать сборки ненужного Web-сайта, когда пользователь переходит к другому Web-сайту.

Web-формы ASP.NET и Web-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 Web-приложение, ASP.NET определяет, были ли уже такие запросы. Если нет, ASP.NET приказывает CLR создать новый AppDomain для данного Web-приложения (каждое Web-приложение идентифицируется собственным виртуальным корневым каталогом). Далее ASP.NET приказывает CLR загрузить в новый AppDomain сборку с типом, поддерживаемым этим Web-приложением, создает экземпляр этого типа и начинает вызывать его методы для исполнения запроса клиента. Если код ссылается на другие типы, CLR загружает в AppDomain Web-приложения дополнительные сборки.

Если клиент запрашивает уже работающее Web-приложение, ASP.NET создает не новый AppDomain, а новый экземпляр типа Web-приложения в существующем AppDomain и начинает вызывать его методы. При этом вызываемые методы уже скомпилированы JIT-компилятором в машинный код, поэтому следующие клиентские запросы обрабатываются намного быстрее.

Если клиент запрашивает другое Web-приложение, ASP.NET заставляет CLR создать новый AppDomain. Обычно он создает его в том же рабочем процессе, где находятся другие домены приложения. Это значит, что в одном процессе Windows может работать несколько Web-приложений, что повышает производительность системы. В этом случае сборки, нужные разным Web-приложениям, загружаются в собственный AppDomain каждого Web-приложения — это необходимо для изоляции кода и объектов Web-приложения от других Web-приложений.

Замечательная особенность ASP.NET — возможность изменять код Web-сайта на лету, без остановки Web-сервера. Когда файл на жестком диске на Web-сайте меняется, ASP.NET обнаруживает это, выгружает AppDomain, содержащий старую версию файлов (после завершения текущего запроса), а затем создает новый AppDomain и загружает в него новые версии файлов. При этом ASP.NET использует особенность AppDomain, называемую *теневым копированием* (shadow copying).

Microsoft SQL Server 2005

Microsoft SQL Server 2005 — неуправляемое приложение, так как большая часть его кода написана на неуправляемом C++. SQL Server 2005 поддерживает создание хранимых процедур на управляемом коде. При первом получении запроса на выполнение хранимой процедуры на управляемом коде SQL Server загружает CLR. Хранимые процедуры выполняются в собственном защищенном домене AppDomain, что не позволяет им нарушить работу сервера базы данных.

Это исключительно замечательная функциональность! Это означает, что разработчики могут выбирать язык программирования для создания хранимых процедур. Кроме того, код JIT компилируется в машинный код и выполняется, а не интепретируется. Также разработчикам таких процедур доступны все типы библиотеки FCL или любой другой сборки. Результат — разработка хранимых процедур значительно упрощается, а приложения работают намного быстрее. Что еще 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 позволят получить доступ к любым сборкам и типам, поддерживающим CLR. Они будут компилироваться, поэтому выполняться быстро, и, что самое важное, макросы будут выполняться в защищенном домене AppDomain, что избавит пользователей от многих неприятных неожиданностей.

Другие методы управления хостом

В этом разделе мы обсудим более продвинутые методы хостинга CLR. Я хочу раскрыть вам глаза на возможности хостинга и помочь осознать всю широту возможностей CLR. Если тема этого раздела покажется вам интересной, настоятельно рекомендую обратиться к другой литературе по теме (особенно к книге Стивена Претчнера «Customizing the Microsoft .NET Framework Common Language Runtime», которая упоминалась в начале главы).

Управление CLR с помощью управляемого кода

Класс *SystemAppDomainManager* позволяет хосту менять поведение CLR по умолчанию, используя управляемый, а не неуправляемый код. Ясно, что использование управляемого кода упрощает реализацию хоста. Все, что требуется, — определить собственный класс, потомок класса *SystemAppDomainManager*, переопределив все необходимые виртуальные методы. Далее этот класс надо скомпоновать в отдельную сборку и установить в GAC, потому что сборке нужно предоставить полное доверие (все сборки в GAC получают полное доверие).

Затем при запуске процесса Windows нужно будет заставить CLR использовать свой производный от *AppDomainManager* класс. В процессе может быть один и только один класс, производный от *AppDomainManager*. И в каждом AppDomain, созданном в процессе, может быть только один экземпляр этого класса. Есть три способа связать производный от *AppDomainManager* класс с процессом Windows:

- **Использование неуправляемого API хостинга (оптимальный вариант)** В этом случае хост должен содержать немного неуправляемого кода. После вызова *CorBindToRuntimeEx* хост должен запросить интерфейс *ICLRControl* вызвать его функцию *SetAppDomainManagerType*, передав идентификационную информацию сборки в GAC и имя класса, производного от *AppDomainManager*.
- **Переменные окружения (второй по приоритетности вариант)** Перед запуском CLR процесс Windows должен установить две переменные окружения: APPDOMAIN_MANAGER_ASM должна указывать на сборку, установленную в GAC, а APPDOMAIN_MANAGER_TYPE должна содержать имя класса, производного от *AppDomainManager*. Часто для определения этих переменных используется программа-заглушка, которая затем запускает управляемый исполняемый файл — это обеспечивает управляемость всего кода; к неуправляемому коду прибегать не приходится.
- **Параметры реестра (наименее предпочтительный вариант)** Перед запуском CLR в раздел реестра *HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework* (или *HKEY_CURRENT_USER\Software\Microsoft\.NETFramework*) нужно доба-

вить два параметра: APPDOMAIN_MANAGER_ASM, указывающий на сборку, установленную в GAC, и APPDOMAIN_MANAGER_TYPE, содержащий имя класса, производного от *AppDomainManager*. Это самый неудачный вариант, так как затрагивает все управляемые приложения, выполняющиеся на машине, — они будут использовать один производный от *AppDomainManager* класс. Это нелепо. Этот способ пригоден только для проверки работы приложения на собственной машине (предполагается, что никаких других управляемых приложений работать не должно). Видите, почему это нелепо?

А теперь поговорим о роли класса, производного от *AppDomainManager*. Его задача — сохранить за хостом контроль, даже когда код подключаемого компонента пытается создать собственный AppDomain. Когда код в процессе попытается создать новый AppDomain, объект производного от *AppDomainManager* класса в этом домене может изменить параметры конфигурации и безопасности. Он может также отказаться от создания AppDomain или вместо этого возвратить ссылку на существующий AppDomain. Создавая новый AppDomain, CLR создает в этом домене новый объект класса, производного от *AppDomainManager*. Этот объект также может изменять параметры конфигурации, определять порядок перехода контекста выполнения между потоками и разрешения, предоставленные сборке.

Создание надежного приложения-хоста

Хост может указать CLR, какие действия предпринимать в случае сбоя в управляемом коде. Вот несколько примеров (перечисленные в порядке от наименее до наиболее серьезного):

- CLR может прервать поток, если тот выполняется слишком долго или на протяжении большого времени не возвращает управление. (Подробнее об этом см. следующий раздел.)
- CLR может выгрузить AppDomain. При этом будут закрыты все потоки домена AppDomain, а проблематичный код будет выгружен.
- CLR может быть отключена. При этом прекращается выполнение любого кода в процессе, но неуправляемому коду работать разрешено.
- CLR может выйти из процесса Windows. При этом закрываются все потоки и выгружаются все домены AppDomains — выполняются операции очистки, после чего процесс закрывается.

CLR может завершить поток или AppDomain корректно или принудительно. Корректное завершение предусматривает выполнение кода очистки. Иначе говоря, выполняется код в блоках *finally* и методы *Finalize* объектов. При принудительном завершении код очистки не выполняется, то есть код в блоках *finally* и методы *Finalize* объектов могут быть не выполненными. При корректном завершении, в отличие от принудительного завершения, не удастся закрыть поток, который находится в блоке *catch* или *finally*. К сожалению, поток, выполняющий неуправляемый код или находящийся в критической области (Critical Execution Region, CER), завершить вообще нельзя.

Хост может установить то, что называют *политикой эскалации* (escalation policy), которая инструктирует CLR, как вести себя со сбоями управляемого кода. Например, SQL Server 2005 определяет, что должна делать CLR в случае возникно-

вения необработанного исключения во время выполнения средой CLR управляемого кода. Когда в потоке возникает необработанное исключение, CLR сначала пытается превратить его в корректное завершение работы потока. Если поток не закрывается за определенное время, CLR пытается перейти от корректного к принудительному завершению потока.

В большинстве случаев происходит именно так, как описано выше. Однако, если поток, в котором возникло необработанное исключение, находится в критической области, действует другая политика. Поток в критической области заблокирован механизмом синхронизации потоков, и блокировку должен снять тот же поток, например тот, что вызвал *Monitor.Enter*, метод *WaitOne* типа *Mutex* или один из методов — *AcquireReaderLock* или *AcquireWriterLock* — типа *ReaderWriterLock*. Ожидание *AutoResetEvent*, *ManualResetEvent* или *Semaphore* не означает, что поток находится в критической области, потому что другой поток может освободить этот объект синхронизации. Когда поток находится в критической области, CLR полагает, что поток работает с данными, совместно используемыми многими потоками того же AppDomain. В конце концов, это наиболее вероятная причина блокировки потока. Если поток работает с общими данными, простое завершение потока — неудачное решение, потому что другие потоки могут обратиться к общим данным, которые окажутся разрушенными, из-за чего AppDomain может вести себя непредсказуемо; также по этой причине возможно появление брешей в защите.

Поэтому, когда в потоке в критической области возникает необработанное исключение, CLR сначала пытается свести исключение к корректной выгрузке AppDomain в попытке избавиться от всех используемых потоков и объектов данных. Если AppDomain не выгружается за указанное время, CLR переходит от корректной к принудительной выгрузке AppDomain.

Как поток возвращается в хост

Обычно приложению-хосту нужно сохранять контроль над собственными потоками. В качестве примера возьмем сервер базы данных. При поступлении на сервер запроса поток принимает его и пересылает другому потоку, который и должен выполнить работу. Может случиться, что второму потоку придется выполнить код, не созданный и протестированный командой разработчиков сервера БД. Например, хранимую процедуру, созданную на управляемом коде программистами компании, использующей сервер. Это замечательно, что сервер базы данных может выполнять код хранимых процедур в собственном AppDomain, который работает с максимальными ограничениями безопасности. Это не позволяет хранимой процедуре обращаться к каким-либо объектам за пределами собственного AppDomain, а также получать доступ к ресурсам, к которым обращаться коду запрещено, например к дисковым файлам или буферу обмена.

Но что, если код хранимой процедуры застрянет в бесконечном цикле? То есть сервер базы данных «отдал» один из своих потоков для выполнения кода хранимой процедуры, но поток никогда не вернет управление. Сервер оказывается в неудобном положении, и его будущее поведение становится непредсказуемым. Например, может сильно упасть производительность из-за того, что поток выполняет бесконечный цикл. Может, серверу стоит создать больше потоков? Но тогда возрастет загрузка ресурсов (таких как место в стеке), да 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. 21-3 показана типичная архитектура хоста, пытающегося «приструнить» вышедший из-под контроля поток. Вот как это происходит (номера операций соответствуют числам в кружках 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, созданный в компании, в которой создано и протестировано приложение-хост.
4. Этот доверенный код входит в блок *try* и из него вызывает другой домен AppDomain (используя тип, производный от *MarshalByRefObject*). Этот AppDomain содержит недоверенный код (например, хранимую процедуру), который не создан и протестирован в компании, где разработано приложение-хост. На этом этапе сервер передал управление потоком некоторому недоверенному коду; теперь сервер начинает «нервничать».
5. Хост фиксирует время получения исходного клиентского запроса. Если недоверенный код не отвечает клиенту за определенное администратором время, хост вызывает метод *Abort* типа *Thread*, требуя от CLR остановить поток из пула и вынуждая сгенерировать исключение *ThreadAbortException*.
6. На этом этапе поток пула начинает завершение, вызывая блоки *finally*, чтобы выполнить код очистки. В конечном счете поток возвращается в домен AppDomain. Так как программа-заглушка вызвала недоверенный код из блока *try*, в ней есть блок *catch*, который перехватывает *ThreadAbortException*.
7. В ответ на захват ThreadAbortException хост вызывает метод *ResetAbort* типа *Thread*. Зачем это нужно, я объясню чуть позже.
8. Теперь, после перехвата *ThreadAbortException*, хост может вернуть информацию о сбое клиенту и вернуть поток в пул, чтобы его можно было снова задействовать для обслуживания клиентских запросов.

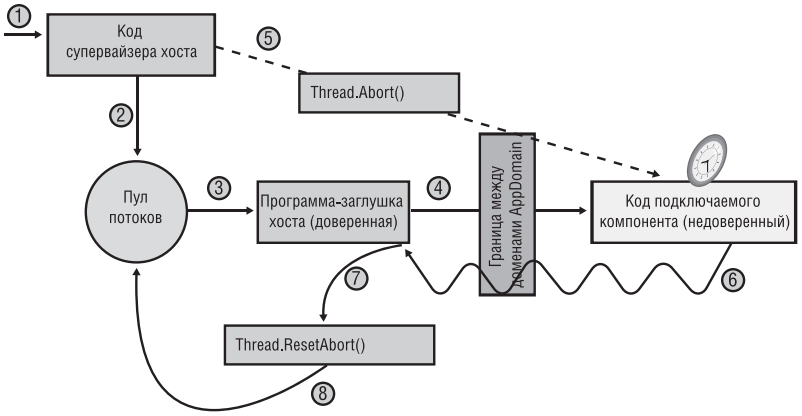


Рис. 21-3. *Возвращение хостом контроля над потоком*

А сейчас я проясню некоторые непонятные места этой архитектуры. Во-первых, метод *Abort* типа *Thread* выполняется асинхронно. *Abort* отмечает целевой

поток флагом *AbortRequested* и немедленно возвращает управление. Когда исполняемая среда обнаруживает, что поток завершается, она пытается перенести поток в *безопасное место* (safe place). Исполняющая среда считает, что поток находится в безопасном месте, если его (по ее мнению) можно остановить без риска серьезных последствий. Поток находится в безопасном месте, если выполняет управляемую операцию блокировки, например бездействует или «спит». Поток может перемещаться в безопасное место, используя перехват (см. главу 20). 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 окажется в безопасном месте, исполняющая среда обнаружит флаг *AbortRequested* потока и заставит его сгенерировать исключение *ThreadAbortException*. Если исключение не будет перехвачено, оно останется необработанным, будут выполнены все блоки *finally* и поток корректно завершит работу. В отличие от других исключений, необработанное *ThreadAbortException* не приводит к остановке приложения. Исполняющая среда «проглатывает» это исключение, и поток завершается, но приложение и все оставшиеся его потоки продолжают работу.

В нашем примере хост перехватывает *ThreadAbortException*, получая возможность вернуть контроль над потоком и возвратить его в пул. Но есть одна неувязка: что может запретить недоверенному коду перехватить *ThreadAbortException*, чтобы сохранить за собой контроль над потоком? Ответ — особое отношение CLR к исключению *ThreadAbortException*. Даже если код перехватит *ThreadAbortException*, CLR не позволит проигнорировать исключение. Иначе говоря, в конце блока *catch* CLR автоматически повторно сгенерирует *ThreadAbortException*.

В связи с этой особенностью CLR возникает другой вопрос: если CLR повторно генерирует *ThreadAbortException* в конце блока *catch*, как же хосту удается перехватить это исключение, чтобы восстановить контроль над потоком? В блоке *catch* хоста есть вызов метода *ResetAbort* типа *Thread*. Вызов этого метода запрещает CLR повторно генерировать *ThreadAbortException* в конце каждого блока *catch*.

И снова возникает законный вопрос: а что может запретить недоверенному коду перехватить *ThreadAbortException* и самому вызвать метод *ResetAbort* типа *Thread*? А вот что: для вызова метода *ResetAbort* у вызывающей программы должно быть разрешение *SecurityPermission* с флагом *ControlThread* равным true. Создавая AppDomain для недоверенного кода, хост не предоставляет такое разрешение, поэтому недоверенный код не сможет сохранить за собой контроль над потоком хоста.

Должен заметить, что брешь во всем этом все-таки возможна: когда поток выполняет раскрутку исключения *ThreadAbortException*, недоверенный код может выполнить блоки *catch* и *finally*, в которых может оказаться код с бесконечным циклом, не позволяющим хосту вернуть контроль над потоком. Эту проблему хост решает определением обсуждавшейся ранее политики эскалации. Если останавливаемый поток не завершается за разумное время, CLR может перейти от корректной к принудительной остановке потока, принудительной выгрузке AppDomain, отключению CLR или уничтожению процесса. Замечу также, что недоверенный код может перехватить *ThreadAbortException* и в блоке *catch* сгенерировать какое-то другое исключение. Если это исключение перехватывается, в конце блока *catch* CLR автоматически повторно генерирует *ThreadAbortException*.

Вместе с тем нужно сказать, что большинство недоверенных программ не представляет угрозы — просто с точки зрения хоста они тратят на выполнение своей задачи слишком много времени. Обычно блоки *catch* и *finally* содержат очень немного кода, который выполняется быстро, без каких-либо бесконечных циклов или «долгоиграющих» операций. Поэтому вероятность срабатывания политики эскалации для возвращения управления потоком хосту довольно мала.

Кстати, у класса *Thread* два метода *Abort*: один не имеет параметров, а второй принимает параметр *Object*, в котором можно передать любой объект. Перехватив *ThreadAbortException*, код может запросить свое неизменяемое свойство *ExceptionState*, которое вернет объект, переданный в качестве параметра. Это позволяет потоку, вызвавшему *Abort*, передать дополнительную информацию коду, перехватившему *ThreadAbortException*. Хост может использовать это для информирования собственного кода обработки о причине остановки потоков.

ГЛАВА 22

Загрузка сборок и отражение

Эта глава рассказывает о возможности обнаружения информации о типах, создания их экземпляров и обращения к их членам несмотря на тот факт, 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 приложение-хост, а другие создают *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, после выпуска приложения-хоста. Вот почему хосту приходится самостоятельно находить подключаемые компоненты во время выполнения.

Динамически расширяемое приложение может использовать хостинг CLR и домены AppDomain, как описано в главе 21. Хост выполняет код подключаемых компонентов в отдельных доменах AppDomain с собственными параметрами безопасности и конфигурации. Хост также может выгрузить подключаемый компонент, выгрузив AppDomain, в котором он выполняется. В конце главы я задействую все эти возможности — хостинг CLR, домены AppDomain, 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 видит, 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 AppDomain (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.ReflectionAssembly*. Этот метод описан в открытой документации, и его можно вызывать для явной загрузки сборки в свой AppDomain. Он представляет собой CLR-эквивалент Win32-функции *LoadLibrary*. В сущности, есть несколько перезагруженных версий метода *Load* класса *Assembly*. Вот прототипы наиболее популярных перегруженных версий:

```
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 GAC, а затем последовательно в базовом каталоге приложения, каталогах закрытых путей и каталоге, указанном в элементе *codeBase* конфигурационного файла. Если *Load* передается сборка с нестрогим именем, он не применяет к ней политику, и CLR не будет искать ее в GAC. Найдя искомую сборку, *Load* возвращает ссылку на объект *Assembly*, представляющий загруженную сборку. Если указанная сборка не найдена, генерируется исключение *System.IO.FileNotFoundException*.

Примечание В небольшом количестве чрезвычайно редких ситуаций может потребоваться загрузить сборку, скомпонованную 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 x86-версия сборки, CLR предпочтет специализированную версию сборки (см. главу 3). Однако можно вынудить 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 загрузить сборку в некоторый AppDomain. Этот метод создан для неуправляемого кода и позволяет хосту загрузить сборку в определенный домен приложения. Разработчикам управляемого кода лучше его избегать, и вот почему.

При вызове методу *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 AppDomain связаны параметры, определяющие правила поиска сборки для CLR. Так вот, при загруз-

ке сборки CLR будет руководствоваться параметрами заданного AppDomain, а не вызывающего.

Так или иначе, метод *Load* объекта *AppDomain* возвращает ссылку на сборку. В силу того, что класс *SystemAssembly* не является потомком *System.MarshalByRefObject*, объект сборки возвращается вызывающему AppDomain маршалингом по значению. Но теперь для поиска и загрузки сборки CLR будут применяться параметры вызывающего AppDomain. Если сборку не удается найти при помощи политики вызывающего AppDomain или в заданных им каталогах поиска, генерируется исключение *FileNotFoundException*. Такая ситуация обычно нежелательна, поэтому следует избегать метода *Load* объекта *SystemAppDomain*.

В большинстве динамически расширяемых приложений метод *Load* объекта *AppDomain* является предпочтительным способом загрузки сборки в AppDomain, но он требует наличия всех частей, идентифицирующих сборку. Часто разработчики создают средства 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, определяющий путь (с расширением) файла 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 *SystemReflectionAssemblyName* (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 *Load* не удается найти сборку, *LoadFrom* загружает сборку, указанную в пути, переданном 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 URL в качестве параметра:

```
private static void SomeMethod() {
   Assembly a = Assembly.LoadFrom(@"http://Wintellect.com/SomeAssembly.dll");
}
```

При получении Интернет-адреса CLR загружает файл, устанавливает его в загрузочный кеш пользователя и уже из него загружает файл. Система должна быть подключена к Интернету, иначе возникнет исключение. Однако, если файл уже загружен в кеш ранее и Internet Explorer настроен на работу в автономном режиме [см. команду Work Offline (Работать автономно) в меню File (Файл)], будет использоваться файл из кеша и исключение не возникнет.

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, *LoadFrom* не принесет неожиданностей. 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* загрузит указанный файл, не получая информацию строгого имени сборки и не выполняя поиск файла в GAC или где-либо еще. Метод *ReflectionOnlyLoad* выполнит 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 в точности та сборка, что ожидалось. Если вы хотите самостоятельно применить политику версий к сборке, можно передать строку с идентификационной информацией в метод *AppDomain* класса *ApplyPolicy*.

При загрузке сборок методами *ReflectionOnlyLoadFrom* или *ReflectionOnlyLoad* среда CLR запрещает выполнение какого-либо кода сборки, а при попытке выполнить код генерирует исключение *InvalidOperationException*. Эти методы позволяют инструменту загружать сборки с отложенным подписанием, 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, указываемые 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 такая ситуация: поток возвращается из метода в код выгруженной сборки и приложение терпит сбой. CLR стоит на страже надежности и безопасности, а подобные сбои непроизводительны и не соответствуют ее целям. Если надо выгрузить сборку, придется выгрузить весь AppDomain, в котором она находится. Подробнее см. главу 21.

Казалось бы, сборки, загруженные методом *ReflectionOnlyLoadFrom* или *ReflectionOnlyLoad*, могут выгружаться, ведь код этих сборок нельзя выполнять. Однако CLR не разрешает выгрузку сборок, загруженных одним из этих методов, по той простой причине, что после такой загрузки сборки отражение может повторно использоваться для создания объектов, ссылающихся на метаданные, определенные в этих сборках. При выгрузке сборки потребовалось бы как-то делать объекты недействительными, но отслеживание всех этих связей — слишком сложная и ресурсоемкая задача.

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 для 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* позволяют запрашивать поля, методы, свойства и события типа путем синтаксического разбора соответствующих таблиц метаданных. Можно узнать, какими атрибутами (о них см. главу 17) помечена та или иная сущность метаданных. Есть даже классы, позволяющие определить указанные сборки и методы, которые возвращают в методе байтовый IL-поток. Имея эти данные, можно создать инструмент вроде ILDasm.exe от 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 «не для всех», то документация выглядит менее запутанной.

В реальности приложениям редко требуются типы отражения. Обычно отражение используется в библиотеках классов, которым нужно понять определение типа, чтобы дополнить его. Так, механизм сериализации из FCL применяет отражение, чтобы выяснить, какие поля определены в типе. Объект форматирования из механизма сериализации получает значения этих полей и записывает их в поток байт для пересылки через Интернет. Аналогично создатели Microsoft Visual Studio используют отражение, чтобы определить, какие свойства показывать разработчикам при размещении элементов на поверхности Web-формы или формы Windows Forms во время ее создания.

Отражение также используется, 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 заданного 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). *Раннее связывание* (early binding) имеет место, когда используемые приложением 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.

- **Отражение не позволяет использовать безопасность во время компиляции** Так как в отражении активно используются строки, теряется безопасность типов во время компиляции. Например, если выполнить оператор *Type.GetType("Jef");*, чтобы задействовать отражения для нахождения типа по имени *Jef* в сборке, в которой есть тип *Jeff*, то такой код скомпилируется, но во время выполнения возникнет ошибка, так как в имени типа, переданного в качестве параметра, опечатка.
- **Отражение работает медленно** При использовании отражения имена типов и их члены не известны на момент компиляции — они определяются в процессе выполнения, причем все типы и члены идентифицируются по строковому имени. Это значит, что при отражении постоянно выполняется поиск строк в процессе просмотра метаданных сборки в пространстве 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 при помощи 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. Также CLR должна проверить 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 ниже 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) 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), и вызывайте методы, определенные в интерфейсе. Я предпочитаю эту методику предыдущей, так как использование базового типа не позволяет разработчику выбрать базовый тип, оптимальный для конкретной ситуации.

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».

Обнаружение типов, определенных в сборке

Отражение часто используется для выяснения, какие типы определены в сборке. Для получения этой информации FCL предлагает много методов. Наиболее популярный — метод *GetExportedTypes* класса *Assembly*. Вот пример кода, который загружает сборку и отображает 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);
   }

   private static void LoadAssemAndShowPublicTypes(String assemId) {
      // Явно загружаем сборку в домен AppDomain.
      Assembly a = Assembly.Load(assemId);

      // Выполняем цикл для каждого открытого типа,
      // экспортируемого загруженной сборкой.
      foreach (Type t in a.GetExportedTypes()) {
         // Отображаем полное имя типа.
         Console.WriteLine(t.FullName);
      }
   }
}
```

Что же из себя представляет объект-тип

Заметьте: приведенный выше 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 *Type* может быть членом другого типа). FCL предоставляет несколько типов, производных от *System.Type*, — *System.RuntimeType*, *System.ReflectionOnlyType*, *System.Reflection.TypeDelegator* и некоторые типы, определенные в пространстве имен *System.Reflection.Emit* (*EnumBuilder*, *GenericTypeParameterBuilder* и *TypeBuilder*).

Примечание Класс *TypeDelegator* позволяет из кода динамически создавать подклассы класса *Type* при помощи инкапсуляции *Type*, которая позволяет переопределить некоторые методы и позволить *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 AppDomain к типу CLR создает экземпляр *RuntimeType* и инициализирует поля объекта информацией о типе.

Как вы помните, в *System.Object* определен открытый невиртуальный метод *GetType*. Если его вызвать, CLR определит тип указанного объекта и вернет ссылку на его *RuntimeType*. Поскольку для каждого типа в AppDomain 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 *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 удается, возвращается null или генерируется исключение *System.TypeLoadException* — все зависит от того, какая перегруженная версия метода *GetType* вызывалась и какие ей передавались параметры. Документация по FCL содержит полное 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 следующие экземплярные методы: *GetNestedType* и *GetNestedTypes*.
- Тип *System.Reflection.Assembly* предлагает следующие экземплярные методы: *GetType*, *GetTypes* и *GetExportedTypes*.
- В типе *System.Reflection.Module* есть следующие 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 оказываются 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».

Во многих языках программирования есть оператор, позволяющий получить объект *Type* по имени типа. Для получения ссылки на *Type* надо стараться использовать именно этот оператор, а не перечисленные выше методы, так как при компиляции оператора получается более быстрый код. В C# это оператор *typeof*, и обычно он используется для сравнения информации между типами, загруженными с использованием позднего и раннего (они известны во время компиляции) связывания. Вот пример:

```
private static void SomeMethod(Object o) {
   // GetType возвращается тип объекта во время выполнения (позднее связывание).
   // typeof возвращается тип указанного класса (раннее связывание).
   if (o.GetType() == typeof(FileInfo))       { ... }
   if (o.GetType() == typeof(DirectoryInfo))  { ... }
}
```

Примечание Первый оператор *if* проверяет, ссылается ли переменная *o* на объект типа *FileInfo*, но не тип, производный от *FileInfo*. Иначе говоря, код выше проверяет на точное, не совместимое соответствие. Совместимое соответствие обычно получают при использовании приведения или оператора *is* или *as* языка C#.

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 так далее, указывает на флаги, связанные с типом. Другие свойства, к ним относятся *Assembly*, *AssemblyQualifiedName*, *FullName*, *Module* и другие, 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 (см. исходный текст ниже) используются описанные выше концепции, чтобы загрузить в AppDomain определенное подмножество сборок и отобразить все типы, которые в конечном итоге наследуют типу *System.Exception*. Кстати, это программа, которую я написал, чтобы создать иерархию исключений, приведенную в главе 19.

```
using System;
using System.Text;
using System.Reflection;
using System.Collections.Generic;

public static class Program {
   public static void Main() {
      // Явно загружаем сборки, нужные для отражения.
      LoadAssemblies();

      // Инициализируем счетчики и список типов исключений.
      Int32 totalPublicTypes = 0, totalExceptionTypes = 0;
      List<String> exceptionTree = new List<String>();

      // Итеративно обрабатываем все сборки, загруженные в этот AppDomain.
      foreach (Assembly a in AppDomain.CurrentDomain.GetAssemblies()) {

         // Итеративно обрабатываем все типы, определенные в сборке.
         foreach (Type t in a.GetExportedTypes()) {
            totalPublicTypes++;

            // Игнорируем тип, если он не является открытым.
            if (!t.IsClass || !t.IsPublic) continue;

            // Создаем строку с родословной типа.
            StringBuilder typeHierarchy = new StringBuilder(t.FullName, 5000);

            // Предположим, что тип не является потомком Exception.
            Boolean derivedFromException = false;

            // Проверяем, не является ли System.Exception базовым для данного типа.
            Type baseType = t.BaseType;
            while ((baseType != null) && !derivedFromException) {
               // Добавляем базовый тип в конец строки.
               typeHierarchy.Append("-" + baseType);

               derivedFromException = (baseType == typeof(System.Exception));
               baseType = baseType.BaseType;
            }
```

```
            // После просмотра всех базовых типов оказалось, что объект
            // не является потомком Exception; перейти к следующему типу.
             if (!derivedFromException) continue;

             // Найден тип-потомок Exception.
             totalExceptionTypes++;

            // Меняем порядок типов в иерархии
            // этого типа-потомка Exception.
             String[] h = typeHierarchy.ToString().Split('-');
             Array.Reverse(h);

            // Создаем новую строку с иерархией, упорядоченной
            // в направлении от предка к потомку.
            // Добавляем эту строку к списку типов-потомков Exception.
             exceptionTree.Add(String.Join("-", h, 1, h.Length - 1));
         }
      }

      // Упорядочиваем список типов по старшинству.
      exceptionTree.Sort();

      // Отображаем дерево типов исключений.
      foreach (String s in exceptionTree) {
         // Разделяем базовые типы для этого типа исключений.
         String[] x = s.Split('-');

        // Сделать отступ в зависимости от числа базовых
        // типов и показать самый дальний потомок.
         Console.WriteLine(new String(' ', 3 * x.Length) + x[x.Length - 1]);
      }

      // Отображаем итоговое состояние обрабатываемых типов.
      Console.WriteLine("\n-> Of {0} types, {1} are " +
         "derived from System.Exception.",
         totalPublicTypes, totalExceptionTypes);
   }

   private static void LoadAssemblies() {
      String[] assemblies = {
         "System,                      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";

        // Получаем версию сборки, содержащей System.Object.
        // Предполагаем, что версии других сборок не отличаются.
        Version version =
            typeof(System.Object).Assembly.GetName().Version;

        // Явно загружаем сборки, которые нужно 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, 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 AppDomain, в другой AppDomain, не загружая в целевой AppDomain 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 маршалинг объекта по ссылке, создаются прокси-тип и прокси-объект. При маршалинге 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 AppDomain методом *LoadFrom* (а не *Load*) объекта *Assembly*. Поскольку ни один из методов *CreateInstanceFrom* не принимает параметр *Type*, все они возвращают ссылку на *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, позволяющими задавать AppDomain, в котором нужно создать объект. Методы, названия которых оканчиваются на *Unwrap*, удобнее, так как позволяют не выполнять дополнительный вызов 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 AppDomain, а затем возвращается ссылка на него. Ниже 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 AppDomain, а затем возвращается ссылка на новый объект. К этому методу мы тоже вернемся позднее.

Примечание CLR не требует, чтобы у значимого типа был конструктор. Но это проблема, 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 *SystemArray*) и делегатов (потомков *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 указывать, для какого из эк-

земплярных методов объекта или статического метода *CreateDelegate* должен служить оболочкой делегат.

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-типов. Затем надо получить возвращенный объект *Type* и передать его в один из описанных выше 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-тип 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, 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 разработчикам подключаемых компонентов (add-in) выбирать 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. Определив интерфейс, дайте сборке строгое имя (см. главу 3) и можете передавать 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 разворачивать ее вместе с файлом политики издателя (см. главу 3).

Примечание Типы, определенные в MSCorLib.dll, можно использовать: CLR всегда загружает ту версию MSCorLib.dll, что соответствует версии самой CLR. Корме 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 задействует меньше памяти.

- Разработчики подключаемых компонентов, конечно, определят свои типы в собственных сборках. Кроме того, их сборка будет ссылаться на вашу интерфейсную сборку. Сторонние разработчики также смогут выдавать новые версии своих сборок, когда захотят: приложение будет 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 типы, которые вы применяете для взаимодействия между сборками в отдельную сборку. Избегайте изменений этих типов и номеров версии такой сборки. Но, 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 сказанное выше. Во-первых, нам нужен код сборки для хоста.

```
using System;

namespace Wintellect.HostSDK {
   public interface IAddIn {
      String DoSomething(Int32 x);
   }
}
```

Затем я приведу код сборки подключаемого компонента, 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();
   }
}
```

В-третьих, приведу код Host.exe, сборки простого хоста (консольное приложение). При компоновке этой сборки используется ссылка на HostSDK.dll. Чтобы определить 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 обнаружения подключаемых модулей — Microsoft планирует реализовать его в одной из будущих версий CLR. К сожалению, для каждого хоста приходится создавать собственный механизм регистрации и обнаружения.

```
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 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 я не использую домены приложения. Но в реальной жизни подключаемые модули создаются в собственных AppDomain с собственными параметрами безопасности и конфигурации. И, конечно же, домен приложения может выгружаться, если нужно удалить подключаемый модуль из памяти. Чтобы обеспечить обмен между доменами AppDomain, лучше всего потребовать от разработчиков подключаемых модулей создавать собственные внутренние типы, производные от *MarshalByRefObject*. При создании нового AppDomain хост будет создавать в нем экземпляр собственного производного от *MarshalByRefObject* типа. Код хоста (в основном AppDomain) будет взаимодействовать с собственным типом (в других доменах AppDomain), заставляя их загружать сборки подключаемых модулей и создавать и использовать экземпляры определенных в этих модулях типов.

Использование отражения для обнаружения членов типа

В этой главе я рассказывал о частях отражения — загрузке сборок, обнаружении типов и создании объектов, — которые необходимы для создания динамически расширяемых приложений. Чтобы обеспечить высокую производительность и безопасность типов во время компиляции, нужно избегать использовать отражение. 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 расскажу о некоторых аспектах отражения, которые можно использовать для обнаружения и вызова членов типа. Возможность обнаружения и вызова членов типа обычно используется для создания инструмен-

тов для разработчиков и средств для 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 обнаружения и вызова членов типа для предоставления богатой функциональности разработчикам. Пример — библиотеки, обеспечивающие сериализацию и десериализацию, а также простую привязку к данным.

Обнаружение членов типа

Членами типа могут быть поля, конструкторы, методы, свойства, события и вложенные типы. В FCL есть тип *System.Reflection.MemberInfo*, абстрактный класс, инкапсулирующий набор свойств, общих для всех членов типа. У *MemberInfo* много дочерних классов (рис. 22-1), каждый из которых инкапсулирует чуть больше свойств отдельных членов типа.

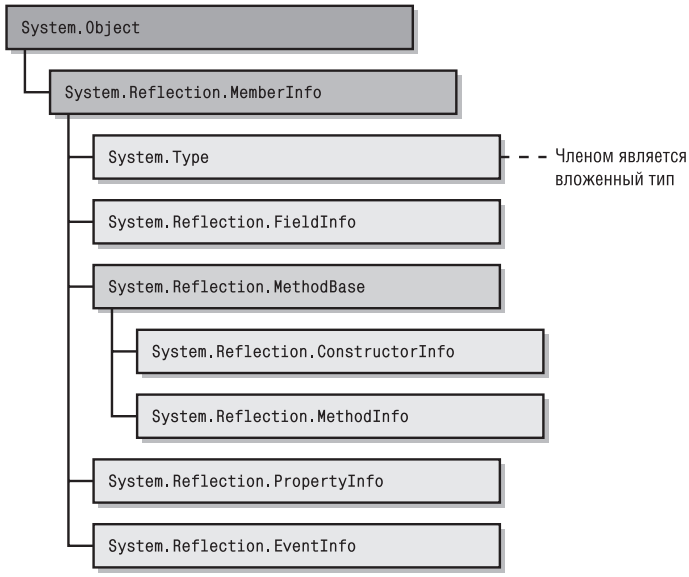


Рис. 22-1. *Иерархия типов отражения*

Следующая программа демострирует, как нужно запрашивать члены типов и отображать информацию о них. Этот код обрабатывает все открытые типы всех сборок, загруженных в вызывающий AppDomain. Для каждого типа вызывается метод *GetMembers*, который возвращает массив объектов типа, производного от *MemberInfo*; каждый объект ссылается на один член из тех, что определены в типе. Переменная *bf* типа *BindingFlags*, передаваемая в метод *GetMembers*, говорит ему, какие члены возвратить. (Подробнее о *BindingFlags* я расскажу чуть позже.) Далее для каждого члена отображается его тип (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 AppDomain.
      Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
      foreach (Assembly a in assemblies) {
         WriteLine(0, "Assembly: {0}", a);

         // Перечисляем типы сборки.
         foreach (Type t in a.GetExportedTypes()) {
            WriteLine(1, "Type: {0}", t);

            // Обнаруживаем 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 предоставляем 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 *MemberInfo* является корнем иерархии, стоит обсудить его подробнее. В табл. 22-1 показаны некоторые неизменяемые свойства и методы типа *MemberInfo*, общие для всех членов типа. Как вы помните, *System.Type* наследует типу *MemberInfo*, поэтому *Type* также обладает всеми свойствами, перечисленными в табл. 22-1.

Табл. 22-1. 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 следует «+» и имя вложенного типа
MemberType	Свойство (перечислитель) *MemberTypes*	Возвращает вид члена [поле, конструктор, метод, свойство, событие, тип (вложенный или нет)]
DeclaringType	Свойство *Type*	Возвращает *Type*, определяющий член
ReflectedType	Свойство *Type*	Возвращает *Type*, использованный для отражения этого члена
Module	Свойство *Module*	Возвращает *Module*, объявляющий член
MetadataToken	Свойство *Int32*	Возвращает маркер метаданных (в рамках модуля), идентифицирующий член
GetCustomAttributes	Метод, возвращающий *Object[]*	Возвращает массив, каждый элемент которого идентифицирует экземпляр нестандартного атрибута, которым помечен этот член. Такие атрибуты можно применять к любому члену
IsDefined	Метод, возвращающий *Boolean*	Возвращает true, если член помечен одним (и только одним) экземпляром указанного атрибута

Большинство названий свойств, указанных в табл. 22-1, самодостаточны. Однако разработчики часто путают *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*. С другой стороны, 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 *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 виды членов: *GetNestedTypes*, *GetFields*, *GetConstructors*, *GetMethods*, *GetProperties* и *GetEvents*. Все эти методы возвращают мас-

сивы, в которых каждый элемент является ссылкой на объект *Type*, *FieldInfo*, *ConstructorInfo*, *MethodInfo*, *PropertyInfo* и *EventInfo* соответственно.

На рис. 22-2 дается сводка типов, позволяющих приложению «пройти» по модели объектов отражения. AppDomain позволяет узнать, какие сборки в него загружены, объект сборки — узнать, из каких модулей она состоит, а сборка или модуль — выяснить определяемые в них типы. В свою очередь тип 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, нужно перечислить все типы в сборке и просмотреть их свойства *Namespace*.

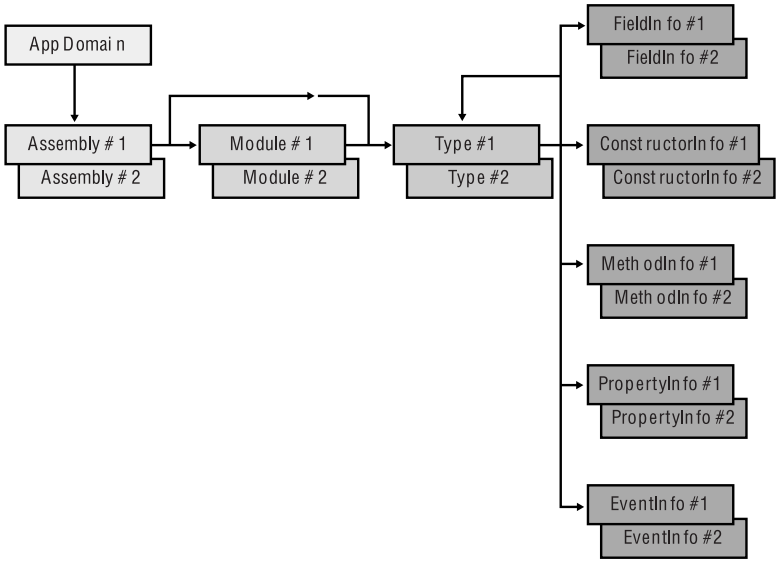


Рис. 22-2. *Типы, позволяющие приложению «пройти» по модели объектов отражения*

Тип также позволяет обнаружить 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 типы параметров членов. Можно также запрашивать свойство *ReturnParameter*, чтобы получить объект *ParameterInfo* с подробной информацией о возвращаемом значении члена. Чтобы получить набор параметров типа 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

Запрашивать члены типа можно, вызывая методы типа *Type* — *GetMembers*, *GetNestedTypes*, *GetFields*, *GetConstructors*, *GetMethods*, *GetProperties* или *GetEvents*. Вы-

зывая любой из этих методов, можно передавать экземпляр перечислимого типа *System.Reflection.BindingFlags*. Он содержит набор битовых флагов, объединенных оператором «или» (OR), и служит для фильтрации членов, возвращаемых этими методами. Идентификаторы, определяемые в перечислимом типе *BindingFlags* указаны в табл. 22-2.

Табл. 22-2. Идентификаторы поиска, определяемые в перечислимом типе *BindingFlags*

Идентификатор	Значение	Описание
Default	0x00	Заполнитель. Используется, если не указан ни один из остальных флагов, перечисленных в остальной части таблицы
IgnoreCase	0x01	Искать члены, имя которых совпадает с заданной строкой без учета регистра
DeclaredOnly	0x02	Искать только члены, с которыми тип был объявлен (игнорировать унаследованные члены)
Instance	0x04	Искать экземплярные члены
Static	0x08	Искать статические члены
Public	0x10	Искать открытые члены
NonPublic	0x20	Искать внутренние члены
FlattenHierarchy	0x40	Искать статические члены, определенные 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 вернут только открытые члены. Иначе говоря, значение по умолчанию — *BindingFlags.Public* | *BindingFlags.Instance* | *BindingFlags.Static*.

Заметьте: в *Type* также определены методы *GetMember*, *GetNestedType*, *GetField*, *GetConstructor*, *GetMethod*, *GetProperty* и *GetEvent*. Они позволяют передать строку, определяющую имя члена, информацию о котором надо получить. Именно в такой ситуации оказывается кстати флаг *IgnoreCase* типа *BindingFlags*.

Обнаружение интерфейсов типа

Для получения набора интерфейсов, наследуемых типом, используют методы *FindInterfaces*, *GetInterface* и *GetInterfaces* типа *Type*. Все они возвращают объекты *Type*, представляющие интерфейс. Они проходят по иерархии 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 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. 22-3).

Табл. 22-3. Открытые поля типа *InterfaceMapping*

Имя поля	Тип данных	Описание
TargetType	*Type*	Тип, использованный для вызова *GetInterfaceMapping*
InterfaceType	*Type*	Тип интерфейса, переданный методу *GetInterfaceMapping*
InterfaceMethods	*MethodInfo[]*	Массив, каждый элемент которого предоставляет информацию о методе интерфейса
TargetMethods	*MethodInfo[]*	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]* идентифицирует определенный в типе метод, реализующий этот метод интерфейса. Вот пример кода, демонстрирующий, как нужно обнаруживать интерфейсы и их методы, определенные 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):");

      // Отображаем 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++) {
           // Отображаем 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 фильтра.
   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, получим:

```
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()
```

Заметьте: интерфейса *IDisposable* в результатах работы программы нет, потому что он не определен в сборке EXE-файла.

Вызов членов типа

Теперь, когда мы знаем, как обнаруживать члены, определенные в типе, можно перейти к вызову этих членов. Что *конкретно* означает «вызывать», зависит от вида

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 возвращенное значение, вызов *PropertyInfo* — вызвать аксессор *get* или *set* свойства, а вызов *EventInfo* — создать или удалить обработчик события.

А теперь поговорим о вызове метода, потому что вызов этого члена самый сложный. А затем обсудим, как вызывать другие члены. В типе *Type* определен метод *InvokeMember*, который позволяет вызвать член. Существует несколько перегруженных версий *InvokeMember*, я собираюсь обсудить самую популярную; 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, используются соединителями.
   ...
}
```

Когда вы вызываете метод *InvokeMember*, он ищет среди членов типа соответствующий заданному. Если такого нет, генерируется исключение *System.MissingMethodException*, *System.MissingFieldException* или *System.MissingMemberException*, а если он найден, *InvokeMember* вызывает его. *InvokeMember* возвращает любое значение, которое вернет вызванный метод; если тот ничего не возвращает, *InvokeMember* возвращает *null*. Если вызываемый метод генерирует исключение, *InvokeMember* перехватывает его и генерирует исключение *System.Reflection.TargetInvocationException*, в свойстве *InnerException* объекта *System.Reflection.TargetInvocationException* при этом находится объект реального исключения, сгенерированного методом. Лично мне это не нравится, я бы предпочел, чтобы *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 у одного метода может быть несколько перегруженных версий, имена поля и методов тоже могут совпадать. Конечно, *InvokeMember* должен привязаться к какому-то одному члену, прежде чем он сможет вызвать его. Все параметры, передаваемые *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-потомок *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*.

Microsoft определила внутренний (недокументированный) тип *System.DefaultBinder* — потомок *Binder*. Этот тип входит в FCL, и Microsoft ожидает, что им будут пользоваться практически все. Некоторые поставщики компиляторов определят собственные типы, производные от *Binder*, и будут поставлять их в библиотеке времени выполнения, используемой кодом, созданным их компилятором. Если в метод *InvokeMember* передать в параметре *binder* значение null, метод будет использовать объект *DefaultBinder*.

При вызове методов *соединителя* (binder) 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.

Выше я показал значение флагов *BindingFlags*: *Default*, *IgnoreCase*, *DeclaredOnly*, *Instance*, *Static*, *Public*, *NonPublic* и *FlattenHierarchy* (табл. 22-2). Эти флаги подсказывают соединителю, какие члены включить в поиск.

Помимо этих флагов, соединитель определяет число аргументов, переданных параметром *args* метода *InvokeMember*. Число аргументов еще больше ограничивает набор возможных членов. Затем соединитель проверяет тип аргументов, еще больше уменьшая число возможных членов. Но когда дело доходит до типов аргумента, соединитель 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*, *DefaultBinder* все равно выберет этот метод. При вызове *InvokeMember* значение *Int32* будет преобразовано в *Int64*. В табл. 22-4 перечислены преобразования, поддерживаемые *DefaultBinder*.

Табл. 22-4. Преобразования, поддерживаемые объектом DefaultBinder

Исходный тип	Целевой тип
Любой тип	Его базовый тип
Любой тип	Реализованный в нем интерфейс
Char	*UInt16*, *UInt32*, *Int32*, *UInt64*, *Int64*, *Single*, *Double*
Byte	*Char*, *UInt16*, *Int16*, *UInt32*, *Int32*, *UInt64*, *Int64*, *Single*, *Double*
SByte	*Int16*, *Int32*, *Int64*, *Single*, *Double*
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

Есть и другие флаги *BindingFlags*, которые служат для тонкой настройки *DefaultBinder* (табл. 22-5):

Табл. 22-5. Флаги *BindingFlags*, используемые *DefaultBinder*

Идентификатор	Значение	Описание
ExactBinding	0x010000	Соединитель будет искать член с параметрами, соответствующими типу переданных аргументов
OptionalParamBinding	0x040000	Соединитель 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 *InvokeMember* объекта *Type*

Последний параметр метода *InvokeMember*, *culture*, также используется для привязки. Однако тип *DefaultBinder* игнорирует его. Определив собственный соединитель, можно использовать *culture* как вспомогательный параметр для преобразования типов аргументов. Скажем, вызывающий код может передать аргумент *String* со значением «1,23». Соединитель проверяет эту строку и выполняет ее синтаксический разбор с учетом региональных стандартов, заданных параметром *culture*, и преобразует тип аргумента в *Single* (если *culture* задан как «de-DE») или оставит его как есть (если *culture* — «en-US»).

Мы рассмотрели 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*, следует передать *null* в этом параметре.

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, укажите один из флагов *BindingFlags* (табл. 22-6).

Табл. 22-6. Флаги *BindingFlags*, используемые методом *InvokeMember*

Идентификатор	Значение	Описание
InvokeMethod	0x0100	Заставляет *InvokeMember* вызвать метод
CreateInstance	0x0200	Заставляет *InvokeMember* создать новый объект, вызвав его конструктор
GetField	0x0400	Заставляет *InvokeMember* получить значение поля
SetField	0x0800	Заставляет *InvokeMember* установить значение поля
GetProperty	0x1000	Заставляет *InvokeMember* вызвать метод-аксессор *get* свойства
SetProperty	0x2000	Заставляет *InvokeMember* вызвать метод-аксессор *set* свойства

Большинство этих флагов являются взаимоисключающими: при вызове *InvokeMember* может быть указан один и только один из них. Однако можно одновременно указывать *GetField* и *GetProperty*, и тогда *InvokeMember* будет искать сначала поле, а если не найдет его, будет искать подходящее свойство. Аналогично можно определить и сопоставить *SetField* и *SetProperty*. Соединитель использует эти флаги для суже-

ния круга возможных кандидатов. Если определить флаг *BindingFlags.CreateInstance*, соединитель 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 безопасность доступа к коду не дает злоупотреблять мощью 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 это так, то привязка будет успешной. Если этот член в обычных обстоятельствах вам недоступен, метод потребует разрешение *System.Security.Permissions.ReflectionPermission*, проверяя, установлен ли разряд *TypeInformation* флага *System.Security.Permissions.ReflectionPermissionFlags*. Если этот флаг установлен, метод привязывается к члену, а если запрос заканчивается неудачей, генерируется исключение *System.Security.SecurityException*.

Если вы вызываете метод для вызова члена, то этот метод производит аналогичную проверку, что и во время привязки к члену. Но в этом случае он проверяет разряд *MemberAccess* флага *ReflectionPermissionFlag*. Если он установлен, член вызывается, в противном случае генерируется исключение *SecurityException*.

Конечно, если ваша сборка пользуется полным доверием, проверки безопасности будут успешными, поэтому разрешение будет предоставлено и вызов успешно выполнен. Но отражение никогда, ни при каких обстоятельствах нельзя использовать для доступа к какому-либо незадокументированному 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 *InvokeMember* сначала привязывается к некоторому члену и только потом вызывает его. Если соединитель 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 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. Вот типы и методы, вызываемые для получения доступа к члену (табл. 22-7):

Табл. 22-7. Как вызвать член после привязки к нему

Тип	Описание
FieldInfo	Вызвать *GetValue* для получения значения поля, а затем — *SetValue*, чтобы задать его значение
ConstructorInfo	Вызвать *Invoke* для создания экземпляра типа и вызова конструктора
MethodInfo	Вызвать *Invoke* для вызова метода типа
PropertyInfo	Вызвать *GetValue* для вызова аксессора *get*, а затем — *SetValue* для вызова аксессора *set*
EventInfo	Вызвать *AddEventHandler* для вызова аксессора *add* события, а затем — *RemoveEventHandler* для вызова аксессора remove этого события

Тип *PropertyInfo* представляет информацию метаданных свойства (см. главу 9), то есть *PropertyInfo* поддерживает неизменяемые свойства *CanRead*, *CanWrite* и *PropertyType*. Эти свойства указывают, можно ли читать и записывать свойство, а также его тип данных. У *PropertyInfo* есть метод *GetAccessors*, возвращающий массив элементов *MethodInfo* — один для аксессора *get* (если он существует) и второй для аксессора *set* (если он существует). Более ценны методы *GetGetMethod* и *GetSetMethod* этого типа, каждый из которых возвращает только один объект *MethodInfo*. Методы *GetValue* и *SetValue* типа *PropertyInfo* существуют для удобства, внутренний их код получает соответствующие объекты *MethodInfo* и вызывает их.

Тип *EventInfo* представляет информацию о метаданных события (см. главу 10). *EventInfo* поддерживает неизменяемое свойство *EventHandlerType*, возвращающее объект *Type* для делегата события. У *EventInfo* также есть методы *GetAddMethod* и *GetRemoveMethod*, которые возвращают соответствующие объекты *MethodInfo*. Методы *AddEventHandler* и *RemoveEventHandler* существуют для удобства, внутренний их код получает соответствующие объекты *MethodInfo* и вызывает их.

Вызывая один из методов, перечисленных в правом столбце табл. 22-7, вы не привязываетесь к члену, а только вызываете его. Любой из этих методов можно вызывать многократно, и, 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 перегруженные 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*. Соединители могут игнорировать этот флаг, но *DefaultBinder* так не поступает. Обнаружив этот флаг, *DefaultBinder* не будет преобразовывать аргументы. Если переданные при этом аргументы не соответствуют ожидаемым методом, генерируется исключение *ArgumentException*.

Обычно, если для привязки к члену использован флаг *BindingFlags.ExactBinding*, то при вызове этого члена задают флаг *BindingFlags.SuppressChangeType*. Если не использовать эти флаги в паре, то вызов этого члена будет неудачным, если только переданные аргументы не совпадают в точности с аргументами, ожидаемыми методом. Кстати, если для привязки к члену и его последующего вызова исполь-

зуют метод *InvokeMethod* объекта *MemberInfo*, то обычно указывают или оба этих флага, или ни одного.

Следующее приложение-пример демонстрирует разные способы применения отражения для доступа к членам типа. Код показывает, как с помощью метода *InvokeMember* типа *Type* привязаться к члену и тут же вызвать его или просто привязаться к члену, чтобы вызвать его позже.

```
using System;
using System.Reflection;

// Этот класс демонстрирует отражение. 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", "value must be > 0");
         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);
      UsingInvokeMemberToBindAndInvokeTheMember(t);
      Console.WriteLine();
      BindingToMemberFirstAndThenInvokingTheMember(t);
   }

   private static void UsingInvokeMemberToBindAndInvokeTheMember(Type t) {
      Console.WriteLine("UsingInvokeMemberToBindAndInvokeTheMember");

      // Создаем экземпляр 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 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);

   // Вызываем метод.
   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);

   // ПРИМЕЧАНИЕ: InvokeMember не поддерживает событий.
}

private static void BindingToMemberFirstAndThenInvokingTheMember(Type t) {
   Console.WriteLine("BindingToMemberFirstAndThenInvokingTheMember");

   // Создаем экземпляр.
   ConstructorInfo ctor = t.GetConstructor(
      new Type[] { Type.GetType("System.Int32&") });
   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 и запись в поле.
   FieldInfo fi = obj.GetType().GetField("m_someField", c_bf);
```

```
      fi.SetValue(obj, 33);
      Console.WriteLine("someField: " + fi.GetValue(obj));

      // Вызываем метод.
      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));
      foreach (MethodInfo m in pi.GetAccessors())
         Console.WriteLine(m);
      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 или удаление делегата из события.
      EventInfo ei = obj.GetType().GetEvent("SomeEvent", c_bf);
      Console.WriteLine("AddMethod: "       + ei.GetAddMethod());
      Console.WriteLine("RemoveMethod: "    + ei.GetRemoveMethod());
      Console.WriteLine("EventHandlerType: " + ei.EventHandlerType);

      EventHandler ts = new EventHandler(EventCallback);
      ei.AddEventHandler(obj, ts);
      ei.RemoveEventHandler(obj, ts);
   }

   private static void EventCallback(Object sender, EventArgs e) {}
}
```

Вот что получится, если скомпоновать и запустить этот код:

```
UsingInvokeMemberToBindAndInvokeTheMember
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
someField: 5
ToString: 5
Property set catch.
SomeProp: 2

BindingToMemberFirstAndThenInvokingTheMember
x before constructor called: 12
```

```
Type: SomeType
x after constructor returns: 24
someField: 33
ToString: 33
Int32 get_SomeProp()
Void set_SomeProp(Int32)
Property set catch.
SomeProp: 2
AddMethod: Void add_SomeEvent(System.EventHandler)
RemoveMethod: Void remove_SomeEvent(System.EventHandler)
EventHandlerType: System.EventHandler
```

Заметьте: конструктор *SomeType* принимает в качестве единственного параметра ссылку на *Int32*. В предыдущем коде показано, как вызвать этот конструктор и как после завершения конструктора проверить модифицированное значение *Int32*. Далее в начале кода метода *BindingToMemberFirstAndThenInvokingTheMember* есть вызов метода *GetType* типа *Type*, которому передается строка «System.Int32&». Знак «амперсанд» (&) в строке позволяет определять параметр, передаваемый по ссылке. Это предусмотрено нотацией Бэкуса-Наура для записи имен типов, подробнее о ней см. документацию к FCL.

Использование описателей привязки для уменьшения рабочего набора

Во многих приложениях выполняется привязка ко многим типам (то есть объектам *Type*) или их членам (объектам-потомкам *MemberInfo*), а эти объекты сохраняются в определенном наборе. Позже приложение ищет нужный объект в наборе 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, его рабочий набор сильно увеличивается, 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 CLR для работы эти большие объекты не нужны. В приложениях, в которых сохраняется и кешируется много объектов *Type* и объектов-потомков *MemberInfo*, можно сократить рабочий набор, если использовать не объекты, а описатели времени выполнения. В FCL определены три типа таких описателей (все они определены 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 AppDomain. Так что теперь нам нужно научиться просто и эффективно преобразовать «тяжелые» объекты *Type/MemberInfo* в «легкие» описатели времени выполнения и наоборот. Это просто, если задействовать следующие методы и свойства преобразования.

- 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*.

Приведенный ниже пример программы создает много объектов *MethodInfo*, преобразует их в экземпляры *RuntimeMethodHandle* и предоставляет информацию о разнице в размере рабочего набора.

```
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() {
      // Отображаем размер кучи до отражения.
      Show("Before doing anything");

      // Создаем кеш 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);
      }

      // Отображаем число методов и размер кучи после привязки всех методов.
      Console.WriteLine("# of methods={0:###,###}", methodInfos.Count);
      Show("After building cache of MethodInfo objects");

      // Создаем кеш описателей RuntimeMethodHandles для всех объектов MethodInfo.
      List<RuntimeMethodHandle> methodHandles =
         methodInfos.ConvertAll<RuntimeMethodHandle>(
            delegate(MethodBase mb) { return mb.MethodHandle; });

      Show("Holding MethodInfo and RuntimeMethodHandle cache");
      GC.KeepAlive(methodInfos); // Запрещаем сборку мусора в кеше.
```

```
      methodInfos = null;       // Разрешаем сборку мусора в кеше.
      Show("After freeing MethodInfo objects");

      methodInfos = methodHandles.ConvertAll<MethodBase>(
         delegate(RuntimeMethodHandle rmh) {
            return MethodBase.GetMethodFromHandle(rmh); });
      Show("Size of heap after re-creating MethodInfo objects");
      GC.KeepAlive(methodHandles);  // Запрещаем сборку мусора в кеше.
      GC.KeepAlive(methodInfos);    // Запрещаем сборку мусора в кеше.

      methodHandles = null;         // Разрешаем сборку мусора в кеше.
      methodInfos = null;           // Разрешаем сборку мусора в кеше.
      Show("After freeing MethodInfos and RuntimeMethodHandles");
   }

   private static void Show(String s) {
      Console.WriteLine("Heap size={0,12:##,###,###} - {1}",
         GC.GetTotalMemory(true), s);
   }
}
```

Вот что получится, если скомпоновать и запустить этот код:

```
Heap size=    150,756   - Before doing anything
# of methods=44,493
Heap size=    3,367,084 - After building cache of MethodInfo objects
Heap size=    3,545,124 - Holding MethodInfo and RuntimeMethodHandle cache
Heap size=    368,936   - After freeing MethodInfo objects
Heap size=    1,445,496 - Size of heap after re-creating MethodInfo objects
Heap size=    191,036   - After freeing MethodInfos and RuntimeMethodHandles
```

ГЛАВА 23

Асинхронные операции

В этой главе я расскажу о различных способах асинхронного выполнения операций. Асинхронная операция, требующая большого количества вычислений, обычно выполняется в нескольких потоках. Но при асинхронных операциях ввода/вывода всю работу выполняет драйвер устройства Microsoft Windows и потоки не требуются. В этой главе я уделю основное внимание разработке и реализации приложения, в котором использованы асинхронные вычислительные операции и операции ввода/вывода, позволяющие улучшить его «отзывчивость» и масштабируемость.

Прежде чем начать, отмечу, что я создал библиотеку классов Wintellect Power Threading, упрощающую программирование асинхронных операций и синхронизации потоков. Библиотека предлагается «как есть», и ее можно бесплатно скачать с Web-сайта *http://Wintellect.com*.

Потоки Windows в CLR

Начну с рассказа об организации потоков и ее отношении к общеязыковой среде CLR. CLR использует реализованные в Windows механизмы организации потоков, и в этой главе я сосредоточусь на том, как они представлены для разработчиков, пишущих код для CLR. Иначе говоря, предполагается, что у читателей уже есть общее представление об организации потоков. Если же тема вам незнакома, рекомендую прочесть об этом в моих книгах, вышедших ранее, например «Programming Applications for Microsoft Windows» (Microsoft Press, 1999).

Все же, за CLR остается право отказаться от потоков Windows, и в некоторых сценариях хостинга нарушается четкое однозначное соответствие между CLR- и Windows-потоками. Например, хост диктует CLR представлять каждый CLR-поток как Windows-волокно[1]. Возможно, в следующей версии CLR будет всегда использоваться имеющийся поток, даже при явном запросе создания нового потока. Кроме того, она сможет выявлять потоки, находящиеся в состоянии ожидания, и пере-

[1] Волокно (fiber) — это облегченный поток, состоящий из стека и контекста (набора процессорных регистров). Windows ничего не знает о волокнах. Волокно может состоять из нескольких волокон, а поток одновременно может выполнять только одно волокно. Поток/волокно должен вызвать метод, планирующий выполнение другого волокна в потоке, потому что Windows не работает с волокнами и не умеет планировать их выполнение.

назначать их для выполнения других задач. Текущие версии CLR пока на это не способны. Но у Microsoft на этот счет есть немало идей, и, скорее всего, подобные возможности будут реализованы в следующих версиях CLR, что позволит повысить производительность путем уменьшения использования ресурсов ОС. Для разработчика это значит, что при работе с потоками в коде должно быть как можно меньше допущений. Например, нужно как можно реже вызывать «родные» функции Windows, потому что они ничего не знают о CLR-потоках, а также использовать типы из FCL. Такой подход позволит легко реализовать преимущества улучшенной производительности последующих версий CLR.

При выполнении программы на хосте (например, Microsoft SQL Server 2005) поток может сообщить хосту, что для выполнения кода ему нужно использовать текущий физический поток операционной системы, вызвав статический метод *BeginThreadAffinity* объекта *System.Threading.Thread*. Когда для работы потока физический поток операционной системы больше не требуется, поток может уведомить хост, вызвав статический метод *EndThreadAffinity* объекта *Thread*.

К вопросу об эффективном использовании потоков

По собственном опыту знаю, что разработчики слишком часто используют потоки в приложениях. Как показывает история, раньше операционные системы не поддерживали потоки. В однопоточных системах был лишь один поток, попеременно выполнявший все программы, а ошибка в программе приводила к сбою всей системы. Например, таковы 16-разрядные версии Windows: бесконечный цикл или сбой одного приложения «подвешивал» всю систему и заставлял принудительно перезагружаться. Поддержка многопоточности впервые была реализована в Windows NT 3.1 с целью повышения устойчивости системы. Каждому процессу выделялся отдельный поток, и зацикливание или зависание потока одного процесса не приводило к сбою потоков других процессов и всей системы.

Если быть точным, потоки приводят к издержкам. Создание потока обходится недешево: объект ядра процесса должен быть размещен и инициализирован, для каждого потока резервируется (и по требованию выделяется) 1 Мб адресного пространства для его стека пользовательского режима, и еще примерно 12 Кб выделяется для стека режима ядра. Создав поток, Windows сразу же вызывает функцию в каждой DLL-библиотеке процесса, уведомляя все DLL-библиотеки о создании нового потока. Уничтожение потока также требует затрат: каждая DLL-библиотека в процессе получает уведомление об уничтожении потока, также следует очистить объект ядра и стеки.

На однопроцессорных компьютерах в каждый момент времени может выполняться всего один поток. Windows должна отслеживать объекты-потоки и время от времени решать, какой поток процессор будет обрабатывать следующим. А это дополнительный код, выполняющийся примерно раз в 20 мс. Остановку выполнения процессором кода одного потока и запуск на выполнение кода другого потока называют *переключением контекста* (context switch). Оно обходится недешево, поскольку ОС должна выполнить следующее.

1. Перейти в режим ядра.
2. Сохранить регистры процессора в объекте ядра текущего процесса. Между прочим, регистры процессора занимают примерно 700 байт в процессорных архитектурах x86, 1240 байт — в x64 и 2500 байт — в IA64.
3. Установить спин-блокировку, определить, какой поток поставить следующим, и освободить спин-блокировку. Если следующий поток относится к другому процессу, затраты увеличатся, потому что потребуется переключение виртуального адресного пространства.
4. Загрузить регистры процессора со значениями из объекта ядра потока, который будет выполняться следующим.
5. Выйти из режима ядра.

Все это — чистые издержки, замедляющие работу Windows и всех приложений по сравнению с однопоточной системой. Но они необходимы, потому что Microsoft стремится к созданию отказоустойчивой ОС, не подверженной сбоям при неполадках в ходе выполнения кода приложений.

Все это аргументы в пользу следующего утверждения: потоки следует использовать как можно реже. Чем больше создается потоков, тем больше издержки в системе и тем медленнее работает приложение. Кроме того, каждому потоку нужны ресурсы (память для объекта ядра и двух стеков), кроме того, поток занимает память, которая становится недоступной для ОС и приложений.

Помимо повышения отказоустойчивости, потоки служат еще одной полезной цели — масштабируемости. В многопроцессорной системе Windows может планировать несколько потоков: по одному на каждый процессор. В таком случае имеет смысл создавать несколько потоков, чтобы эффективно задействовать возможности аппаратного обеспечения. Именно к этому все чаще прибегают разработчики. Мы привыкли, что с каждым годом процессоры становились все мощнее, а, значит, написанные тогда приложения работали быстрее на компьютерах с более мощным процессором. Но производителям процессоров, в числе которых Intel и AMD, становится все сложнее создавать более мощные процессоры, потому что современные процессорные архитектуры и аппаратные компоненты почти достигли пределов в рамках существующих конструкторских решений.

Современные технологии производства процессоров ограничены физическим размером процессоров, поэтому производители процессоров считают более эффективным создавать отдельные чипы, включающие несколько процессоров. Такие чипы повышают производительность многопоточных ОС и приложений за счет одновременного выполнения нескольких процессов. Современные производители процессорных чипов используют два типа технологий — Hyperthreading и *многоядерную* (multi-core), — которые позволяют представить для Windows и приложений отдельный чип как два (и более) процессора. На самом деле есть чипы (например, Intel Pentium Extreme), воплощающие обе технологии. Так, для ОС Windows один процессор Pentium Extreme равнозначен четырем процессорам!

В некоторых чипах, например Intel Xeon и Intel Pentium 4, реализована Hyperthreading. В этой технологии физический процессор состоит из двух логических. У каждого из них есть своя архитектура (регистры процессора), но они пользуются общими рабочими ресурсами, например кешем процессора. Когда один логический процессор приостанавливается (из-за промаха кеша, ошибочного

прогнозирования ветви или ожидания результатов предыдущей команды), чип переключается на другой логический процессор. В идеале для процессоров, использующих Hyperthreading, можно рассчитывать на стопроцентное улучшение производительности, ведь теперь вместо одного процессора работают сразу два. Но, поскольку они используют общие рабочие ресурсы, ожидаемого резкого повышения производительности нет. Так, по сообщению Intel, производительность повышается всего на 10–30%.

В некоторых чипах, таких как Intel Pentium B и AMD Athlon 64 X2, реализована технология нескольких ядер. На многоядерном чипе есть два физических процессора. У каждого — своя архитектура и рабочие ресурсы. Можно ожидать, что его рабочие характеристики будут лучше, чем у чипа Hyperthreading, и обеспечат стопроцентную производительность, потому что два процессора выполняют независимые задачи. Производители чипов уже объявили о создании в ближайшие несколько лет чипов с 4, 8, 16 и даже 32 независимыми процессорами. Вот это мощь!

Итак, на сегодняшний день производители процессоров не могут создать более быстрые процессоры, поэтому они совмещают несколько более медленных. Поэтому для обеспечения быстрой работы приложений в будущем их следует создавать с расчетом на использование дополнительных процессоров. Для этого нужны несколько потоков, а потоки — это лишние издержки; они замедляют работу и расходуют ресурсы. Немало приложений вполне обходится возможностями имеющихся процессоров, но если производительность приложения важна, этот вопрос необходимо как-то решать. Я думаю, со временем связанные с этим противоречием проблемы встанут более остро. Мои мысли, изложенные в этой и следующей главе, также были навеяны этим вопросом.

Пул потоков в CLR

Как отмечалось выше, создание и уничтожение потока — довольно дорогая операция в плане временны́х затрат. Кроме того, при наличии множества потоков впустую расходуется память и ухудшается производительность из-за того, что ОС должна планировать потоки и выполнять переключения контекста между потоками, готовыми к выполнению. К счастью, в CLR есть код для управления собственным пулом потоков. Пул потоков можно рассматривать как набор потоков, доступных для использования приложением. У каждого процесса есть один пул потоков, который используется всеми доменами AppDomain этого процесса.

При инициализации CLR пул потоков пуст. Сам пул потоков обслуживает очередь запросов на обработку. Когда приложению нужно выполнить асинхронную операцию, вызывается метод, размещающий соответствующий запрос в очередь пула потоков. Код пула потоков извлекает записи из очереди и передает их потоку из пула. Если пул пуст, создается новый поток. Как уже говорилось, создание потока снижает производительность. Но по завершении своей задачи поток из пула не уничтожается, а возвращается в пул и ожидает следующего запроса. Поскольку поток не уничтожается, производительность от этого не страдает.

Когда приложение отправляет много запросов пула потоков, пул пытается обслужить все запросы с помощью одного потока. Но, если приложение создает очередь запросов, а поток из пула не успевает их обработать, создаются допол-

нительные потоки. В конце концов, все запросы приложения будут обрабатываться небольшим числом потоков, и пулу не придется создавать множество потоков.

Если же приложение перестанет создавать запросы пула потоков, в пуле может оказаться много незанятых потоков, впустую занимающих ресурсы памяти. Поэтому после примерно 2 минут бездействия (в следующих версиях CLR это время может измениться) поток пробуждается и самоуничтожается, чтобы освободить ресурсы. Хотя самоуничтожение потока ухудшает производительность, это не столь важно, поскольку этот поток не был занят и приложение в данный момент не выполняет много работы.

Пул потоков позволяет найти золотую середину между двумя крайностями: малое число потоков экономит ресурсы, а большое — позволяет воспользоваться преимуществами многопроцессорных систем, многоядерных процессоров и технологии Hyperthreading. Пул потоков действует по эвристическому алгоритму, приспосабливаясь к текущей ситуации. Если приложение должно выполнить множество задач и есть доступные процессоры, пул создает больше потоков. При уменьшении загруженности приложения потоки из пула самоуничтожаются.

В пуле различают два типа потоков: *рабочие потоки* и *потоки ввода-вывода*. Первые используются, когда приложение запрашивает пул выполнить асинхронную вычислительную операцию (которая может включать инициацию операции ввода-вывода). А потоки ввода-вывода служат для уведомления кода о завершении асинхронной операции ввода-вывода. Это значит, что для выполнения запросов, например обращения к файлу, сетевому серверу, базе данных, Web-службе или аппаратному устройству, используется модель APM (Asynchronous Programming Model). Далее я расскажу о выполнении асинхронных вычислительных операций и об использовании APM-модели для выполнения асинхронных операций ввода-вывода.

Ограничение числа потоков в пуле

CLR позволяет разработчикам задавать максимальное число рабочих потоков и потоков ввода-вывода в пуле, а CLR гарантирует, что число созданных потоков никогда не превысит заданное. Я встречал немало разработчиков, которые тратили массу времени и сил на определение оптимального ограничения на число рабочих потоков. Чтобы облегчить вам жизнь, приведу несколько правил.

Верхний предел числа потоков в пуле задавать не следует — это может привести к нехватке процессорного времени и даже к взаимной блокировке потоков. Только представьте: 1000 рабочих элементов в очереди блокируются одним событием, которое освобождает 1001-й элемент. Если максимальное число потоков в пуле — 1000, 1001-й рабочий элемент не выполнится, и все 1000 потоков окажутся заблокированными навсегда. Вдобавок, разработчики обычно не прибегают к искусственному ограничению ресурсов, доступных для приложения. Например, кто станет при запуске приложения давать системе указание ограничить доступный для него объем памяти или пропускную способность сети? Но разработчики почему-то стремятся ограничить число потоков в пуле.

На самом деле сама идея ограничения числа потоков, реализованная в CLR, неудачна, потому что из-за нее программисты впустую тратят время. В предыдущих версиях CLR максимальное число рабочих потоков и потоков ввода-вывода

было ограничено небольшим значением по умолчанию, что иногда приводило к взаимной блокировке и недостатку процессорных ресурсов. К счастью, команда разработчиков CLR учла этот момент и в CLR версии 2.0 увеличила максимальное число рабочих потоков по умолчанию до 25 на каждый процессор компьютера[2], а число потоков ввода-вывода — до 1000. Последнее, в сущности, не накладывает никакого ограничения. Полностью убирать эти ограничения нельзя, потому что в таком случае нарушится работа приложений, созданных для предыдущих версий CLR. Кстати, в API Win32 нет средств ограничения числа потоков в пуле — по указанным выше причинам.

В классе *System.Threading.ThreadPool* есть несколько статических методов, которые служат для изменения числа потоков в пуле. Вот прототипы этих методов:

```
void GetMaxThreads(out Int32 workerThreads, out Int32 completionPortThreads);
Boolean SetMaxThreads(Int32 workerThreads, Int32 completionPortThreads);

void GetMinThreads(out Int32 workerThreads, out Int32 completionPortThreads);
Boolean SetMinThreads(Int32 workerThreads, Int32 completionPortThreads);

void GetAvailableThreads(out Int32 workerThreads, out Int32 completionPortThreads);
```

Метод *GetMaxThreads* позволяет узнать максимальное число потоков в пуле. Для изменения этого значения служит метод *SetMaxThreads*. Я настоятельно не рекомендую вызывать эти методы. Манипуляции с максимальным числом потоков в пуле обычно ухудшают, а не улучшают производительность приложения. Если же приложению нужно больше 25 потоков на процессор, значит, в его архитектуре и способе использования потоков есть серьезные изъяны. В этой главе я покажу, как правильно использовать потоки.

Вместе с тем, в пул потоков CLR заложен механизм предотвращения слишком частого создания потоков — разрешается создавать не более одного потока в 500 мс. Некоторых разработчиков это не устраивает, потому что задания из очереди обрабатываются медленнее. В этом случае в приложение можно добавить вызов метода *SetMinThreads* и передать в него минимальное допустимое число потоков в пуле. Пул быстро создаст указанное число потоков, а если при появлении в очереди новых заданий все потоки будут заняты, он продолжит создавать потоки со скоростью не более одного потока в 500 мс. По умолчанию минимальное число рабочих потоков и потоков ввода-вывода равно 1.

И, наконец, метод *GetAvailableThreads* позволяет узнать число дополнительных потоков, которые разрешено добавить в пул. Он возвращает максимальное число потоков за вычетом текущего числа потоков в пуле. Значение, возвращаемое этим методом, действительно лишь на момент вызова метода — к тому моменту, когда метод вернет управление приложению, в пул могут быть добавлены новые потоки или уничтожены имеющиеся. Этот метод вызывается в приложении лишь для диагностики, мониторинга или настройки производительности.

---

[2] Узнать число процессоров позволяет статическое свойство *ProcessorCount* объекта *System.Environment*. Оно возвращает число логических процессоров на компьютере. На компьютере с одним процессором Hyperthreading это свойство вернет значение 2.

Использование пула потоков для выполнения асинхронных вычислительных операций

В общем случае при выполнении вычислительных операций (например, при перерасчете ячеек в электронной таблице и проверке орфографии и грамматики в текстовом редакторе) не должны происходить никакие синхронные операции ввода-вывода, потому что они приостанавливают вызывающий поток на время работы аппаратных устройств (дисковода, сетевого адаптера и т. п.). Нужно всегда стремиться к тому, чтобы потоки были активны. Приостановленный поток не выполняет никакой полезной работы, но использует ресурсы системы — именно этого следует избегать при создании высокопроизводительного масштабируемого приложения. Далее в этой главе я покажу, как с помощью APM-модели эффективно выполнять асинхронные операции ввода-вывода.

Чтобы добавить в очередь пула потоков асинхронную вычислительную операцию, обычно вызывают один из следующих методов класса *ThreadPool*:

```
static Boolean QueueUserWorkItem(WaitCallback callBack);
static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);
static Boolean UnsafeQueueUserWorkItem(WaitCallback callBack, Object state);
```

Эти методы ставят «рабочий элемент» (вместе с дополнительными данными состояния) в очередь пула потоков и сразу возвращают управление приложению. Рабочий элемент — это просто указанный в параметре *callback* метод, который вызывается потоком из пула. Этому методу можно передать один параметр, указанный в параметре *state* (данные состояния). Версия метода *QueueUserWorkItem* без параметра *state* передает *null* методу обратного вызова. В конце концов один из потоков пула обработает рабочий элемент, что приведет к вызову указанного метода. Создаваемый метод обратного вызова должен соответствовать типу-делегату *System.Threading.WaitCallback*, который определяется так:

```
delegate void WaitCallback(Object state);
```

В следующем коде показано, как заставить поток из пула вызвать метод асинхронно.

```
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);  // Выполнение других задач (10 с).
      Console.WriteLine("Hit <Enter> to end this program...");
      Console.ReadLine();
   }

   // Сигнатура этого метода должна соответствовать делегату WaitCallback.
   private static void ComputeBoundOp(Object state) {
      // Этот метод выполняется потоком из пула.
```

```
        Console.WriteLine("In ComputeBoundOp: state={0}", state);
        Thread.Sleep(1000);  // Выполнение других задач (1 секунда).

        // Когда этот метод возвращает управление приложению,
        // поток возвращается в пул и ожидает следующее задание.
    }
}
```

Скомпилировав и выполнив этот код, получим следующее.

```
Main thread: queuing an asynchronous operation
Main thread: Doing other work here...
In ComputeBoundOp: state=5
```

А иногда при выполнении этого кода результат будет таким:

```
Main thread: queuing an asynchronous operation
In ComputeBoundOp: state=5
Main thread: Doing other work here...
```

Разный порядок строк в данном случае объясняется тем, что два метода выполняются асинхронно по отношению друг к другу. Если приложение запущено на многопроцессорном компьютере, планировщик Windows определяет, какой поток будет первым или будут ли они выполняться одновременно.

Примечание Если метод обратного вызова генерирует необработанное исключение, CLR завершит процесс (если это не противоречит политике хоста). Подробнее о необработанных исключениях см. главу 19.

Метод *UnsafeQueueUserWorkItem* класса *ThreadPool* очень похож на чаще используемый метод *QueueUserWorkItem*. Вкратце скажу, чем они различаются. При обращении к ограниченному ресурсу (например, при открытии файла) CLR выполняет проверку доступа к коду (CAS), то есть наличие разрешений на доступ к ресурсу у всех сборок в стеке вызывающего потока. При отсутствии у какой-либо из них нужных разрешений CLR генерирует исключение *SecurityException*. В частности, это исключение возникнет, когда поток, выполняющий код сборки, у которой нет разрешения на открытие файла, все-таки попытается его открыть.

Чтобы обойти это ограничение, поток может поставить рабочий элемент в очередь пула потоков, тогда код открытия файла будет выполнен потоком из пула. Конечно, все это должно происходить в сборке, имеющей необходимые разрешения. Это «решение» обходит защиту и открывает злоумышленникам доступ к ресурсам с ограниченным доступом. Чтобы устранить эту брешь в защите, метод *QueueUserWorkItem* просматривает стек вызывающего потока и отслеживает все разрешения на доступ, а затем ассоциирует их с потоком из пула, когда он начинает выполняться. Таким образом, поток из пула работает с теми же разрешениями, что и поток, вызвавший метод *QueueUserWorkItem*.

Просмотр стека потока и отслеживание всех разрешений на доступ заметно снижают производительность. Чтобы улучшить быстродействие постановки в очередь асинхронной вычислительной операции, можно вместо *QueueUserWorkItem* вызывать метод *UnsafeQueueUserWorkItem*, который просто ставит элемент в очередь к пулу потоков и не просматривает стек вызывающего потока. В результате

этот метод выполняется быстрее, чем *QueueUserWorkItem*, но создает брешь в защите приложения. Поэтому *UnsafeQueueUserWorkItem* нужно вызывать, только будучи уверенным, что поток из пула будет выполнять код, не обращающийся к ресурсу с ограниченным доступом, или если обращение к этому ресурсу предусмотрено в приложении. Также учтите, что в коде, вызывающем метод *UnsafeQueueUserWorkItem*, должны быть установлены флаги *ControlPolicy* и *ControlEvidence* для *SecurityPermission* — это не позволит ненадежному коду случайно или преднамеренно повысить уровень разрешений.

В разделе «Контексты выполнения» этой главы подробнее рассказывается о переключении контекста (и разрешений доступа) между потоками.

Использование выделенного потока для выполнения асинхронной операции

Я настоятельно рекомендую применять пул потоков для выполнения асинхронных операций везде, где это возможно. Но иногда приходится явно создавать поток, предназначенный для выполнения конкретной вычислительной операции. Обычно выделенный поток создается для выполнения кода, требующего особого режима потока, нетипичного для потоков из пула. В частности, выделенный поток создается для выполнения задачи с определенным приоритетом (все потоки из пула выполняются со стандартным приоритетом, который не следует изменять). Также имеет смысл создать выделенный поток, если нужно, чтобы поток работал в активном (foreground) режиме (все потоки из пула выполняются в фоновом режиме). В этом случае приложение завершится лишь после того, как поток выполнит свою задачу. Выделенный поток также пригодится для вычислительной задачи, требующей очень много времени. Это освободит логику пула потоков от необходимости решать вопрос о создании дополнительных потоков. И, наконец, выделенный поток нужен, когда может потребоваться завершить его досрочно, вызывав метод *Abort* объекта *Thread* (см. главу 21).

Перед созданием выделенного потока нужно создать экземпляр класса *System.Threading.Thread* и передать имя метода его конструктору. Так выглядит прототип конструктора.

```
public sealed class Thread : CriticalFinalizerObject, ... {
   public Thread(ParameterizedThreadStart start);
   // Конструкторы, которые используются реже, здесь не приводятся.
}
```

В параметре *start* передается метод, который будет выполнять выделенный поток. Он должен соответствовать сигнатуре делегата *ParameterizedThreadStart*:

```
delegate void ParameterizedThreadStart(Object obj);
```

Как видите, у делегатов *ParameterizedThreadStart* и *WaitCallback* (который рассматривался в предыдущем разделе) одинаковые сигнатуры. Это значит, что один и тот же метод может вызываться как потоком из пула, так и выделенным потоком.

Создание объекта *Thread* не снижает производительность, потому что поток операционной системы пока не создается. Чтобы создать поток операционной системы и заставить его начать выполнение метода обратного вызова, нужно

вызвать метод *Start* объекта *Thread* и передать ему объект (состояние), который нужно передать как параметр метода обратного вызова. Следующий код создает выделенный поток и заставляет его асинхронно вызвать метод.

```
using System;
using System.Threading;

public static class Program {
   public static void Main() {
      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();    // Ожидает завершение потока.
      Console.WriteLine("Hit <Enter> to end this program...");
      Console.ReadLine();
   }

   // Сигнатура этого метода должна соответствовать делегату ParameterizedThreadStart
   private static void ComputeBoundOp(Object state) {
      // Этот метод выполняется выделенным потоком.

      Console.WriteLine("In ComputeBoundOp: state={0}", state);
      Thread.Sleep(1000); // Имитирует другую работу (1 секунда).

      // Когда этот метод возвращает управление программе,
      // выделенный поток уничтожается.
   }
}
```

Скомпилировав и выполнив этот код, получим следующее:

```
Main thread: starting a dedicated thread to do an asynchronous operation
Main thread: Doing other work here...
In ComputeBoundOp: state=5
```

Иногда, выполняя этот код, я получаю следующее (как отмечалось в предыдущем разделе):

```
Main thread: starting a dedicated thread to do an asynchronous operation
In ComputeBoundOp: state=5
Main thread: Doing other work here...
```

Обратите внимание, что в этом примере, как и в предыдущем разделе, использован метод *ComputeBoundOp*, а *Main* вызывает метод *Join*. Последний заставляет вызывающий поток полностью прекратить выполнение кода, пока поток, указанный в *dedicatedThread*, не самоуничтожится или не завершится. При постановке в очередь асинхронной операции с помощью метода *QueueUserWorkItem* объекта

ThreadPool в среде CLR невозможно определить время завершения операции. В случае выделенного потока это можно сделать с помощью метода *Join*. Все же, не следует применять выделенный поток вместо вызова метода *QueueUserWorkItem*, если нужно узнать, когда завершилось выполнение операции. Для этого следует использовать APM-модель (подробнее о ней далее в этой главе).

Периодическое выполнение асинхронной операции

В пространстве имен *System.Threading* определен класс *Timer*, который позволяет периодически вызывать методы в CLR. Создание экземпляра класса *Timer* говорит CLR что вскоре придется выполнить обратный вызов метода 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 четыре конструктора создают объект *Timer*. Параметр *callback* содержит имя метода, обратный вызов которого должен выполниться потоком из пула. Конечно, созданный метод обратного вызова должен соответствовать типу-делегату *System.Threading.TimerCallback*:

```
delegate void TimerCallback(Object state);
```

Параметр *state* конструктора служит для передачи методу обратного вызова данных о состоянии или *null*, если таких данных нет. Параметр *dueTime* позволяет задать для CLR время ожидания (в мс) перед первым вызовом метода обратного вызова. Оно представляется 32-разрядным значением со знаком или без, 64-разрядным значением со знаком или значением *TimeSpan*. Если метод обратного вызова должен вызываться немедленно, надо приравнять параметр *dueTime* к 0. Последний параметр, *period*, указывает периодичность (в мс) последующих вызовов метода обратного вызова. Если ему передано *Timeout.Infinite (- 1)*, поток из пула вызовет метод обратного вызова только раз.

В CLR есть лишь один поток, используемый для всех объектов *Timer*, который осведомлен о времени следующего объекта *Timer*. Когда приходит время следующего объекта *Timer*, поток CLR пробуждается и вызывает метод *QueueUserWorkItem* объекта *ThreadPool*, чтобы добавить запись в очередь пула потоков для вызова метода обратного вызова. Если метод обратного вызова выполняется долго, таймер может сработать опять. Вполне возможна ситуация, в которой один метод

обратного вызова выполняется несколькими потоками из пула. Будьте внимательны: если метод обращается к совместно используемым данным, лучше добавить блокировки синхронизации потоков, чтобы защитить эти данные от повреждения.

Класс *Timer* поддерживает несколько дополнительных методов, позволяющих указать для CLR время и условия обратного вызова метода. В частности, это методы *Change* и *Dispose* (у каждого есть перегруженные версии):

```
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);

   public Boolean Dispose();
   public Boolean Dispose(WaitHandle notifyObject);
}
```

Метод *Change* позволяет изменить или сбросить значение счетчика *Timer*, метод *Dispose* — полностью отключить таймер или сообщить объекту ядра, указанному в параметре *notifyObject*, когда все ожидающие обратные вызовы будут завершены.

Внимание! При утилизации объекта *Timer* сборщиком мусора CLR останавливает таймер, чтобы он больше не срабатывал. Поэтому при использовании объекта *Timer* следует проверять, есть ли переменная, поддерживающая его «на плаву», иначе таймер будет уничтожен и вызовы метода обратного вызова прекратятся (см. главу 20).

В следующем коде показано, как поток из пула вызывает метод, который выполняется немедленно, а затем каждые две секунды.

```
using System;
using System.Threading;

public static class Program {
   public static void Main() {
      Console.WriteLine("Main thread: starting a timer");
      Timer t = new Timer(ComputeBoundOp, 5, 0, 2000);

      Console.WriteLine("Main thread: Doing other work here...");
      Thread.Sleep(10000);   // Имитация другой работы (10 секунд).
      t.Dispose();           // Остановка таймера.
   }

   // Сигнатура этого метода должна соответствовать делегату 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).
```

```
        // Когда этот метод возвращает управление программе,
        // поток возвращается в пул и ожидает другое задание.
    }
}
```

История трех таймеров

Библиотека FCL поставляется с тремя таймерами, но для большинства программистов остается загадкой, чем они отличаются. Попробую объяснить.

- **Класс *Timer* из пространства имен *System.Threading*** рассматривался в предыдущем разделе. Он лучше других подходит для выполнения периодических фоновых задач в другом потоке.
- **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*). Когда таймер срабатывает, Windows добавляет в очередь сообщений потока сообщение таймера (WM_TIMER). Поток должен выполнить прокачку сообщений, чтобы извлечь эти сообщения и передать их нужному методу обратного вызова. Обратите внимание: вся работа ложится на один поток — установка таймера и обработка метода обратного вызова выполняются одним и тем же потоком. Это предотвращает параллельное выполнение метода таймера несколькими потоками.
- **Класс *Timer* из пространства имен *System.Timers*** является, по сути, оболочкой для класса *Timer* из пространства имен *System.Threading*. Он заставляет CLR по срабатыванию таймера ставить события в очередь пула потоков. Класс *System.Timers.Timer* происходит от класса *Component* из пространства имен *System.ComponentModel*, что позволяет объекты-таймеры размещать в области конструктора форм в Microsoft Visual Studio. Также члены этого класса немного отличаются от других. Этот класс был добавлен в FCL давным-давно, когда у Microsoft еще не было четкой концепции потоков и таймеров. Вообще говоря, его стоило бы удалить, чтобы вместо него применялся класс *System.Threading.Timer*. Я никогда не использую класс *System.Timers.Timer* и вам не советую, за исключением случаев, когда нужно поместить таймер в область конструктора форм.

Модель асинхронного программирования

Асинхронные операции — это ключ к созданию высокопроизводительных, масштабируемых приложений, которые позволяют выполнять много операций, используя очень небольшое число потоков. А вкупе с пулом потоков они дают возможность эффективно задействовать все процессоры в системе. Осознавая этот огромный потенциал, группа разработчиков CLR приступила к созданию модели, которая сделала бы его доступным для всех программистов. Эта модель была названа *моделью асинхронного программирования* (APM).

Лично мне эта модель очень симпатична, потому что она предоставляет единую модель асинхронного выполнения как вычислительных операций, так и ввода-вывода. Ее довольно легко освоить, она проста в использовании и поддерживается многими типами библиотеки FCL. Приведу 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*. Заметьте: классы, производные от *Stream*, которые не взаимодействуют с аппаратными устройствами (в том числе классы *BufferedStream*, *MemoryStream* и *CryptoStream*), также поддерживают эти методы и могут использоваться в модели APM. Но код в этих методах выполняет лишь вычислительные операции, но не операции ввода-вывода, поэтому для выполнения последних нужен поток.
- Класс *System.Net.Dns* поддерживает методы *BeginGetHostAddresses*, *BeginGetHostByName*, *BeginGetHostEntry* и *BeginResolve*.
- 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*), поддерживают методы *BeginGetRequestStream* и *BeginGetResponse*.
- У класса *System.IO.Ports.SerialPort* есть доступное только для чтения свойство *BaseStream*, возвращающее объект *Stream*, который, как известно, поддерживает методы *BeginRead* и *BeginWrite*.
- Класс *System.Data.SqlClient.SqlCommand* поддерживает методы *BeginExecuteNonQuery*, *BeginExecuteReader* и *BeginExecuteXmlReader*.

Кроме того, во всех типах-делегатах определен метод *BeginInvoke*, который можно использовать в APM. Наконец, инструменты создания типов-прокси Web-сервисов (такие как WSDL.exe и SvcUtil.exe) также генерируют методы с названиями, начинающимися с *Begin*, пригодные для использования в рамках APM. Кстати, у каждого такого метода есть парный метод, название которого начинается с *End*. Как видите, поддержка APM пронизывает всю библиотеку FCL.

Важная особенность модели APM — наличие в ней трех методов стыковки. Вспомните: метод *QueueUserWorkItem* объекта *ThreadPool* не обеспечивает встроенных механизмов определения времени завершения асинхронной операции и ее результатов. Например, если в очередь поставлена асинхронная операция проверки правописания слова, поток из пула не сможет сообщить о результатах проверки. Так вот, модель APM поддерживает три механизма, позволяющие определить время завершения асинхронной операции, а также ее результат.

Использование модели APM для выполнения асинхронного ввода-вывода

Допустим, нужно асинхронно прочесть несколько байт из файла с использованием модели APM. Сначала следует создать объект *System.IO.FileStream*, вызвав один из его конструкторов, который принимает параметр *System.IO.FileOptions*. Для этого параметра нужно передать флаг *FileOptions.Asynchronous*, который сообщает объекту *FileStream*, что нужно выполнить асинхронную операцию чтения и записи в файле.

Чтобы синхронно прочесть байты из объекта *FileStream*, нужно вызвать метод *Read*. Вот его прототип:

```
public Int32 Read(Byte[] array, Int32 offset, Int32 count)
```

Метод *Read* принимает ссылку на массив *Byte[]*, в который будут записываться байты из файла. Параметр *count* задает максимальное число байт, которые нужно прочесть. Байты помещаются в массив *array* под номерами от *offset* до (*offset* + *count* – 1). Метод *Read* возвращает число байт, фактически прочитанных из файла. При вызове этого метода чтение выполняется синхронно. Это значит, что метод возвращает управление приложению лишь после того, как все запрошенные байты считаны в байтовый массив. Синхронная операция ввода-вывода очень неэффективна, поскольку время ее выполнения непредсказуемо, а вызывающий поток приостанавливается до ее завершения, поэтому не может выполнять другие задачи и впустую расходует ресурсы.

Если Windows сохранила содержимое файла в кеше, этот метод почти сразу вернет управление приложению. Но если нужных данных в кеше нет, Windows придется обращаться к жесткому диску, чтобы загрузить с него содержимое файла. Возможно, ОС даже придется обращаться по сети к серверу, чтобы тот обратился к своему жесткому диску (или кешу) и вернул нужные данные. Нужно понимать, что, вызывая метод *Read*, невозможно предугадать, когда он получит данные и вернет управление программе. А в случае сбоя сетевого подключения время ожидания метода *Read* истечет, и он сгенерирует исключение, информирующее о сбое.

Чтобы асинхронно считать байты из файла, нужно вызвать метод *BeginRead* объекта *FileStream*:

```
IAsyncResult BeginRead(Byte[] array, Int32 offset, Int32 numBytes,
   AsyncCallback userCallback, Object stateObject)
```

Заметьте: первые три параметра у методов *BeginRead* и *Read* совпадают. Но у *BeginRead* есть еще два параметра — *userCallback* и *stateObject*. О них чуть позже. Вызов метода *BeginRead* дает Windows команду прочесть байты из файла в байтовый массив. Поскольку это операция ввода-вывода, метод *BeginRead* ставит этот запрос в очередь драйвера соответствующего аппаратного устройства. Вся работа ложится на аппаратное устройство, и потокам больше ничего не нужно делать — даже ждать результатов операции. Это очень эффективно!

Примечание Хотя в данном разделе речь идет о файловом вводе-выводе, важно помнить, что другие типы ввода-вывода работают аналогично. Например, при использовании объекта *NetworkStream* запрос на ввод-вывод ставится в очередь драйвера Windows сетевого устройства и ожидается ответ. Также при сетевом вводе-выводе связанные с этой операцией запросы (например, поиск в DNS, запросы Web-сервисов и баз данных) в конечном итоге приводят к размещению в очереди драйвера сетевого устройства асинхронных операций ввода-вывода. Уровень абстрагирования просто анализирует полученные в ответ данные и делает их более удобными для использования в коде приложения.

Метод *BeginRead* возвращает ссылку на объект, тип которого реализует интерфейс *System.IAsyncResult*. При вызове метода *BeginRead* он создает объект, который уникально идентифицирует запрос на ввод-вывод, ставит запрос в очередь драйвера устройства Windows, а затем возвращает объект *IAsyncResult*. Этот объект подобен расписке в получении. Когда метод *BeginRead* возвращает управление

приложению, операция ввода-вывода была лишь поставлена в очередь, но еще не завершена. Поэтому не пытайтесь использовать байты из байтового массива — ведь в нем еще нет всех запрошенных данных.

На самом деле, массив уже может содержать все запрошенные данные, потому что операция ввода-вывода выполняется асинхронно и к моменту завершения работы метода *BeginRead* чтение нужных данных может быть завершено. Или же данные поступят с сервера через несколько секунд. А, может быть, произошел сбой подключения к серверу, и данные не поступят никогда. Поскольку все эти ситуации возможны, необходим способ определить, что и когда произошло на самом деле. Я называю это *стыковкой* (rendezvousing) с результатом асинхронной операции. Как уже говорилось ранее, модель APM предлагает три метода стыковки. Позже я рассмотрю и сравню их друг с другом.

Примечание При создании объекта *FileStream* можно задать тип операции для взаимодействия с этим объектом (синхронная или асинхронная) с помощью флага *FileOptionsAsynchronous* (это равносильно вызову Win32-функции *CreateFile* и передаче ей флага *FILE_FLAG_OVERLAPPED*). Если этот флаг не задать, Windows выполнит все операции с файлом синхронно. Конечно, можно вызвать метод *BeginRead* объекта *FileStream* — приложение будет считать, что операция выполняется асинхронно, но на самом деле класс *FileStream* будет использовать другой поток для эмуляции асинхронной операции. Этот дополнительный поток расходует ресурсы и ухудшает производительность.

С другой стороны, можно создать объект *FileStream*, задав флаг *FileOptionsAsynchronous*. Затем можно вызвать метод *Read* объекта *FileStream* для выполнения синхронной операции. Сам класс *FileStream* эмулирует это действие так: сразу после начала выполнения асинхронной операции он переводит вызывающий поток в спящий режим до завершения операции. Это тоже неэффективно, но не так, как вызов метода *BeginRead* объекта *FileStream*, созданного без флага *FileOptionsAsynchronous*.

Итак: при работе с объектом *FileStream* сначала нужно решить, какая операция ввода-вывода будет выполняться с файлом (синхронная или асинхронная), и соответственно установить (или сбросить) флаг *FileOptionsAsynchronous*. При установленном флаге необходимо вызывать метод *BeginRead*, иначе — *Read*. Это обеспечит наилучшую производительность. Если с объектом *FileStream* нужно выполнить несколько синхронных и асинхронных операций, этот объект лучше создать с флагом *FileOptionsAsynchronous*.

Три метода стыковки в модели APM

Модель APM поддерживает три метода стыковки: ожидание завершения, опрос и обратный вызов метода. Рассмотрим их подробнее.

Стыковка с использованием ожидания завершения

Асинхронная операция инициируется вызовом метода, название которого начинается с *Begin* (*BeginXxx*). Такие методы ставят операцию в очередь и возвращают объект *IAsyncResult*, указывающий на выполняемую операцию. Для получения результата операции нужно просто вызвать соответствующий метод *EndXxx*, передав ему объект *IAsyncResult*. Учтите, что любой метод *EndXxx* принимает в качестве параметра объект *IAsyncResult*. По сути, при вызове метода *EndXxx* CLR получает команду вернуть результат асинхронной операции, на которую указывает объект *IAsyncResult* (полученный в результате вызова метода *BeginXxx*).

Итак, если на момент вызова метода *EndXxx* асинхронная операция уже завершилась, он сразу вернет ее результат. В противном случае *EndXxx* приостановит вызывающий поток до завершения операции, а затем вернет результат.

Вернемся к примеру считывания байт из объекта *FileStream*. В классе *FileStream* есть метод *EndRead*:

```
Int32 EndRead(IAsyncResult asyncResult)
```

Заметьте: у него всего один параметр — *IAsyncResult*, и, что особенно важно, его возвращаемый тип (*Int32*) совпадает с возвращаемым типом метода *Read*. Метод *EndRead* возвращает число байт, считанных из *FileStream*.

Проиллюстрирую все вышесказанное на примере.

```
using System;
using System.IO;
using System.Threading;

public static class Program {
   public static void Main() {
      // Открытие файла для асинхронного ввода-вывода.
      FileStream fs = new FileStream(@"C:\Boot.ini", FileMode.Open,
         FileAccess.Read, FileShare.Read, 1024,
         FileOptions.Asynchronous);

      Byte[] data = new Byte[100];

      // Начало асинхронной операции чтения из файла FileStream.
      IAsyncResult ar = fs.BeginRead(data, 0, data.Length, null, null);

      // Здесь выполняется другой код…

      // Приостановка этого потока до завершения асинхронной операции
      // и получения ее результата.
      Int32 bytesRead = fs.EndRead(ar);

      // Других операций нет. Закрытие файла.
      fs.Close();

      // Теперь можно обратиться к байтовому массиву и вывести результат операции.
      Console.WriteLine("Number of bytes read={0}", bytesRead);
```

```
      Console.WriteLine(BitConverter.ToString(data, 0, bytesRead));
   }
}
```

Скомпилировав и запустив эту программу на своем компьютере, я получил следующий результат (на других компьютерах получится не точно такой, но аналогичный результат):

```
Number of bytes read=100
5B-62-6F-6F-74-20-6C-6F-61-64-65-72-5D-0D-0A-74-69-6D-65-6F-
75-74-3D-33-30-0D-0A-64-65-66-61-75-6C-74-3D-6D-75-6C-74-69-
28-30-29-64-69-73-6B-28-30-29-72-64-69-73-6B-28-30-29-70-61-
72-74-69-74-69-6F-6E-28-31-29-5C-57-49-4E-44-4F-57-53-0D-0A-
5B-6F-70-65-72-61-74-69-6E-67-20-73-79-73-74-65-6D-73-5D-0D
```

В этой программе модель APM использована неэффективно. Нет смысла вызывать метод *BeginXxx*, а затем сразу — *EndXxx*, потому что до завершения операции вызывающий поток переходит в режим ожидания. Для синхронного выполнения этой операции достаточно было бы просто вызвать метод *Read*, и это было бы гораздо эффективнее.

Однако, если между вызовами *BeginRead* и *EndRead* добавить определенный код, преимущество модели APM станет явным: этот код будет выполняться во время считывания байт из файла. Вот та же программа, но с важными изменениями. Теперь она считывает данные из нескольких потоков одновременно. Здесь я использовал объекты *FileStream*, но программа прекрасно работает с любыми объектами, производными от *Stream* — она одновременно считывает байты из нескольких файлов, сокетов и даже с последовательных портов. Для этого нужно всего лишь воспользоваться моим классом *AsyncStreamRead* и передать соответствующие параметры конструктору этого класса.

```
private static void ReadMultipleFiles(params String[] pathnames) {
   AsyncStreamRead[] asrs = new AsyncStreamRead[pathnames.Length];

   for (Int32 n = 0; n < pathnames.Length; n++) {
      // Открытие файла для асинхронного ввода-вывода.
      Stream stream = new FileStream(pathnames[n], FileMode.Open,
         FileAccess.Read, FileShare.Read, 1024,
         FileOptions.Asynchronous);

      // Начало асинхронной операции чтения из объекта Stream.
      asrs[n] = new AsyncStreamRead(stream, 100);
   }

   // Все объекты Stream были открыты,
   // а все запросы на чтение поставлены в очередь.
   // Все они выполняются одновременно!

   // Получение и вывод результатов.
   for (Int32 n = 0; n < asrs.Length; n++) {
      Byte[] bytesRead = asrs[n].EndRead();
```

```
      // Теперь можно обратиться к байтовому массиву и вывести результат.
      Console.WriteLine("Number of bytes read={0}", bytesRead.Length);
      Console.WriteLine(BitConverter.ToString(bytesRead));
   }
}

private sealed class AsyncStreamRead {
   private Stream      m_stream;
   private IAsyncResult m_ar;
   private Byte[]      m_data;

   public AsyncStreamRead(Stream stream, Int32 numBytes) {
      m_stream = stream;
      m_data = new Byte[numBytes];

      // Начало асинхронной операции чтения из объекта Stream.
      m_ar = stream.BeginRead(m_data, 0, numBytes, null, null);
   }

   public Byte[] EndRead() {
      // Приостанавливаем поток до завершения асинхронной операции
      // и получения результата.
      Int32 numBytesRead = m_stream.EndRead(m_ar);

      // Больше операций нет, закрываем поток.
      m_stream.Close();

      // Изменение размера массива для экономии места.
      Array.Resize(ref m_data, numBytesRead);

      // Возвращаем байты.
      return m_data;
   }
}
```

Теперь этот код выполняет все операции чтения одновременно, и все же он мог бы быть эффективнее. После постановки в очередь всех запросов на чтение метод *ReadMultipleFiles* переходит ко второму циклу, в конце которого он последовательно вызывает метод *EndRead* для каждого потока в порядке поступления запросов на чтение. Это неэффективно, потому что время чтения данных в разных потоках неодинаково. Возможна ситуация, когда данные от второго потока поступят раньше данных от первого потока. В таком случае хорошо бы сначала обработать данные от второго потока, а потом, по мере их поступления, от первого. Модель APM поддерживает такую возможность, но метод ожидания завершения, который здесь продемонстрирован, для этого не годится.

Стыковка с использованием регулярного опроса

В интерфейсе *IAsyncResult* определены несколько свойств только для чтения:

```
public interface IAsyncResult {
   Object       AsyncState             { get; }
   WaitHandle   AsyncWaitHandle        { get; }
   Boolean      IsCompleted            { get; }
   Boolean      CompletedSynchronously { get; }
}
```

Чаще всего используется *AsyncState*. Свойства *AsyncWaitHandle* и *IsCompleted* применяются при последовательном опросе, о котором мы сейчас поговорим. Свойство *CompletedSynchronously* иногда запрашивается разработчиками, реализующими методы *BeginXxx* и *EndXxx* (но я таких никогда не встречал).

Мне не нравится этот метод стыковки, и я не советовал бы вообще им пользоваться, потому что он неэффективен. В коде, реализующем метод регулярного опроса, поток периодически осведомляется у CLR, завершен ли асинхронный запрос. Время работы потока расходуется впустую. Но, если вы готовы немного пожертвовать эффективностью ради упрощения кода, этот метод в некоторых случаях очень удобен. Проиллюстрирую метод регулярного опроса на примере.

```
public static void PollingWithIsCompleted() {
   // Открытие файла для асинхронного ввода-вывода.
   FileStream fs = new FileStream(@"C:\Boot.ini", FileMode.Open,
      FileAccess.Read, FileShare.Read, 1024,
      FileOptions.Asynchronous);

   Byte[] data = new Byte[100];

   // Инициирование асинхронной операции чтения объекта FileStream.
   IAsyncResult ar = fs.BeginRead(data, 0, data.Length, null, null);

   while (!ar.IsCompleted) {
      Console.WriteLine("Operation not completed; still waiting.");
      Thread.Sleep(10);
   }

   // Получение результата.
   // ПРИМЕЧАНИЕ: метод EndRead не приостанавливает поток.
   Int32 bytesRead = fs.EndRead(ar);

   // Все операции выполнены. Закрываем файл.
   fs.Close();

   // Теперь можно обратиться к байтовому массиву и вывести результат.
   Console.WriteLine("Number of bytes read={0}", bytesRead);
   Console.WriteLine(BitConverter.ToString(data, 0, bytesRead));
}
```

За вызовом метода *BeginRead* следует цикл, который периодически опрашивает свойство *IsCompleted* объекта *IasyncResult*, возвращающее значение *false*, если асинхронная операция еще не завершилась, иначе — *true*. В последнем случае программа переходит к выполнению цикла, в котором выводится текст: «Operation not completed; still waiting.» («Операция не завершена. Ожидание завершения операции».) Затем для порядка я вызываю метод *Sleep*, чтобы приостановить поток, иначе он будет прокручивать этот цикл, впустую расходуя процессорное время. Если же асинхронная операция завершится до того, как цикл впервые запросит свойство *IsCompleted*, оно вернет значение *true* и цикл не будет выполняться вообще.

Рано или поздно асинхронная операция будет выполнена, свойство *IsCompleted* вернет значение *true*, и цикл завершится. И тогда я вызываю метод *EndRead* для получения результатов. Но теперь я уверен, что *EndRead* сразу вернет управление программе и не приостановит вызывающий поток — ведь операция уже выполнена.

Вот еще пример использования метода регулярного опроса, в котором запрашивается свойство *AsyncWaitHandle* объекта *IAsyncResult*.

```
public static void PollingWithAsyncWaitHandle() {
   // Открываем файл для асинхронного ввода-вывода.
   FileStream fs = new FileStream(@"C:\Boot.ini", FileMode.Open,
      FileAccess.Read, FileShare.Read, 1024,
      FileOptions.Asynchronous);

   Byte[] data = new Byte[100];

   // Инициируем асинхронную операцию чтения объекта FileStream.
   IAsyncResult ar = fs.BeginRead(data, 0, data.Length, null, null);

   while (!ar.AsyncWaitHandle.WaitOne(10, false)) {
      Console.WriteLine("Operation not completed; still waiting.");
   }

   // Получаем результат. Заметьте: метод EndRead не приостанавливает поток.
   Int32 bytesRead = fs.EndRead(ar);

   // Больше операций нет. Закрываем файл.
   fs.Close();

   // Теперь можно обратиться к байтовому массиву и вывести результат.
   Console.WriteLine("Number of bytes read={0}", bytesRead);
   Console.WriteLine(BitConverter.ToString(data, 0, bytesRead));
}
```

Свойство *AsyncWaitHandle* объекта *IAsyncResult* возвращает ссылку на объект, производный от *WaitHandle*, — обычно это *System.Threading.ManualResetEvent*. В главе 24 приводится описание класса *WaitHandle* и его производных типов. Цикл в этой программе отличается от цикла из предыдущего примера тем, что вызов метода *Sleep* для потока не нужен, потому что методу *WaitOne* в качестве параметра передано значение времени приостановки потока — 10 мс.

Стыковка путем обратного вызова метода

Из всех методов стыковки в APM этот метод лучше всего подходит для создания приложения с высокими требованиями к производительности и масштабируемости. А все потому, что при использовании этого метода поток никогда не переводится в режим ожидания (в отличие от метода ожидания завершения) и не происходит напрасного расходования ресурсов процессора, когда программа периодически проверяет окончание асинхронной операции (как в методе последовательного опроса).

Принцип работы этого метода таков. Сначала запрос на асинхронный ввод-вывод ставится в очередь, а затем поток выполняет любую другую работу. Когда запрос ввода-вывода выполнится, Windows поставит рабочий элемент в очередь пула потоков CLR. В конечном итоге поток из пула выберет из очереди рабочий элемент и вызовет определенный разработчиком метод — так приложение узнает о завершении асинхронного ввода-вывода. Теперь, для получения результатов асинхронной операции в методе обратного вызова сначала вызывается метод *EndXxx*, после чего метод обратного вызова может продолжить обработку результата. Когда метод возвращает управление программе, поток возвращается в пул и готов к обслуживанию следующего рабочего элемента в очереди (или, если очередь пуста, он будет ждать его появления).

Теперь посмотрим, как это реализуется на практике. Вот прототип метода *BeginRead* объекта *FileStream*:

```
IAsyncResult BeginRead(Byte[] array, Int32 offset, Int32 numBytes,
   AsyncCallback userCallback, Object stateObject)
```

Как и у *BeginRead*, последние два параметра у любого из методов *BeginXxx* одинаковы — *SystemAsyncCallback* и *ObjectAsyncCallback*. Они представляют собой типделегат, определяемый так:

```
delegate void AsyncCallback(IAsyncResult ar);
```

Этот делегат определяет сигнатуру метода обратного вызова, которую нужно реализовать. В качестве параметра *stateObject* метода *BeginXxx* можно передать любое значение. Этот параметр просто позволяет передать некие данные от метода, ставящего операцию в очередь, методу обратного вызова, обрабатывающему завершение этой операции. Метод обратного вызова получает ссылку на объект *IAsyncResult*, он также может получить ссылку на статический объект, запросив свойство *AsyncState* объекта *IAsyncResult*. Следующий пример демонстрирует стыковку через обратный вызов метода.

```
using System;
using System.IO;
using System.Threading;

public static class Program {
   // Это статический массив, поэтому к нему могут обращаться методы Main и
   // ReadIsDone.
   private static Byte[] s_data = new Byte[100];
```

```
   public static void Main() {
      // Отображаем идентификатор потока, выполняющего метод Main.
      Console.WriteLine("Main thread ID={0}",
         Thread.CurrentThread.ManagedThreadId);

      // Открываем файл для асинхронного ввода-вывода.
      FileStream fs = new FileStream(@"C:\Boot.ini", FileMode.Open,
         FileAccess.Read, FileShare.Read, 1024,
         FileOptions.Asynchronous);

      // Начало асинхронной операции чтения в FileStream.
      // Передача FileStream (fs) методу обратного вызова (ReadIsDone).
      fs.BeginRead(s_data, 0, s_data.Length, ReadIsDone, fs);

      // Здесь могут выполняться другие команды.

      // В этой демонстрационной версии я просто приостанавливаю основной поток.
      Console.ReadLine();
   }

   private static void ReadIsDone(IAsyncResult ar) {
      // Отображаем идентификатор потока, выполняющего метод ReadIsDone.
      Console.WriteLine("ReadIsDone thread ID={0}",
         Thread.CurrentThread.ManagedThreadId);

      // Извлекаем FileStream (state) из объекта IAsyncResult.
      FileStream fs = (FileStream) ar.AsyncState;

      // Получаем результат.
      Int32 bytesRead = fs.EndRead(ar);

      // Других операций нет. Закрытие файла.
      fs.Close();

      // Теперь можно обратиться к байтовому массиву и отобразить результат.
      Console.WriteLine("Number of bytes read={0}", bytesRead);
      Console.WriteLine(BitConverter.ToString(s_data, 0, bytesRead));
   }
}
```

Скомпилировав и запустив эту программу на своем компьютере, я получил следующее (на других компьютерах получится не точно такой, но аналогичный результат):

```
Main thread ID=1
ReadIsDone thread ID=4
Number of bytes read=100
5B-62-6F-6F-74-20-6C-6F-61-64-65-72-5D-0D-0A-74-69-6D-65-6F-
75-74-3D-33-30-0D-0A-64-65-66-61-75-6C-74-3D-6D-75-6C-74-69-
28-30-29-64-69-73-6B-28-30-29-72-64-69-73-6B-28-30-29-70-61-
72-74-69-74-69-6F-6E-28-31-29-5C-57-49-4E-44-4F-57-53-0D-0A-
5B-6F-70-65-72-61-74-69-6E-67-20-73-79-73-74-65-6D-73-5D-0D
```

Прежде всего обратите внимание, что метод *Main* выполнялся основным потоком с идентификатором 1, а *ReadIsDone* — потоком из пула с идентификатором 4. Таким образом, в выполнении этой программы участвовали два различных потока. Во-вторых, объект *IAsyncResult*, возвращаемый методом *BeginRead*, не сохраняется в переменной в методе *Main*. Это и не нужно, поскольку CLR сама передает объект *IAsyncResult* методу обратного вызова. В-третьих, я передал имя метода обратного вызова в качестве четвертого параметра методу *BeginRead*. В-четвертых, я передаю *fs* как последний параметр методу *BeginRead*. Так я передаю *FileStream* методу обратного вызова. Метод обратного вызова получает ссылку на *FileStream*, запрашивая свойство *AsyncState* переданного объекта *IAsyncResult*.

Я вполне доволен этой программой за одним исключением: мне пришлось сделать поле *Byte[]* (с именем *s_data*) статическим, чтобы к нему можно было обращаться из методов *Main* и *ReadIsDone*. Для примера этого достаточно, но обычно я не использую такую архитектуру, потому что она не позволяет расширять программу. Когда-нибудь мне нужно будет выполнить несколько запросов ввода-вывода — тогда лучше всего динамически создавать *Byte[]* для каждого запроса. Есть два пути улучшения этой архитектуры.

- Можно определить другой класс, содержащий поля *Byte[]* и *FileStream*, создать экземпляр этого класса, инициализировать поля и передать эту ссылку как последний параметр метода *BeginRead*. Тогда метод обратного вызова сможет обращаться к обоим источникам данных и использовать их внутри метода.
- Можно использовать анонимные методы в C# (см. главу 15), чтобы компилятор C# сгенерировал этот код автоматически.

Вот как эта программа выглядит при использовании анонимных методов C#:

```
public static void Main() {
   // Отображаем идентификатор потока, выполняющего метод Main.
   Console.WriteLine("Main thread ID={0}",
      Thread.CurrentThread.ManagedThreadId);

   // Открываем файл для асинхронного ввода-вывода.
   FileStream fs = new FileStream(@"C:\Boot.ini", FileMode.Open,
      FileAccess.Read, FileShare.Read, 1024,
      FileOptions.Asynchronous);

   Byte[] data = new Byte[100];

   // Начало асинхронной операции чтения в FileStream.
   // Передаем FileStream (fs) методу обратного вызова (ReadIsDone).
   fs.BeginRead(data, 0, data.Length,
      delegate(IAsyncResult ar)
      {
         // Отображаем идентификатор потока, выполняющего метод ReadIsDone.
         Console.WriteLine("ReadIsDone thread ID={0}",
            Thread.CurrentThread.ManagedThreadId);

         // Получаем результат.
         Int32 bytesRead = fs.EndRead(ar);
```

```
            // Больше операций нет, закрываем файл.
            fs.Close();

            // Теперь можно обратиться к байтовому массиву и отобразить результат.
            Console.WriteLine("Number of bytes read={0}", bytesRead);
            Console.WriteLine(BitConverter.ToString(data, 0, bytesRead));

        }, null);

    // Здесь может выполняться другой код.

    // В этой демонстрационной версии программы я просто
    // приостанавливаю основной поток.
    Console.ReadLine();
}
```

В этом варианте программы я просто передал *null* в качестве последнего параметра метода *BeginRead*. Это возможно из-за того, что анонимный метод может обращаться к любой локальной переменной (в том числе *data* и *fs*), определенной в методе *Main*. Выбирайте, какой вариант программы вам больше по душе: с использованием анонимного метода или с явной реализацией вспомогательного класса и применением локальных переменных в качестве его полей.

Примечание Во многих примерах этой главы асинхронная операция инициируется из метода *Main* приложения. Во многих примерах мне также пришлось вызывать метод *ReadLine* объекта *Console*, чтобы *Main* не возвращал управление, иначе прекратится выполнение всего процесса, пула потоков и незавершенных асинхронных операций.

Во многих приложениях асинхронная операция инициируется из метода, выполняемого потоком из пула. Затем метод должен вернуть управление приложению, чтобы поток возвратился в пул. Далее этот поток может использоваться для вызова метода при завершении любой асинхронной операции или для обработки завершения асинхронной операции, только что добавленной в очередь. Часто при вызове метода для обработки завершения одной асинхронной операции этот же метод инициирует другую асинхронную операцию.

Впечатляет? Так небольшое число потоков позволяет в кратчайшие сроки и с минимальным потреблением ресурсов выполнить большой объем работы.

В разделе «Стыковка с использованием ожидания завершения» я показал метод *ReadMultipleFiles*, который считывает данные из нескольких потоков. Этот метод был недостаточно эффективен, поскольку он обрабатывал данные в порядке постановки в очередь, а не в порядке их поступления. Я изменил эту программу, воспользовавшись методом обратного вызова. Новая версия программы, по сравнению с предыдущей, более эффективна и является лучшим применением модели APM для достижения высокой производительности и расширяемости.

```
private static void ReadMultipleFiles(params String[] pathnames) {
   for (Int32 n = 0; n < pathnames.Length; n++) {
      // Открываем файл для выполнения асинхронного ввода-вывода.
      Stream stream = new FileStream(pathnames[n], FileMode.Open,
         FileAccess.Read, FileShare.Read, 1024,
         FileOptions.Asynchronous);

      // Начало асинхронной операции чтения в Stream.
      new AsyncStreamRead(stream, 100,
         delegate(Byte[] data)
         {
            // Обрабатываем данные.
            Console.WriteLine("Number of bytes read={0}", data.Length);
            Console.WriteLine(BitConverter.ToString(data));
         });
   }

   // Все потоки были открыты, все запросы на чтение были поставлены в очередь.
   // Все они выполняются одновременно до своего завершения.

   // Здесь основной поток может выполнять другую работу…

   // В этой демонстрационной версии я просто приостановил основной поток.
   Console.WriteLine("Hit <Enter> to end this program...");
   Console.ReadLine();
}

private delegate void StreamBytesRead(Byte[] streamData);

private sealed class AsyncStreamRead {
   private Stream m_stream;
   private Byte[] m_data;
   StreamBytesRead m_callback;

   public AsyncStreamRead(Stream stream, Int32 numBytes,
      StreamBytesRead callback) {
      m_stream = stream;
      m_data = new Byte[numBytes];
      m_callback = callback;

      // Начало асинхронной операции чтения в Stream.
      stream.BeginRead(m_data, 0, numBytes, ReadIsDone, null);
   }

   // Вызывается при завершении операции ввода-вывода.
   private void ReadIsDone(IAsyncResult ar) {
      Int32 numBytesRead = m_stream.EndRead(ar);

      // Больше операций нет, закрываем поток.
      m_stream.Close();
```

```
      // Изменяем размеры массива для экономии места.
      Array.Resize(ref m_data, numBytesRead);

      // Вызываем метод обратного вызова приложения.
      m_callback(m_data);
   }
}
```

Использование модели APM для выполнения асинхронных вычислительных операций

До сих пор все рассуждения о модели APM сводились к ее использованию для выполнения асинхронных операций ввода-вывода. Эта модель очень эффективна: ведь потокам не приходится работать или ждать завершения ввода-вывода. Ее также можно использовать для вычислительных операций. Однако вычислительные операции активно загружают потоки работой, поэтому они не столь эффективны. Ничего не поделаешь, компьютеры не только постоянно выполняют ввод-вывод — им приходится также обрабатывать данные.

Используя модель APM, можно вызвать любой метод, но сначала нужно определить делегат с такой же сигнатурой, как и у вызываемого метода. Допустим, нужно вызвать метод, суммирующий числа от 1 до *n*. Эта вычислительная задача (не выполняющая ввода-вывода) потребует много времени, если *n* — большое число[3]. Так выглядит метод *Sum*:

```
private static UInt64 Sum(UInt64 n) {
   UInt64 sum = 0;
   for (UInt64 i = 1; i <= n; i++) {
      checked {
         // Я использую оператор checked, чтобы в случае, если тип суммы
         // отличен от UInt65, было сгенерировано исключение OverflowExcept.
         sum += i;
      }
   }
   return sum;
}
```

Если *n* — большое число, метод *Sum* будет выполняться очень долго. Чтобы интерфейс приложения откликался на действия пользователя и чтобы можно было использовать другие процессоры системы, я выполню этот метод асинхронно. Для этого сначала нужно определить делегат с такой же сигнатурой, что и у метода, который будет вызываться асинхронно (*Sum*):

```
internal delegate UInt64 SumDelegate(UInt64 n);
```

Вспомните из главы 15, посвященной делегатам: компилятор C# скомпилирует эту строку кода в определение класса, которое логически будет выглядеть так:

---

[3] Я в курсе, что сумму можно быстро вычислить по формуле n(n+1)/2 для любого n. Но давайте на этот раз забудем о ней и выберем способ решения задачи «в лоб», который требует много времени.

```
internal sealed class SumDelegate : MulticastDelegate {
   public SumDelegate(Object object, IntPtr method);
   public UInt64 Invoke(UInt64 n);
   public IAsyncResult BeginInvoke(UInt64 n,
      AsyncCallback callback, Object object);
   public UInt64 EndInvoke(IAsyncResult result);
}
```

При определении делегата в исходном тексте на C# компилятор всегда создает класс, в котором есть методы *BeginInvoke* и *EndInvoke*. У первого те же параметры, что и в определении делегата, а также два дополнительных параметра — *AsyncCallback* и *Object*. Все методы *BeginInvoke* возвращают *IAsyncResult*. У метода *EndInvoke* один параметр; *IAsyncResult*; метод *EndInvoke* и сигнатура делегата возвращают один тип данных.

Теперь понятно, что использование делегата для выполнения вычислительной операции — простая задача, потому что она укладывается в рамки модели APM. При асинхронном вызове метода с помощью делегата можно воспользоваться любым из методов стыковки: ожидания завершения, регулярного опроса или обратного вызова метода. Следующая программа демонстрирует асинхронный вызов *Sum* с помощью метода обратного вызова.

```
public static void Main() {
   // Инициализируем переменную делегата, ссылающуюся
   // на метод, который нужно вызвать асинхронно.
   SumDelegate sumDelegate = Sum;

   // Вызываем метод с помощью потока из пула.
   sumDelegate.BeginInvoke(1000000000, SumIsDone, sumDelegate);

   // Здесь можно выполнить другой код…

   // В этой демонстрационной версии программы я просто приостановил основной поток.
   Console.ReadLine();
}
```

Переменная *sumDelegate* сначала инициализируется ссылкой на метод, который нужно вызвать асинхронно. Затем асинхронный вызов метода выполняется вызовом метода *BeginInvoke*. При этом CLR создает объект *IAsyncResult*, идентифицирующий асинхронную операцию. Как уже говорилось, операции ввода-вывода ставятся в очередь драйвера устройства Windows. А метод *BeginInvoke* делегата ставит вычислительные операции в очередь пула потоков CLR, вызывая метод *QueueUserWorkItem* объекта *ThreadPool*. И, наконец, *BeginInvoke* возвращает объект *IAsyncResult* вызывающему коду. Теперь этот объект можно использовать так же, как и при выполнении асинхронной операции ввода-вывода.

После того как метод *BeginInvoke* поставит операцию в очередь в пул потоков CLR, один поток из пула активизируется, извлекает из очереди рабочий элемент и вызывает вычислительный метод (например, *Sum*). Обычно, завершив выполнение метода, поток возвращается в пул. Но в данном примере при вызове *BeginInvoke* ему было передано в качестве предпоследнего параметра имя метода *SumIsDone*. Поэтому, когда *Sum* возвращает управление приложению, поток из пула не

возвращается обратно в пул, а вызывает метод *SumIsDone*. Иначе говоря, обратный вызов инициируется при завершении вычислительной операции так же, как и при завершении операции ввода-вывода. Метод *SumIsDone* выглядит так:

```
private static void SumIsDone(IAsyncResult ar) {
   // Извлекаем SumDelegate (state) из объекта IAsyncResult.
   SumDelegate sumDelegate = (SumDelegate) ar.AsyncState;

   // Получаем результат.
   UInt64 sum = sumDelegate.EndInvoke(ar);

   // Выводим результат.
   Console.WriteLine("Sum={0}", sum);
}
```

Модель APM и исключения

При любом вызове метода *BeginXxx* может возникнуть исключение, на основании которого вполне логично предположить, что асинхронная операция не была поставлена в очередь.

При обработке запроса на асинхронный ввод-вывод драйвером устройства Windows может произойти сбой, и Windows должна сообщить об этом приложению. В частности, Windows ожидает поступление данных из сети лишь в течение указанного вызывающим кодом времени. Если данные не поступят вовремя, ОС должна сообщить пользователю об ошибке асинхронной операции. Для этого Windows передает уведомление в пул потоков CLR. Приложение «стыкуется» с результатами операции, вызывая метод *EndXxx*. Обычно *EndXxx* возвращает результат операции в приложение, но, если операция не была выполнена, *EndXxx* сгенерирует исключение. Тип исключения зависит от причины сбоя операции.

Аналогично, при использовании APM-модели для выполнения асинхронной вычислительной операции метод (выполняемый потоком из пула) может выполнить код, который приведет к генерации исключения. Если исключение останется необработанным, CLR его автоматически перехватит. При использовании метода стыковки путем обратного вызова метода поток из пула вызывает метод обратного вызова. При вызове метода *EndInvoke* в методе обратного вызова ранее перехваченное исключение генерируется повторно. Если оно опять останется необработанным, CLR уничтожит процесс (если это не противоречит политике хоста). Но исключение можно перехватить и обработать — это позволит корректно продолжить работу приложения.

Далее приведен метод *SumIsDone* (из предыдущего раздела) с изменениями: он перехватывает возможные исключения *OverflowException* и корректно восстанавливает работу приложения.

```
private static void SumIsDone(IAsyncResult ar) {
   // Извлекаем SumDelegate (state) из объекта IAsyncResult.
   SumDelegate sumDelegate = (SumDelegate)ar.AsyncState;

   try {
      // Получаем результат. Может появиться исключение OverflowException.
      UInt64 sum = sumDelegate.EndInvoke(ar);
```

```
        // Отображаем результат.
        Console.WriteLine("Sum={0}", sum);
    }
    catch (OverflowException) {
        // EndInvoke  сгенерировал исключение OverflowException.
        // Корректное восстановление приложения.
        Console.WriteLine("Sum can't be shown: number is too big.");
    }
}
```

Важные замечания о модели APM

Я большой поклонник модели APM, но должен признать, что у нее есть недостатки. Хотелось бы, чтобы Microsoft устранила часть из них или хотя бы показала разработчикам, что с ними делать. Рассмотрим их подробнее.

Чтобы избежать утечки ресурсов, нужно вызывать метод *EndXxx*. Некоторые разработчики создают код, вызывающий метод *BeginXxx* для записи данных в устройство; если после записи данных других операций нет, разработчики обычно забывают вызвать метод *EndXxx*. Но это делать обязательно по двум причинам. Во-первых, при инициации асинхронной операции CLR выделяет под нее внутренние ресурсы. При завершении операции CLR резервирует эти ресурсы до вызова метода *EndXxx*. Если его не вызвать, эти ресурсы освободятся лишь при завершении процесса. Во-вторых, при инициации асинхронной операции нельзя предсказать ее исход — успех или сбой. Узнать это можно, только вызвав метод *EndXxx* и посмотрев возвращаемое значение или сгенерированное им исключение.

В любой асинхронной операции не следует вызывать *EndXxx* более одного раза. При вызове метода *EndXxx* он обращается к внутренним ресурсам, а затем освобождает их. Результат повторного вызова *EndXxx* непредсказуем — ведь ресурсы уже были освобождены. На практике многократный вызов *EndXxx* может иметь разные последствия: это зависит от того, как был написан класс, реализующий интерфейс *IAsyncResult*. Поскольку Microsoft не предоставила разработчикам указаний по этому поводу, каждый реализует его по-своему. Единственное, в чем можно быть уверенным — однократный вызов *EndXxx* точно сработает.

Методы *BeginXxx* и *EndXxx* должны вызываться с одним и тем же объектом. Так, не следует создавать делегат и вызывать его метод *BeginInvoke*, а позже создавать еще один делегат (того же типа, ссылающийся на тот же объект/метод) и использовать его для вызова *EndInvoke*. На первый взгляд кажется, что такая программа должна работать (потому что оба делегата одинаковы), но это не так, потому что в объекте *IAsyncResult* хранится ссылка на исходный объект, использованный при вызове *BeginInvoke*, и если они не совпадают, *EndInvoke* сгенерирует исключение *InvalidOperationExcept* с таким сообщением: «The IAsyncResult object provided does not match this delegate» («Представленный объект *IAsyncResult* не соответствует этому делегату»). Все же, использование разных объектов для вызова методов *BeginInvoke* и *EndInvoke* может сработать для некоторых типов объектов, если это предусмотрено их реализацией.

Параметры методов *BeginXxx* и *EndXxx* будут немного отличаться от стандартов, описанных в этой главе, если вариант этого метода для не-асинхронных опе-

раций использует какой-либо из параметров *out/ref* и если у него есть параметр, помеченный ключевым словом *params*. Такое бывает очень редко, поэтому пример я приводить не буду, но имейте это в виду. Тогда вы легко разберетесь, как правильно вызвать эти методы.

В настоящее время способов отмены незавершенной асинхронной операции нет. Многие разработчики обрадовались бы такой возможности, но ее очень сложно реализовать. И если вы запросили 1000 байт данных с сервера, а потом передумали, то никак нельзя заставить сервер «забыть» о вашем запросе. Остается получить все запрошенные данные, а потом выкинуть их. Нужно учесть и конкуренцию запросов: ваш запрос на отмену может прийти в момент считывания последнего байта исходного запроса. Какова должна быть реакция приложения? Поэтому такой вариант развития событий нужно учесть в своем приложении и решить, удалять ли все данные или закончить считывание. Некоторые методы *BeginXxx* возвращают объекты, реализующие интерфейс *IAsyncResult*, а также поддерживают методы отмены. В таком случае операцию можно отменить. В документации к методу *BeginXxx* или возвращаемому им классу можно узнать, поддерживает ли он отмену операций.

При вызове *BeginXxx* он создает экземпляр типа, реализующего интерфейс *IAsyncResult*. Это значит, что для каждой выполняемой асинхронной операции создается объект. Это создает дополнительную нагрузку и приводит к появлению большего числа объектов в куче, что требует более частого сбора мусора. Результат — снижение производительности приложения. Так что, если вы уверены, что операции ввода-вывода выполнятся очень быстро, лучше выполнять их синхронно. Многие разработчики (в том числе и ваш покорный слуга) предпочли бы, чтобы в такой ситуации модель APM возвращала значимые типы или поддерживала другой, более простой способ выявления поставленной в очередь асинхронной операции. Может быть, Microsoft когда-нибудь улучшит CLR, реализовав такую функциональность.

В интерфейсе Win32 API есть много функций для ввода-вывода, но, увы, значительное их число не позволяет выполнять ввод-вывод асинхронно. Например, метод *CreateFile* (вызываемый конструктором *FileStream*) всегда выполняется синхронно. При попытке создать/открыть файл на сетевом сервере *CreateFile* вернет управление приложению через несколько секунд — вызывающий поток все это время бездействует. Если производительность и расширяемость приложения важны, лучше вызвать Win32-функцию, которая позволяет создавать/открывать файл асинхронно, чтобы поток не просто простаивал в ожидании ответа от сервера. К сожалению, в Win32 нет функции, подобной *CreateFile*, которая позволила бы это сделать. Поэтому в FCL нет эффективного способа асинхронного открытия файла.

Любой метод можно вызвать асинхронно с помощью метода *BeginInvoke* делегата, но в таком случае используется поток и эффективность падает. Кроме того, делегат нельзя использовать для вызова конструктора. Поэтому единственный способ обновить объект *FileStream* асинхронно — вызвать асинхронно метод, который обновит объект *FileStream*. Windows не предоставляет функций для асинхронного доступа к реестру, журналам событий, файлам и подпапкам в каталоге, а также для асинхронного изменения свойств файла/папки. Это далеко не полный список.

В Windows окно всегда создается потоком, который должен обрабатывать все действия для этого окна. Это объясняется тем, что 16-разрядные версии Windows были однопоточными и для обеспечения обратной совместимости в 32- и 64-разрядных версиях сохранили однопоточную архитектуру для обработки оконных операций, в том числе WM_MOVE, WM_SIZE, WM_PAINT, WM_CLOSE и так далее. В приложениях Windows Forms часто используются асинхронные операции, описанные в этой главе. Однако Windows Forms создано на основе Windows, поэтому поток из пула не может напрямую взаимодействовать с окном, а точнее с классом, производным от *System.Windows.Forms.Control*.

К счастью, в классе *System.Windows.Forms.Control* есть три метода — *Invoke*, *BeginInvoke* и *EndInvoke*, которые можно вызвать из любого потока (в том числе потока из пула), чтобы выполнить маршалинг операции из вызывающего потока в поток, создавший окно. Метод *Invoke* класса *Control* вызывает Win32-метод *SendMessage*, чтобы поток окна выполнил задачу синхронно. Метод *BeginInvoke* класса *Control* вызывает Win32-метод *PostMessage*, чтобы поток окна выполнил задачу асинхронно. Если вызывающему потоку нужно узнать, когда поток окна завершил выполнение задачи, он может вызвать метод *EndInvoke* класса *Control*. Если это знать необязательно, это один из тех редких случаев, когда метод *EndInvoke* вызывать не нужно.

Контексты выполнения

У каждого потока есть связанный с ним контекст выполнения, который включает в себя следующее:

- **параметры безопасности** (сжатый стек разрешений, состоящий из объектов *IPermission*, свойства *Principal* объекта *Thread* и маркера потока Windows);
- **параметры локализации** (свойства *CurrentCulture* и *CurrentUICulture* объекта *Thread* и региональные стандарты потока Windows);
- **параметры транзакций** (статическое свойство *Current* объекта *System.Transactions.Transaction*).

Когда поток выполняет код, значения параметров контекста выполнения потока оказывают влияние на некоторые операции. В идеале всякий раз, когда поток использует другой (вспомогательный) поток для выполнения некоторых задач, контекст выполнения первого потока должен копироваться во вспомогательный поток. Так, все операции, выполненные вспомогательным потоком, будут выполняться с теми же параметрами безопасности, региональных стандартов и транзакций, что и первый поток.

По умолчанию CLR автоматически копирует контекст выполнения первого потока во все вспомогательные потоки. Это гарантирует безопасность, но в ущерб производительности, потому что в контексте выполнения содержится много информации. Сбор всей информации и ее копирование во вспомогательные потоки занимает немало времени, особенно когда CLR должна пройтись по стеку первого потока и сжать все объекты-разрешения безопасности.

Класс *ExecutionContext* в пространстве имен *System.Threading* позволяет управлять копированием контекста выполнения потока. Но использовать этот класс для запрета копирования контекста исполнения опасно, потому что это дает вспомогательному потоку больше привилегий. Поэтому многие методы класса *Execution-*

Context требуют, чтобы у вызывающей сборки было разрешение *SecurityPermission* с установленным флагом *SecurityPermissionFlag.Infrastructure*. Более того, когда контекст выполнения не копируется во вспомогательный поток, вспомогательный поток выполняется в рамках последнего сохраненного в нем контекста выполнения. Поэтому, если копирование контекста выполнения отключено, поток не должен выполнять никакой код, основываясь на своем текущем контексте выполнения (безопасности, локализации или транзакции).

Помимо запрета копирования контекста выполнения потока класс *ExectionContext* позволяет перехватывать контекст выполнения потока в произвольный момент выполнения программы. Затем в другой момент выполнения можно вызвать метод, используя перехваченный контекст выполнения. В следующем коде показано, как управлять копированием контекста выполнения в CLR. В нем видно, как изменять контекст выполнения, как предотвращать копирование контекста выполнения потока в другой поток (для повышения производительности), как перехватывать контекст выполнения потока и как использовать полученный контекст для вызова другого метода. (Для работы этой программы на компьютере должен быть файл ReadME.txt в корне диска C:.)

```
using System;
using System.IO;
using System.Security;
using System.Threading;
using System.Security.Permissions;

public static class Program {
   public static void Main() {
      // Демонстрируем, что метод работает, как ожидается.
      AttemptAccess("Default context");

      // Изменяем контекст выполнения потока,
      // чтобы запретить любой доступ к файлу.
      new FileIOPermission(PermissionState.Unrestricted).Deny();

      // Демонстрируем, что метод не работает
      // из-за изменения контекста выполнения.
      AttemptAccess("No file permissions");

      ECFlowing();
      ECFlowingSuppressed();

      // Перехватываем текущий контекст выполнения потока.
      ExecutionContext ec = ExecutionContext.Capture();

      // Изменяем контекст выполнения потока,
      // чтобы разрешить доступ к файлу.
      SecurityPermission.RevertDeny();

      // Демонстрируем нормальную работу метода.
      AttemptAccess("Default context again");
```

```
      ECCaptureAndRun(ec);
   }

   private static void ECFlowing() {
      // Инициализируем переменную делегата ссылкой
      // на вызываемый асинхронно метод.
      WaitCallback wc = AttemptAccess;

      // Используем поток из пула для попытки доступа.
      // Метод EndInvoke возвращает управление,
      // когда поток из пула возвращается в пул.
      wc.EndInvoke(wc.BeginInvoke("ECFlowing", null, null));
   }

   private static void ECFlowingSuppressed() {
      // Инициализируем переменную делегата ссылкой на метод,
      // вызываемый асинхронно.
      WaitCallback wc = AttemptAccess;

      // Временно отключаем копирование контекста выполнения потока
      // во вспомогательные потоки.
      using (AsyncFlowControl afc = ExecutionContext.SuppressFlow()) {
         wc.EndInvoke(wc.BeginInvoke("ECFlowingSuppressed", null, null));
      }
   }

   private static void ECCaptureAndRun(ExecutionContext ec) {
      // Вызываем AttemptAccess с применением
      // ранее захваченного контекста выполнения.
      ExecutionContext.Run(ec, AttemptAccess,
         "ECCaptureAndRun with Run");
   }

   private static void AttemptAccess(Object test) {
      // Допустим, метод дал сбой.
      Boolean success = false;
      try {
         // Пытаемся получить атрибуты файла.
         File.GetAttributes(@"C:\ReadMe.txt");

         // Если атрибуты получены, метод выполнился успешно.
         success = true;
      }
      catch (SecurityException) {
         // Метод дал сбой из-за недостаточной безопасности
         // в контексте выполнения потока.
      }

      // Показываем результат попытки доступа: успех или ошибка.
      Console.WriteLine("{0}: {1}", test, success);
   }
}
```

Скомпилировав и запустив эту программу, получим следующее:

```
Default context: True
No file permissions: False
ECFlowing: False
ECFlowingSuppressed: True
Default context again: True
ECCaptureAndRun with Run: False
```

ГЛАВА 24

Синхронизация потоков

Как говорилось в главе 23, приложения отличаются высокой производительностью, если потоки приложений не ожидают завершения операций. Поэтому лучше всего реализовывать методы, оперирующие собственными данными, и избегать методов, обращающихся к каким-либо совместно используемым данным. Однако редко бывает так, что поток работает только со своими данными, не обращаясь к общим.

Все потоки в системе должны иметь доступ к системным ресурсам: кучам, последовательным портам, файлам, окнам и многому другому. Если один поток запрашивает исключительный доступ к ресурсу, остальные потоки, которым тоже нужен этот ресурс, не смогут завершить свою работу. С другой стороны, нельзя просто разрешать любому потоку работать с любым ресурсом в любое время. Представьте, что поток выполняет запись в блок памяти, когда другой поток считывает информацию из этого блока. Это равносильно чтению книги в то время, когда кто-то изменяет ее текст. Информация смешается, и ничего хорошего из этого не получится.

Чтобы предотвратить повреждение ресурса по причине одновременного доступа многих потоков, нужно использовать конструкции синхронизации потоков. В Windows и CLR много таких конструкций, и у каждой свои преимущества и недостатки. Сейчас мы поговорим о них поподробнее. В этой главе я не буду вдаваться в детали внутренней реализации конструкций синхронизации потоков, поскольку большинству программистов нет до этого дела. Но, если вам все же интересно понять, как они реализованы, советую обратиться к статьям в моей колонке Concurrent Affairs в библиотеке MSDN (например, *http://msdn.microsoft.com/msdnmag/issues/05/10/ConcurrentAffairs/*). Также предлагаю собственную библиотеку Power Threading Library (ее можно скачать с сайта *http://wintellect.com*), содержащую много созданных мной конструкций синхронизации потоков. В библиотеке есть исходный код этих конструкций.

Многие из конструкций синхронизации потоков в CLR в действительности всего лишь объектно-ориентированные оболочки классов, построенные на базе конструкций синхронизации потоков Win32. В конце концов, потоки CLR — это потоки Windows, а это значит, что именно Windows планирует и контролирует их синхронизацию. Конструкции синхронизации потоков Windows существуют с 1992 года, и о них написано несметное множество материалов. (В моей книге Programming Applications for Microsoft Windows (Microsoft Press, 2000) этой теме по-

священо несколько глав.) Поэтому в этой главе они рассматриваются лишь поверхностно.

Примечание При построении повторно используемой библиотеки классов нужно позаботиться, чтобы все статические методы типов обеспечивали *безопасность потоков*, то есть чтобы при одновременном вызове несколькими потоками статического метода типа состояние типа не повреждалось. Для этого нужно использовать одну из конструкций синхронизации потоков, описанных в этой главе. Обеспечивать безопасность потоков при работе статических методов в многопоточной среде обязательно, потому что пользователи классов просто не в состоянии реализовать это в своем коде.

Представьте, что статический метод *Load* одновременно пытаются вызвать код библиотечной сборки этой сборки и другой код в исполняемой сборке. Обеим сборкам придется согласовать используемую конструкцию синхронизации потока, и они обе должны «уметь» обнаруживать ее. На самом деле, все сборки в AppDomain должны использовать одну конструкцию и иметь возможность найти ее. Поскольку CLR не предоставляет встроенного механизма синхронизации, задача решается в типе *Assembly* за счет выполнения собственной конструкции синхронизации потоков (которая легко обнаруживается, так как создает ее сам тип).

С другой стороны, библиотека повторно используемых классов не обязана обеспечивать безопасность потоков во всех экземплярных методах типа. Причина в том, что обеспечение безопасности работы с потоками замедляет работу методов, а большинство объектов используется одним потоком; обеспечение безопасности потоками значительно ударит по производительности. Кроме того, когда код создает объект, ни у какого другого кода нет доступа к этому объекту, если только ссылка на него каким-то образом не передана в этот код. Любой код, передающий ссылку на объект, может также передавать информацию о конструкции синхронизации потоков, чтобы код, исполняемый другими потоками, мог обратиться к объекту, обеспечивая безопасность потоков.

Конечно, Microsoft следует этим правилам. Все статические методы типов из библиотеки Framework Class Library обеспечивают безопасность потоков, но нет никаких гарантий, что такую безопасность обеспечивают экземплярные методы типов.

Целостность памяти, временный доступ к памяти и volatile-поля

Для повышения производительности при частых обращениях к памяти на современных процессорах предусмотрена встроенная кеш-память. Обращение к этой памяти выполняется чрезвычайно быстро, особенно по сравнению со скоростью доступа к памяти на материнской плате. Когда поток впервые считывает значение из памяти, процессор извлекает нужное значение из оперативной памяти и размещает в собственном встроенном кеше. В действительности, производительность

дополнительно повышается за счет того, что за одно обращение процессор извлекает несколько расположенных рядом байт (это называют строкой кеша), поскольку приложение обычно считывает из памяти байты, расположенные рядом. Если один из расположенных рядом байт тоже считывается, он уже может быть в кеше; если это так, обращения к оперативной памяти не происходит и операция выполняется быстрее. Когда же поток выполняет запись в память, процессор просто обновляет соответствующий байт в кеше и не записывает обновленное значение в память на материнской плате. Это тоже способствует повышению производительности. В конечном итоге процессор сбросит все данные из кеша, но попозже.

Если приложение работает на однопроцессорном компьютере (на котором может быть установлен процессор Hyperthreading), разработчику приложения эта информация ни к чему, поскольку это никак не влияет на работу программы. Тем не менее производительность повышается за счет использования кеш-памяти. Если на компьютере один процессор Hyperthreading, никакого видимого эффекта тоже не будет, так как два логических процессора используют один процессорный кеш. Однако, если приложение работает на многопроцессорной системе или системе с двухъядерным процессором, роль процессорного кеша может стать весьма заметной, и разработчикам следует принимать это во внимание при написании кода. Конечно, даже на таких процессорах разница заметна, только если несколько потоков обращается к одним и тем же байтам памяти. Если они обращаются к разным байтам, особой разницы не будет.

Вот пример:

```
internal sealed class CacheCoherencyProblem {
   private Byte  m_initialized = 0;
   private Int32 m_value = 0;

   // Этот метод выполняется одним потоком.
   public void Thread1() {
      m_value = 5;
      m_initialized = 1;
   }

   // Этот метод выполняется другим потоком.
   public void Thread2() {
      if (m_initialized == 1) {
         // Эта строка может выполняться и отображать значение 0.
         Console.WriteLine(m_value);
      }
   }
}
```

Теперь допустим, что экземпляр этого класса создается на компьютере с двумя процессорами. Первый процессор выполняет метод *Thread1*, а второй — метод *Thread2*. Представьте, что выполнение происходит следующим образом.

- Поток второго процессора считывает из памяти байт, и этот байт располагается сразу перед байтами поля *m_value* объекта. В этом случае считывается целая строка кеша, а значит, байты поля *m_value* попадают в кеш второго процессо-

ра. С точки зрения программиста, процессор считал значение из этого поля прежде, чем программа запросила считывание этого поля.

- Поток первого процессора выполняет метод *Thread1*, который присваивает полю *m_value* значение 5, но это изменение может произойти в кеше первого процессора. В конечном счете, это изменение будет записано в оперативную память. С точки зрения программиста, процессор записывает значение этого поля в память намного позже того момента, когда код запросил изменение поля.
- Поток первого процессора продолжает выполнять метод *Thread1* и изменяет значение поля *m_initialized* на 1. Это также происходит в кеше первого процессора, но байты поля *m_initialized* находятся в другой строке кеша, и поэтому могут быть сброшены в оперативную память.
- Поток второго процессора начинает выполнение метода *Thread2*, который сначала запрашивает значение поля *m_initialized*. Поскольку байт этого значения в кеше второго процессора нет, они считываются из оперативной памяти. Поле *m_initialized* будет содержать значение 1, поэтому будет выполнено условие оператора *if* и его тело будет выполняться.
- Поток второго процессора продолжает выполнять метод *Thread2*, который считывает значение поля *m_value*. Поскольку байты этого поля уже находятся в кеше второго процессора, они считываются оттуда, а не из оперативной памяти, и значение поля *m_value* считается равным 0!

Надеюсь, вы поняли, в чем проблема. Подведем итог: кеш процессоров улучшает производительность, но может привести к тому, что различные потоки будут использовать разные значения одной и той же переменной. Согласованность значений в кеше и оперативной памяти — серьезная проблема, и есть разные способы ее решения, но все они сильно снижают производительность. Лучше всего реализовать методы так, чтобы предотвратить одновременный доступ нескольких потоков к общим данным. Например, попытаться реализовать методы, которые обращаются только к параметрам и локальным переменным, чтобы другие потоки не имели доступа к этим переменным. Конечно, сама по себе переменная ссылочного типа обеспечивает безопасность потоков, но объект, на который она ссылается, — нет, поскольку иметь ссылку на него могут несколько потоков сразу. Более того, если несколько потоков обращаются к общим данным в режиме только для чтения, проблем не возникает. Проблема согласованности кеша существует только для общих данных, которые могут изменяться потоками.

Дополнительно ухудшает ситуацию то, что компиляторы C# и JIT при компиляции кода могут изменить порядок операторов и они могут выполняются не в том порядке, в каком были расположены в исходном коде. Да и сам процессор может вносить коррекцию в очередность выполнения машинных команд. Но при всем при этом компиляторы C# и JIT и процессор действуют так, чтобы все работало, как запланировал программист, но только с расчетом на один поток — когда несколько потоков обращается к общей памяти (как показано в предыдущем примере), выполнение операторов в не предусмотренном программистом порядке может создать проблемы.

Скорее всего, вы скажете, что никогда не видели, чтобы в вашей программе возникали проблемы из-за изменения порядка выполнения операторов, и это может

оказаться чистой правдой. Это, без сомнения, так, если программа выполнялась на однопроцессорной системе или даже на компьютере с процессором Hyper-threading. Но в многопроцессорной системе или на компьютере с двухъядерным процессором неполадки такого рода вполне возможны. Кроме того, разные процессоры по-разному ведут себя в отношении согласованности кеша. Например, в процессорах x86 для обеспечения согласованности сделано многое. Поскольку архитектура x64 обратно совместима с архитектурой x86, процессоры x64 также обеспечивают согласованность кеша. Так что, если выполнять методы класса *CacheCoherencyProblem* на компьютерах с процессорами x86 или x64, проблем не будет. Метод *Thread2* либо не вернет ничего, либо вернет значение 5.

Однако процессоры новой архитектуры (например, IA64) проектировались для полноценного использования того факта, что у каждого процессора собственный кеш, и для повышения производительности согласованность кеша практически не отслеживается. Процессоры IA64 поддерживают обычные машинные команды для считывания значения из памяти в регистр и для записи значения регистра в память. Но есть и модифицированная версия команды чтения, которая считывает байты из памяти, а затем делает недействительным кеш процессора, чтобы следующая команда чтения извлекала значения из оперативной памяти. Это называют *временным чтением* (volatile read) или *чтением с семантикой запроса* (read with acquire semantics). В архитектуре IA64 также есть видоизмененная версия команды записи, которая записывает байты из регистра в память и сбрасывает кеш процессора в основную память, так что остальные потоки, считывающие те же данные, получают самые последние значения. Это называют *временной записью* (volatile write) или *записью с семантикой освобождения* (write with release semantics). В дополнение, в архитектуре IA64 предусмотрена команда *защиты памяти* (memory fence), которая сбрасывает кеш процессора в основную память, а затем объявляет его недействительным.

Временная запись и чтение

Теперь мы знаем, как процессорный кеш влияет на поведение приложений и что некоторые процессоры предлагают особые команды, предоставляющие программисту определенный контроль над согласованностью кеша. Познакомимся, как CLR позволяет задействовать преимущества этих команд. Ясно, что одна из основных задач CLR — предоставление абстрактной виртуальной машины, чтобы программисту не приходилось писать код, рассчитанный на какую-либо конкретную архитектуру процессора. Наиболее распространенный способ абстрагироваться от реализации для конкретного процессора — предоставить соответствующий метод. Класс *System.Threading.Thread* предлагает несколько статических методов, которые выглядят примерно так:

```
static Object  VolatileRead(ref Object address);
static Byte    VolatileRead(ref Byte address);
static SByte   VolatileRead(ref SBbyte address);
static Int16   VolatileRead(ref Int16 address);
static UInt16  VolatileRead(ref UInt16 address);
static Int32   VolatileRead(ref Int32 address);
static UInt32  VolatileRead(ref UInt32 address);
```

```
static Int64   VolatileRead(ref Int64 address);
static UInt64  VolatileRead(ref UInt64 address);
static IntPtr  VolatileRead(ref IntPtr address);
static UIntPtr VolatileRead(ref UIntPtr address);
static Single  VolatileRead(ref Single address);
static Double  VolatileRead(ref Double address);

static void VolatileWrite(ref Object address, Object value);
static void VolatileWrite(ref Byte address, Byte value);
static void VolatileWrite(ref SByte address, SByte value);
static void VolatileWrite(ref Int16 address, Int16 value);
static void VolatileWrite(ref UInt16 address, UInt16 value);
static void VolatileWrite(ref Int32 address, Int32 value);
static void VolatileWrite(ref UInt32 address, UInt32 value);
static void VolatileWrite(ref Int64 address, Int64 value);
static void VolatileWrite(ref UInt64 address, UInt64 value);
static void VolatileWrite(ref IntPtr address, IntPtr value);
static void VolatileWrite(ref UIntPtr address, UIntPtr value);
static void VolatileWrite(ref Single address, float value);
static void VolatileWrite(ref Double address, Double value);

static void MemoryBarrier();
```

Все методы *VolatileRead* выполняют чтение с семантикой запроса; они считывают значение, на которое ссылается параметр *address*, а затем объявляют недействительным кеш процессора. Все методы *VolatileWrite* выполняют запись с семантикой освобождения; они сбрасывают содержимое кеша в основную память, а затем изменяют значение, на которое ссылается параметр *address*, на значение аргумента *value*. Метод *MemoryBarrier* обеспечивает защиту памяти — он сбрасывает данные процессорного кеша в основную память, после чего объявляет кеш недействительным.

Теперь мы можем на основе кода класса *CacheCoherencyProblem* создать новый класс *VolatileMethod*:

```
internal sealed class VolatileMethod {
   private Byte  m_initialized = 0;
   private Int32 m_value = 0;

   // Этот метод выполняется одним потоком.
   public void Thread1() {
      m_value = 5;
      Thread.VolatileWrite(ref m_initialized, 1);
   }

   // Этот метод выполняется другим потоком.
   public void Thread2() {
      if (Thread.VolatileRead(ref m_initialized) == 1) {
         // Если выполняется эта строка, отобразится значение 5.
         Console.WriteLine(m_value);
      }
   }
}
```

Внимание! Если для взаимодействия друг с другом потоки используют общую память, последний байт должен записываться в общую память с использованием временной записи. Считывание должно выполняться с использованием временного чтения. Приведенный выше код демонстрирует это.

Поддержка volatile-полей в C#

Сложно сделать так, чтобы все программисты вызывали методы *VolatileRead* и *VolatileWrite* корректно. Разработчикам сложно помнить обо всем этом и прогнозировать, что могут делать с общими данными другие потоки, работающие в фоновом режиме. Компилятор C# предлагает ключевое слово *volatile*, которое можно применять к статическим полям или экземплярам полей следующих типов: *Byte*, *SByte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Char*, *Single* или *Boolean*. Это ключевое слово можно также использовать для ссылки на типы и любые поля перечислений, если в основе перечислимого типа лежит тип *Byte*, *SByte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Single* или *Boolean*. Компилятор JIT гарантирует, что все обращения к volatile-полям выполняются по механизму временного считывания и записи, так что не обязательно явно вызывать какие-либо статические методы *VolatileXxx*. Кроме того, ключевое слово *volatile* сообщает компиляторам C# и JIT, что поле не нужно кешировать в регистре процессора, в этом случае во всех операциях с этим полем будет использоваться значение, считанное из основной памяти.[1]

Ключевое слово *volatile* позволяет на основе кода класса *VolatileMethod* создать новый класс *VolatileField*:

```
internal sealed class VolatileField {
   private volatile Byte m_initialized = 0;
   private Int32 m_value = 0;

   // Этот метод выполняется одним потоком.
   public void Thread1() {
      m_value = 5;
      m_initialized = 1;
   }

   // Этот метод выполняется другим потоком.
   public void Thread2() {
      if (m_initialized == 1) {
         // Если выполняется эта строка, отобразится значение 5.
         Console.WriteLine(m_value);
      }
   }
}
```

Некоторым разработчикам на C# (и мне в том числе) не нравится ключевое слово *volatile* — они считают, что в языке программирования этого слова быть не

[1] Исторически сложилось, что ключевое слово *volatile* обозначает в неуправляемом коде C/C++ именно это.

должно. Они полагают, что в большинстве алгоритмов требуется совсем немного операций временного чтения или записи в поле, а большинство обращений можно выполнить обычным путем, что, к тому же, положительно скажется на производительности. Редко требуется, чтобы все операции доступа к полю были временными. Например, сложно сказать, как выполнять операции временного считывания в алгоритмах, подобных этому:

```
m_amount = m_amount + m_amount; // m_amount - определенное в классе поле.
m_amount *= m_amount
```

Кроме того, C# не поддерживает передачу volatile-полей в метод по ссылке. Например, если поле *m_amount* определено как *volatile Int32*, при попытке вызвать метод *TryParse* типа *Int32* компилятор выдаст предупреждение:

```
Boolean success = Int32.TryParse("123", out m_amount);
// При обработке этой строки кода компилятор C# покажет следующее предупреждение:
// CS0420: a reference to a volatile field will not be treated as volatile
// (CS0420: ссылка на volatile-поле не будет считаться временной).
```

Создавая JIT-компилятор для архитектуры IA64, разработчики CLR поняли, что у многих программистов (включая их самих) есть код, который не будет корректно работать при вызове не из временных (неупорядоченных) операций записи и чтения. Поэтому они решили, что JIT-компилятор IA64 должен всегда создавать инструкции чтения, включающие семантику запроса, а операции записи должны всегда включать семантику освобождения. Это позволило существующим приложениям, работающим в архитектуре x86, работать без ошибок и в архитектуре IA64. К сожалению, это сильно ударило бы по производительности, поэтому пришлось пойти на компромисс. Во всех операциях записи в JIT-компиляторе IA64 используется семантика освобождения, а чтение выполняется в обычном порядке. Чтобы выполнить чтение с семантикой запроса, нужно вызвать метод *VolatileRead* или применить ключевое слово *volatile*. Это намного более понятная модель управления памятью, которую разработчики вполне в состоянии освоить.

Microsoft обещает, что все имеющиеся и любые будущие JIT-компиляторы будут четко придерживаться этой модели памяти, в которой используются обычные операции чтения, но запись всегда выполняется с семантикой освобождения. Если быть совсем точным, правила доступа к памяти немного сложнее. Впрочем, большинство программистов об этом догадывается.

Примечание Как говорилось в главе 1, ассоциация ECMA определила стандартную версию CLR, названную Common Language Infrastructure (CLI). В документации ECMA описывается модель памяти, которой должны соответствовать все CLI-совместимые среды времени выполнения. В этой версии модели памяти все операции чтения и записи являются неупорядоченными, если только программист явно не вызывает метод *VolatileRead* или *VolatileWrite* или не использует ключевое слово *volatile*. Это также объясняет, почему FCL все еще содержит методы *VolatileWrite*, хотя они бесполезны, если приложение работает в CLR, созданной компанией Microsoft.

В Microsoft посчитали модель ECMA слишком сложной и непонятной, поэтому реализовали собственную версию CLI (то есть CLR), использу-

ющую более сильную модель памяти, в которой все операции записи выполняются с семантикой освобождения. Такое изменение позволяет считать CLR компании Microsoft совместимой со стандартом, так как любые приложения, написанные для модели памяти ECMA, могут работать в CLR от Microsoft.

Однако существует возможная проблема, заключающаяся в том, что кто-то может написать приложение и протестировать его при помощи CLR компании Microsoft. Это приложение может отлично работать, но в другой реализации CLI оно может не работать. Было бы лучше для всех, если бы стандарт ECMA и все реализации этого стандарта придерживались единой модели памяти. Microsoft надеется, что в следующих версиях стандарта ECMA будет принята модель памяти, которая сейчас применяется в CLR.

Наконец, нужно сказать, что всегда, когда поток вызывает *Interlocked*-метод (см. следующий раздел), процессор принудительно обеспечивает согласованность кеша. Так что при работе с переменными при помощи *Interlocked*-методов не нужно беспокоиться о соответствии этой модели памяти. Кроме того, все блокировки синхронизации потоков (*Monitor*, *ReaderWriterLock*, *Mutex*, *Semaphore*, *AutoResetEvent*, *ManualResetEvent* и другие) вызывают *Interlocked*-методы, так что программисту также не приходится заботиться о моделях памяти.

Внимание! В общем случае, я настоятельно не рекомендую вызывать методы *VolatileRead* и *VolatileWrite*, а также ключевое слово *volatile*. Вместо этого лучше использовать *Interlocked*-методы или конструкции синхронизации потоков более высокого уровня. Эти методы работают всегда, независимо от модели памяти и платформы процессора.

Семейство *Interlocked*-методов

Если несколько потоков обращается к общим данным, нужно обеспечить безопасность потоков. Самый быстрый способ — задействовать семейство *Interlocked*-методов. Эти методы работают очень быстро (по сравнению с другими конструкциями синхронизации потоков) и просты в применении. Недостаток один: очень ограниченные возможности. В классе *System.ThreadingInterlocked* определено несколько статических методов, которые могут автоматически изменять переменную безопасно для потоков. Методы, оперирующие переменными типа *Int32*, используются наиболее часто:

```
public static class Interlocked {
   // Автоматически выполняет (location++).
   public static Int32 Increment(ref Int32 location);

   // Автоматически выполняет (location–).
   public static Int32 Decrement(ref Int32 location);

   // Автоматически выполняет (location1 += value).
   // ПРИМЕЧАНИЕ: значение может быть отрицательным числом,
```

```
   // что позволяет выполнять вычитание.
   public static Int32 Add(ref Int32 location1, Int32 value);

   // Автоматически выполняет (location1 = value).
   public static Int32 Exchange(ref Int32 location1, Int32 value);

   // Автоматически выполняет следующее:
   // if (location1 == comparand) location1 = value
   public static Int32 CompareExchange(ref Int32 location1,
      Int32 value, Int32 comparand);
   ...
}
```

Помимо этого, в классе *Interlocked* есть методы *Exchange* и *CompareExchange*, принимающие параметры *Object*, *IntPtr*, *Single* и *Double*. Также есть обобщенная версия, в которой тип ограничен типом *class* (любой ссылочный тип). Все *Interlocked*-методы требуют, чтобы передаваемый адрес переменной был корректно выровнен, иначе генерируется исключение *DataMisalignedException*. К счастью, CLR автоматически выравнивает поля типов, если атрибут *[StructLayout(LayoutKind.Explicit)]* не применен к типу, а атрибут *[FieldOffcet(…)]* не применен к отдельным полям и принудительно не нарушает их выравнивание.

Внимание! В классе *Interlocked* также есть методы, оперирующие переменными *Int64*. Однако никогда не следует вызывать эти методы, поскольку CLR не обеспечивает корректное выравнивание таких переменных.

В своей колонке Concurrent Affairs в библиотеке MSDN я привожу несколько примеров того, как правильно использовать различные *Interlocked*-методы. Одну из этих статей можно найти по адресу *http://MSDN.Microsoft.com/MSDNMag/Issues/05/10/ConcurrentAffairs/Default.aspx*.

Класс *Monitor* и блоки синхронизации

В Win32 API структура *CRITICAL_SECTION* и связанные с ней функции предлагают самый быстрый и эффективный способ синхронизации потоков для взаимоисключающего доступа к общему ресурсу, когда все потоки работают в одном процессе. Взаимоисключающий доступ к общему ресурсу нескольких потоков — это, наверное, самая распространенная форма синхронизации потоков. CLR не предоставляет структуру *CRITICAL_SECTION*, но предлагает похожий механизм, позволяющий организовать взаимоисключающий доступ к ресурсу в наборе потоков одного процесса. Используя этот механизм, задействуют класс *System.Threading.Monitor* и блоки синхронизации.

Сейчас я объясню, как эта популярная форма синхронизации потоков реализована в CLR. В частности, я расскажу, почему класс *Monitor* и блоки синхронизации созданы именно такими и как они работают. В конце раздела я укажу на недостатки используемой архитектуры, и продемонстрирую, как их обойти.

«Отличная» идея

CLR — объектно-ориентированная платформа. Это значит, что разработчики создают объекты, а затем управляют ими, вызывая члены типа. Иногда к этим объектам обращается несколько потоков, и, чтобы состояние объектов не повреждалось, нужно обеспечить синхронизацию потоков. При проектировании CLR специалисты Microsoft решили создать механизм, который позволил бы разработчикам просто и без усилий синхронизировать доступ к состоянию объекта.

Основная идея такова: с каждым объектом в куче связана структура данных (аналогично структуре *CRITICAL_SECTION* в Win32), которую можно использовать как блокировку синхронизации. В библиотеке FCL (Framework Class Library) есть методы, позволяющие получить ссылку на объект. Эти методы должны использовать структуру данных объекта для установки и освобождения блокировки синхронизации потока.

В Win32 реализующий эту идею неуправляемый класс C++ будет выглядеть примерно так:

```
class SomeType {
private:
   // Закрытое поле CRITICAL_SECTION, связанное с объектом.
   CRITICAL_SECTION m_csObject;

public:
   SomeType() {
      // Конструктор инициализирует поле CRITICAL_SECTION объекта.
      InitializeCriticalSection(&m_csObject);
   }

   ~SomeType() {
      // Деструктор удаляет поле CRITICAL_SECTION объекта.
      DeleteCriticalSection(&m_csObject);
   }

   void SomeMethod() {
      // Мы используем поле CRITICAL_SECTION объекта для синхронизации
      // доступа к объекту нескольких потоков.
      EnterCriticalSection(&m_csObject);
      // Здесь располагается код, исполняемый безопасно в отношении потоков.
      LeaveCriticalSection(&m_csObject);
   }

   void AnotherMethod() {
      // Мы используем поле CRITICAL_SECTION объекта для синхронизации
      // доступа к объекту нескольких потоков.
      EnterCriticalSection(&m_csObject);
      // Здесь располагается исполняемый безопасный код...
      LeaveCriticalSection(&m_csObject);
   }
};
```

В сущности, CLR предоставляет для каждого объекта отдельное поле по типу поля *CRITICAL_SECTION* и берет на себя задачи инициализации и удаления этого поля. Все, что остается сделать разработчику, — написать код для ввода поля в каждом методе, которому требуется синхронизация потоков.

Реализация «отличной» идеи

Очевидно, что сопоставление поля *CRITICAL_SECTION* (которое занимает примерно 24 байта в 32-разрядных системах и около 40 байт в 64-разрядных) с каждым объектом в куче довольно расточительно, особенно если для большинства объектов никогда не требуется безопасный доступ. Чтобы снизить расходование памяти, команда разработчиков CLR использует более эффективный способ предоставления только что описанной функциональности. Вот как это работает: при инициализации CLR выделяется память под массив блоков синхронизации. Блок синхронизации — это участок памяти, который можно связать с объектом. Каждый блок синхронизации содержит те же поля, что и Win32-структура *CRITICAL_SECTION*.

Как уже говорилось, при создании объекта в куче, с ним связываются два дополнительных служебных поля. Первое служебное поле, указатель на объект-тип, содержит адрес памяти, по которому находится объект-тип типа. Второе служебное поле, индекс блока синхронизации, содержит целочисленный индекс блока в массиве блоков синхронизации.

При конструировании объекта индекс блока синхронизации получает отрицательное значение, указывающее, что нет блока синхронизации, поставленного в соответствие объекту. Затем, когда вызывается метод для входа в блок синхронизации объекта, CLR находит в массиве свободный блок синхронизации и задает индексу блока синхронизации объекта значение, соответствующее найденному блоку. Иначе говоря, блоки синхронизации связываются с объектами прямо во время выполнения. После того как все потоки освобождают блок синхронизации объекта, индекс блока синхронизации опять получает отрицательное значение, чтобы считаться свободным, и может связываться с другим объектом. На рис. 24-1 этот процесс показан наглядно.

Логически у каждого объекта в куче есть связанный с ним блок синхронизации, который можно использовать для быстрой синхронизации потоков. Однако физически структуры блоков синхронизации связываются с объектом только по необходимости, а когда эта необходимость отпадает, привязка снимается. Это обеспечивает эффективность использования памяти. Кстати, при необходимости в массиве блоков синхронизации могут создаваться дополнительные блоки, поэтому не стоит беспокоиться, что системе может не хватить блоков, если потребуется синхронизировать много объектов.

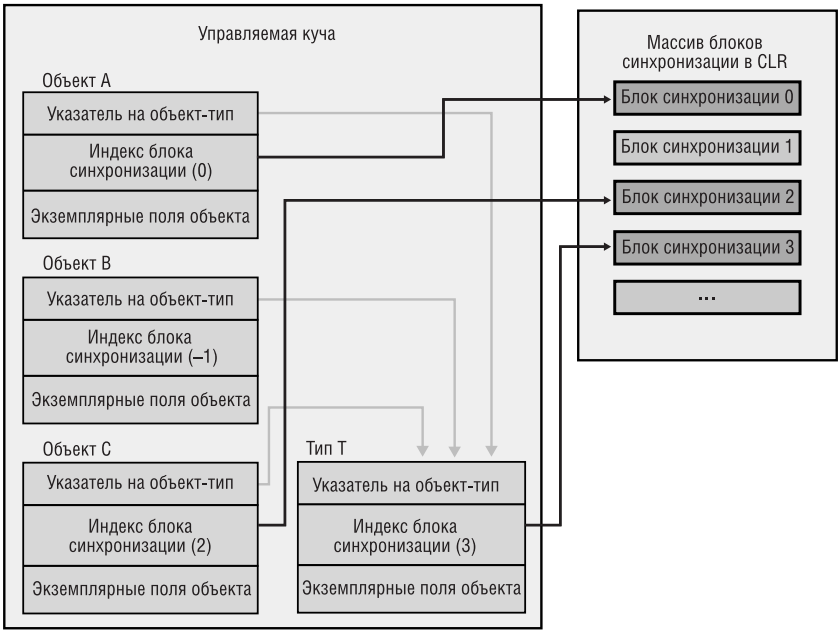


Рис. 24-1. *Индекс блока синхронизации объектов в куче (включая объекты типов) может ссылаться на запись в массиве блоков синхронизации CLR*

Использование класса *Monitor* для управления блоком синхронизации

Познакомившись с инфраструктурой блоков синхронизации, перейдем к рассмотрению установки и отмены блокировки объекта. Чтобы заблокировать/разблокировать блок синхронизации, вызывают статические методы, определенные в классе *System.Threading.Monitor*. Вот метод, который можно вызвать, чтобы заблокировать блок синхронизации объекта:

```
static void Enter(Object obj);
```

При вызове этот метод сначала проверяет, отрицателен ли указанный индекс блока синхронизации; если да, метод находит свободный блок синхронизации и записывает его индекс в индекс блока синхронизации объекта. Кстати, в CLR механизм поиска свободного блока синхронизации и связывания его с объектом обеспечивает безопасность потоков. После связывания блока с объектом метод *Monitor.Enter* проверяет указанный блок синхронизации — не занят ли он другим потоком. Если блок свободен, вызывающий поток становится владельцем блока. Если же этим блоком владеет другой поток, вызывающий поток приостанавливается, пока владеющий этим блоком поток не освободит его.

Если нужно создать более надежный код, вместо *Monitor.Enter* можно вызвать метод *Monitor.TryEnter* (у него три перегруженных версии):

```
static Boolean TryEnter(Object obj);
static Boolean TryEnter(object obj, Int32 millisecondsTimeout);
static Boolean TryEnter(Object obj, TimeSpan timeout);
```

Первая версия просто проверяет, может ли вызывающий поток занять блок синхронизации и при положительном результате возвращает значение *true*. Две другие версии позволяют указать время тайм-аута — сколько вызывающий поток будет ожидать освобождения блока. Все методы возвращают *false*, если занять блок синхронизации не удалось.

После того как поток стал владельцем блока синхронизации, код получает доступ к любым данным блока синхронизации, для защиты которых он используется. По завершении работы с блоком поток должен освободить его вызовом метода *Monitor.Exit*:

```
static void Exit(Object obj);
```

Если поток, вызывающий метод *Monitor.Exit*, не владеет указанным блоком синхронизации, генерируется исключение *SynchronizationLockException*. Заметьте также, что поток может владеть блоком синхронизации рекурсивно, но при этом каждому успешному вызову метода *Monitor.Enter* или *Monitor.TryEnter* должен соответствовать вызов метода *Monitor.Exit* — только после этого блок может считаться свободным.

Способ синхронизации, предлагаемый Microsoft

Познакомимся с примерами кода, демонстрирующими, как Microsoft предлагает разработчикам использовать класс *Monitor* и блоки синхронизации:

```
internal sealed class Transaction {

   // Поле, показывающее время последней выполненной транзакции.
   private DateTime timeOfLastTransaction;

   public void PerformTransaction() {
      Monitor.Enter(this); // Enter this object's lock
      // Выполнение транзакции...

      // Запись времени последней транзакции.
      timeOfLastTransaction = DateTime.Now;

      Monitor.Exit(this);  // Отмена блокировки объекта.
   }

   // Неизменяемое свойство, возвращающее время последней транзакции.
   public DateTime LastTransaction {
      get {
         Monitor.Enter(this); // Блокировка объекта.

         // Сохранение времени выполнения последней транзакции
         // во временной переменной.
         DateTime dt = timeOfLastTransaction;

         Monitor.Exit(this);  // Отмена блокировки объекта.
         return dt; // Возвращение сохраненных даты и времени.
      }
   }
}
```

В этом примере показано, как использовать методы *Enter* и *Exit* метода *Monitor* для блокировки и разблокировки блока синхронизации объекта. Заметьте: реализация свойства требует вызовов методов *Enter* и *Exit* и наличия временной переменной *dt*. Это нужно для исключения возможности повреждения возвращаемого значения — это возможно, если в момент вызова потоком метода *PerformTransaction* к свойству обращается другой поток.

Внимание! Приведенный выше код демонстрирует, как специалисты Microsoft первоначально предлагали использовать блоки синхронизации объекта. Однако в процессе работы этого кода возможно возникновение серьезных проблем, поэтому его не следует копировать в собственные проекты. Чуть ниже я объясню, в чем дело и как писать код, в котором этих проблем не возникает.

Упрощение кода C# при помощи оператора *lock*

Поскольку последовательность «вызов метода *Monitor.Enter* — обращение к защищенному ресурсу — вызов метода *Monitor.Exit*» используется очень часто, в C# предусмотрен особый синтаксис для упрощения кода. Два следующие фрагмента кода идентичны:

```
private void SomeMethod() {
   lock (this) {

      // Доступ к объекту...

   }
}
```

```
private void SomeMethod() {
Object temp = this;
Monitor.Enter(temp);
try {
   // Доступ к объекту...

}
finally {
Monitor.Exit(temp);
}
   }
```

Оператор *lock* упрощает класс *Transaction*. Взгляните на новое, усовершенствованное свойство *LastTransaction*; временная переменная больше не нужна.

```
internal sealed class Transaction {

   // Поле, указывающее время выполнения последней транзакции.
   private DateTime timeOfLastTransaction;

   public void PerformTransaction() {
      lock (this) {  // Enter this object's lock
         // Выполнение транзакции...

         // Запись времени выполнения последней транзакции.
         timeOfLastTransaction = DateTime.Now;
      }  // Отмена блокировки объекта.
   }
```

```
   // Неизменяемое свойство, возвращающее время последней транзакции.
   public DateTime LastTransaction {
      get {
         lock (this) {  // Блокировка объекта.
            return timeOfLastTransaction; // Возвращение даты и времени.
         }  // Отмена блокировки объекта.
      }
   }
}
```

Кроме того, что код стал короче и проще, оператор *lock* гарантирует вызов метода *Monitor.Exit*, а значит, блок синхронизации освобождается, даже если в блоке *lock* возникает исключение. Использовать обработку исключений в механизмах синхронизации потоков следует всегда — это позволит корректно отменять блокировки. Если используется оператор *lock* языка C# , компилятор автоматически создает нужный код.

Внимание! Напомню еще раз, что приведенный выше код демонстрирует, как в Microsoft первоначально предполагали использовать блоки синхронизации объекта. Однако у этого кода есть серьезный недостаток, поэтому не следует копировать этот код в собственные проекты.

Способ синхронизации статических членов, предлагаемый Microsoft

Класс *Transaction* показывает, как синхронизировать доступ к полям экземпляров объектов. Но что, если в типе определено несколько статических полей и статических методов, которые обращаются к этим полям? В этом случае у вас нет экземпляра типа в куче, а значит, нет блока синхронизации, который можно использовать, и нет ссылки на объект, которую можно передать методам *Enter* и *Exit* объекта *Monitor*.

На рис. 24-1 видно, что у объекта A, объекта B и объекта C указатель на объект-тип ссылается на *Type-T* (объект-тип). Иначе говоря, у всех трех объектов один тип. Как говорилось в главе 4, объект-тип также является объектом в куче и, как и у остальных объектов, у него два служебных члена: индекс блока синхронизации и указатель на объект-тип. Это значит, что блок синхронизации можно связывать с объектом-типом, а ссылку на объект-тип — передавать методам *Monitor*, *Enter*, *TryEnter* и *Exit*. В приведенной ниже версии класса *Transaction* все члены сделаны статическими, а метод *PerformTransaction* и свойство *LastTransaction* были изменены — они демонстрируют, как в Microsoft первоначально предлагали разработчикам синхронизировать доступ к статическим членам.

```
internal static class Transaction {

   // Поле, указывающее время выполнения последней транзакции.
   private static DateTime timeOfLastTransaction;

   public static void PerformTransaction() {
      lock (typeof(Transaction)) {  // Enter the type object's lock
         // Выполнение транзакции...
```

```
            // Запись времени выполнения последней транзакции.
            timeOfLastTransaction = DateTime.Now;
        }  // Отмена блокировки объекта.
    }

    // Неизменяемое свойство, возвращающее время последней транзакции.
    public static DateTime LastTransaction {
        get {
            lock (typeof(Transaction)) {  // Блокировка объекта-типа.
                return timeOfLastTransaction; // Возвращение даты и времени.
            }  // Отмена блокировки объекта.
        }
    }
}
```

В этом коде статический метод и аксессор *get* свойства не могут ссылаться на объект *this*, так как он не доступен для статических членов. Вместо этого оператору *lock* передается ссылка на тип объекта-типа (полученный при помощи оператора *typeof* языка C#).

Внимание! Напомню еще раз, что приведенный выше код демонстрирует, как в Microsoft первоначально предполагали использовать блоки синхронизации объекта. Однако у этого кода есть серьезный недостаток, поэтому не следует копировать этот код в собственные проекты. В следующем разделе я наконец объясню, что в предыдущих трех примерах неверно и как устранить проблему.

Почему же «отличная» идея оказалась такой неудачной

Идея логически связать структуру данных синхронизации с каждым объектом кучи выглядит очень заманчивой, и это действительно так. Но разработчики Microsoft совершили большую ошибку при реализации этой идеи в CLR. Сейчас объясню почему. Помните, неуправляемый код C++, приведенный в разделе «Отличная идея» в этой главе? Если бы вы писали этот код самостоятельно, сделали ли бы вы поле *CRTICAL_SECTION* открытым? Конечно, нет, это же глупо! Открытое поле позволяет любому коду приложения изменять структуру *CRTICAL_SECTION*. Злоумышленнику ничего бы не стоило вызвать взаимную блокировку потоков, в которых работают экземпляры этого типа.

Ну и что же — блок синхронизации как раз имеет открытую структуру синхронизации данных, связанную с каждым объектом кучи! Любой код, имеющий ссылку на объект, в любой момент может передать эту ссылку в методы *Enter* и *Exit* и перехватить блокировку. Еще хуже то, что любой код может также передать в эти методы ссылку на любой объект-тип и перехватить блокировку этого типа. Проблема также возникает при передаче интернированного объекта *String*, поскольку теперь несколько строк использует одну блокировку. И, если передавать ссылку на объект производного от *MarshalByRefObject* типа, можно заблокировать либо сам объект, либо прокси-объект (сам объект останется не заблокированным). Как видите, специалисты Microsoft не очень хорошо над этим подумали и допустили много ошибок.

Ниже приведен код, показывающий, к каким катастрофическим последствиям это может привести. В этом коде поток *Main* создает объект *SomeType* и налагает на этот объект блокировку. В какой-то момент времени происходит сборка мусора (в этом коде она принудительно инициируется только для демонстрации), и когда вызывается метод *Finalize* объекта *SomeType*, он пытается перехватить блокировку объекта. К сожалению завершающий поток CLR не может получить блокировку объекта, так как ею владеет основной поток приложения. Это приводит к остановке CLR-потока деструктора — теперь ни один процесс (включая все AppDomain процесса) не сможет завершиться и никакая память, занятая объектами в куче, не может освободиться!

```
using System;
using System.Threading;

public sealed class SomeType {
   // Это метод Finalize объекта SomeType.
   ~SomeType() {
      // С целью демонстрации CLR-поток деструктора
      // пытается перехватить блокировку объекта.
      // ПРИМЕЧАНИЕ: поскольку блокировкой объекта владеет поток Main,
      // завершающий поток блокируется!
      Monitor.Enter(this);
   }
}

public static class Program {
   public static void Main() {
      // Создание объекта SomeType.
      SomeType st = new SomeType();

      // Этот некорректный код перехватывает блокировку объекта
      // и не освобождает ее.
      Monitor.Enter(st);

      // Для демонстрации запускаем сборку мусора
      // и ожидаем завершения выполнения методов-деструкторов.
      st = null;
      GC.Collect();
      GC.WaitForPendingFinalizers();

      Console.WriteLine("We never get here, both threads are deadlocked!");
   }
}
```

К сожалению, CLR, FCL и компиляторы предлагают много функций, в которых используется блокировка объектов. В главе 10 я уже упоминал о возможных проблемах с событиями. CLR также использует открытую блокировку объекта типа при вызове конструктора класса для типа.

В пространстве имен *System.Runtime.CompilerServices* есть особый класс атрибутов по имени *MethodImplAttribute*. Этот атрибут можно применять к методу, ус-

танавливая флаг *MethodImplOptions.Synchronized*. Если это экземплярный метод, то при этом JIT-компилятор заключает весь код метода в функцию *lock(this)*. Если это статический метод, код помещается в функцию *lock(typeof(<имя_типа>))*, где *<имя_типа>* — имя самого типа. Это плохо — о причинах уже говорилось ранее, поэтому никогда не следует использовать *MethodImplAttribute* с флагом *MethodImplOptions.Synchronized*.

Нам не удастся исправить CLR, FCL или компилятор C#, но при написании собственного кода, можно соблюдать максимальную осторожность и обходить проблему открытой блокировки. Для этого лишь нужно определить закрытое поле *System.Object* в качестве члена своего типа, создать объект, а затем передать ссылку на закрытый объект в оператор *lock*. Вот исправленная версия класса *Transaction*, которая более защищена за счет использования закрытой блокировки объекта:

```
internal sealed class TransactionWithLockObject {
   // Выделяем память для закрытого объекта, используемого для блокирования.
   private Object m_lock = new Object();

   // Поле, указывающее время выполнения последней транзакции.
   private DateTime timeOfLastTransaction;

   public void PerformTransaction() {
      lock (m_lock)   { // Блокируем объект закрытого поля.
         // Выполнение транзакции...

         // Записываем время последней транзакции.
         timeOfLastTransaction = DateTime.Now;
      }  // Отмена блокировки объекта закрытого поля.
   }

   // Неизменяемое свойство, возвращающее время последней транзакции.
   public DateTime LastTransaction {
      get {
      lock (m_lock){ // Блокировка объекта закрытого поля.
            return timeOfLastTransaction; // Возвращение даты и времени.
         }  // Отмена блокировки объекта закрытого поля.
      }
   }
}
```

Внимание! В этом коде показано, как избавляться от проблем, о которых говорилось ранее. Самая важная его особенность — блокировка синхронизации потока с использованием закрытого поля. Это не позволяет постороннему коду получить доступ к блокировке и, следовательно, вмешаться в код вашего типа и вызвать взаимную блокировку потоков.

Кажется странным создавать в классе *Monitor* объект *System.Object* только для синхронизации. Честно говоря, в Microsoft спроектировали класс *Monitor* некорректно. Его нужно было спроектировать так, чтобы для создания блокировки синхронизации потока нужно было создать экземпляр типа *Monitor*. Затем статические

методы должны быть экземплярными методами, самостоятельно работающими с объектом блокировки — тогда не пришлось бы передавать аргумент *System.Object* в методы класса *Monitor*. Это решило бы все проблемы и упростило бы модель программирования. Кстати, приведенный выше код легко модифицировать для синхронизации статических методов — достаточно просто изменить все члены на *static*.

Если в вашем типе уже определены какие-либо закрытые поля данных, можете задействовать одно из них как объект блокировки, передаваемый в методы класса *Monitor*. Это поможет сэкономить немного памяти, поскольку не нужно будет размещать объект *System.Object*. Однако я бы не стал делать этого просто ради незначительной экономии памяти; код типов закрытых полей может вызывать метод *lock(this)*. Если это случится, код будет заблокирован, и может возникнуть взаимная блокировка.

Внимание! Никогда не передавайте переменную значимого типа в метод *Monitor.Enter* или оператор *lock* языка C#. У распакованных экземпляров значимого типа нет члена «индекс блокировки синхронизации», а поэтому их нельзя использовать для синхронизации. Если передать распакованный экземпляр значимого типа в метод *Monitor.Enter*, компилятор C# автоматически создаст код упаковки экземпляра. Если тот же экземпляр передать в *Monitor.Exit*, снова произойдет упаковка. В результате код заблокирует один, но разблокирует совершенно другой объект, и никакой безопасности потоков обеспечено не будет. Если передать распакованный экземпляр значимого типа оператору *lock*, компилятор обнаружит это и вернет ошибку: error CS0185: 'valuetype' is not a reference type as required by the lock statement (ошибка CS0185: значимый тип не является ссылочным типом, который требуется оператору *lock*). Однако компилятор не вернет ошибку или предупреждение, если в метод *Monitor.Enter* или *Monitor.Exit* передать экземпляр значимого типа.

Знаменитый способ блокировки с двойной проверкой

Существует способ, известный как *блокировка с двойной проверкой* (double-check locking). Этот способ используется, если нужно отложить создание singleton-объекта до тех пор, пока приложение не запросит его [иногда это называют *отложенной инициализацией* (lazy initialization)]. Если приложение не запросит объект, он никогда не будет создан, что позволяет экономить время и память. Возможность возникновения проблем появляется, когда несколько потоков одновременно запрашивают singleton-объект. В этом случае нужно использовать какую-либо синхронизацию потоков, чтобы singleton-объект создавался только раз.

Этот способ знаменит не тем, что он особо интересен или полезен — просто о нем очень много писали. Он часто применялся в языке Java, но позже обнаружилось, что Java не обеспечивает его работу во всех возможных ситуациях. Известный документ, описывающий эту проблему, есть в Интернете по адресу *http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html*.

В CLR возникла такая же проблема, и в Microsoft приложили много усилий, чтобы решить ее. В результате в Microsoft пришли к модели памяти, которая сейчас и используется в CLR и которую я уже описал в разделе «Целостность памяти, вре-

менный доступ к памяти и volatile-поля». В этой модели памяти проблема блокировки с двойной проверкой не стоит. Вот пример кода, показывающий, как применять этот способ блокировки в C#:

```
public sealed class Singleton {
   // Объект s_lock требуется для обеспечения безопасности в многопоточной среде.
   // Наличие этого объекта предполагает, что для создания singleton-объекта требуется
   // больше ресурсов, чем для создания объекта System.Object, и что его создание
   // может вовсе не понадобиться. В противном случае проще и эффективнее просто создать
   // singleton-объект в конструкторе класса.
   private static Object s_lock = new Object();

   // Временные операции в модели памяти CLR не требуются, но они обязательны в
   // модели памяти ECMA. Чтобы соответствовать стандарту, я включил ее сюда.
   private static volatile Singleton s_value;

   // Закрытый конструктор не позволяет создавать экземпляры любому,
   // не относящемуся к классу коду.
   private Singleton() { }

   // Открытое статическое свойство,
   // возвращающее singleton-объект (который создается при необходимости).
   public static Singleton Value {
      get {
         // Был ли уже создан singleton-объект?
         if (s_value == null) {
            // Нет, только один поток должен создавать его.
            lock (s_lock) {
               // Создал ли его другой поток?
               if (s_value == null) {
                  // Нет, все в порядке, этот поток создаст его.

                  // Временность операций гарантирует, что все поля singleton-объекта
                  // (инициализированные конструктором) будут очищены, прежде чем
                  // другие потоки увидят ссылку на этот объект.
                  s_value = new Singleton();
               }
            }
         }

         // Возвращение ссылки на singleton-объект.
         return s_value;
      }
   }
}
```

Принцип работы блокировки с двойной проверкой заключается в том, что вызов метода-аксессора *get* быстро проверяет поле *s_value*, выясняя, был ли объект уже создан, и если да, метод возвращает ссылку на него. Изюминка здесь в том, что после создания объекта синхронизация потоков не требуется, и приложение будет работать очень быстро. С другой стороны, если первый поток, вызвавший ме-

тод получения доступа к свойству *value*, увидит, что объект не был создан, он устанавливает блокировку синхронизации потоков, чтобы только один поток мог создать singleton-объект. Это значит, что производительность страдает только однажды, при первом запросе singleton-объекта.

Теоретически возможно, что JIT-компилятор считает значение поля *s_value* из регистра процессора в начале метода-аксессора *get* и просто запросит этот регистр при проверке условия второго оператора *if*. Если JIT-компилятор создаст именно такой код, значение во втором операторе *if* всегда будет равно *true* и singleton-объект может быть создан несколькими потоками. Это совсем не то, что нужно. Однако JIT-компилятор знает, что поле, помеченное ключевым словом *volatile*, кешировать не надо. Кроме того, JIT-компилятор всегда заново считывает поле, к которому обращается после вызова метода, если этот метод включает операции временного чтения или записи (например, *Monitor.Enter* или *Monitor.Exit*).[2]

В начале этого раздела я сказал, что блокировка с двойной проверкой не особо интересна. По-моему, разработчики считают ее интересной и используют гораздо чаще, чем следовало бы. В большинстве случаев этот способ только снижает производительность. Вот гораздо более простая версия класса *Singleton*, которая ведет себя так же, как и предыдущая версия, но в ней не используется блокировка с двойной проверкой.

```
public sealed class Singleton {
   private static Singleton s_value = new Singleton();

   // Закрытый конструктор не позволяет какому-либо внешнему коду
   // создавать экземпляры класса.
   private Singleton() { }

   // Открытое статическое свойство, возвращающее singleton-объект.
   public static Singleton Value {
      get {
         return s_value;  // Возвращение ссылки на singleton-объект.
      }
   }
}
```

CLR автоматически вызывает конструктор класса типа при первой попытке получить доступ к члену этого класса, поэтому, когда поток в первый раз запросит метод-аксессор *get* свойства *Value* объекта *Singleton*, CLR автоматически вызо-

---

[2] В действительности JIT-компилятор не знает, что метод *Monitor.Enter* (или любой другой) выполняет временное чтение или запись. Чтобы узнать это, JIT-компилятору пришлось бы проанализировать код этого метода и других вызываемых им методов. Во-первых, на это требуется время, а во-вторых, некоторые методы написаны как неуправляемый код, который JIT-компилятор анализировать не умеет. Поэтому компилятор просто считает, что все методы выполняют операции временного чтения или записи, и поэтому всегда повторно считывает значение поля после вызова метода. В будущих версиях JIT-компиляторов могут появиться средства выяснения, выполняет ли метод временные операции, и тогда не придется писать подобный код — основанный на поведении имеющейся сейчас версии. Единственная гарантия того, что JIT-компилятор не будет кешировать значение после вызова метода состоит в том, что этот метод выполняет операцию временного чтения или записи.

вет конструктор класса, который создаст экземпляр объекта. Кроме того, CLR уже обеспечил безопасность конструктора класса в отношении синхронизации потоков — я объяснял все это в главе 8.

Блокировка с двойной проверкой менее эффективна, чем использование конструктора класса, поскольку приходится создавать собственный объект блокировки (в конструкторе класса) и самостоятельно писать весь дополнительный код для блокировки. Блокировка с двойной проверкой интересна только в том случае, если класс имеет много членов и нужно создавать singleton-объект, только когда вызывается один из членов. Кроме этого, чтобы получить выигрыш, создание singleton-объекта должно быть намного более затратным, чем создание объекта, используемого для блокировки.

Класс *ReaderWriterLock*

Очень часто встречается проблема синхронизации потоков, известная как проблема «множественного чтения и одиночной записи» (multiple-reader/single-writer). Эта проблема возникает, когда произвольное число потоков пытается получить доступ к общему ресурсу. Одни потоки (записывающие) хотят изменить содержимое данных, другие (считывающие) — только считать данные. Синхронизация необходима по четырем причинам:

- когда один поток записывает данные, остальным потокам нельзя их изменять;
- когда один поток записывает данные, остальным потокам нельзя их считывать;
- когда один поток считывает данные, остальные потоки не должны изменять их;
- когда один поток считывает данные, остальные потоки также могут считывать их.

FCL предоставляет класс *ReaderWriterLock*, инкапсулирующий эти четыре правила. Можно создать экземпляр этого класса и вызывать методы для установки и освобождения блокировки. У *ReaderWriterLock* нет статических членов, но есть несколько экземплярных методов и свойств. Обычно, подходя к этой теме, я описываю, как использовать такую блокировку в приложениях, но сейчас я этого сделать не могу, порекомендовать ее использовать могу только врагу. Как вы увидите далее, этот класс создает массу проблем.

Во-первых, производительность только входа и выхода из блокировки ужасно низкая. Она неудовлетворительна, даже если нет конкурирующих за блокировку потоков. Мои измерения показали, что вход и выход в блокировку с использованием класса *ReaderWriterLock* выполняется примерно в пять раз медленнее, чем при помощи класса *Monitor*.

Во-вторых, когда поток завершает запись, из всех ожидающих доступа потоков эта блокировка отдает приоритет считывающим, что приводит к исключительно медленной работе записывающих потоков. Обычно класс *ReaderWriterLock* используют, когда мало записывающих, но много считывающих потоков. В конце концов, если бы к ресурсу обращались только считывающие потоки, его не нужно было бы блокировать, а если обращаются только записывающие, то нужно использовать взаимоисключающую блокировку, например блокировку синхронизации (которой можно управлять при помощи класса *Monitor*). Так что если используется класс *ReaderWriterLock* и есть ожидающие своей очереди записывающие потоки, нужно отдавать приоритет им. Я знаю несколько человек, которые пытались ис-

пользовать класс *ReaderWriterLock*, но из-за того, что он отдает приоритет считывающим потокам, записывающие потоки надолго блокировались и очень долго не могли выполнить свою задачу.

В-третьих, класс *ReaderWriterLock* поддерживает рекурсию. То есть вошедший в блокировку и потом выходящий из нее поток должен быть одним и тем же потоком. Многие разработчики считают это преимуществом — я считаю это ошибкой. Все чаще возникают ситуации, когда один поток входит в блокировку и запускает операцию, обращающуюся к каким-то данным, а затем другой поток завершает эту операцию с данными и выходит из блокировки. Особенно часто это встречается в асинхронной модели программирования, описанной в главе 23. Блокировки, поддерживающие рекурсию, такие как *ReaderWriterLock* и *Monitor*, не поддерживают асинхронное программирование.

Если вы хотите применять подобную блокировку чтения/записи в своих приложениях, я бы порекомендовал реализовать собственную или использовать мою библиотеку Power Threading (доступна по адресу *http://Wintellect.com*). Мой класс блокировки чтения/записи называется *OneManyResourceLock* и работает почти так же быстро, как и блокировки класса *Monitor*. При выходе из блокировки класс отдает приоритет записывающим, а не считывающим потокам, и не поддерживает рекурсию.

Использование объектов ядра Windows в управляемом коде

Windows предлагает несколько объектов ядра для выполнения синхронизации потоков: мьютексы, семафоры и события. В этом разделе мы поговорим об использовании этих объектов в CLR. Я не буду детально описывать, как работают эти объекты и когда их нужно использовать, поскольку эта информация доступна уже много лет и детально описана во многих книгах, в том числе в моей — Programming Applications for Microsoft Windows, 4 Edition, Microsoft Press, 1999 (Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows, 4-е изд., СПб.: Питер, М.: Издательско-торговый дом «Русская редакция», 2001).

Методы классов *Monitor* и *ReaderWriterLock* позволяют выполнять синхронизацию потоков одного домена AppDomain, а объекты ядра можно использовать для синхронизации потоков различных AppDomain или процессов. Поэтому при создании объекта ядра можно задать права доступа, указав, кто и что может делать с этим объектом. Ожидая освобождения объекта ядра, поток должен переходить из пользовательского режима в режим ядра, что сильно снижает производительность. Поэтому синхронизация с использованием объектов ядра — самый медленный из механизмов синхронизации. Он намного медленнее, чем использование класса *ReaderWriterLock*. По моим измерениям, использование объектов ядра в 33 раза медленнее применения методов класса *Monitor*.

В пространстве имен *System.Threading* есть абстрактный класс *WaitHandle*. Это довольно простой класс, единственная задача которого — служить оболочкой описателя объекта ядра. В FCL есть несколько классов, производных от *WaitHandle*. Все классы определены в пространстве имен *System.Threading* и реализованы в библиотеке MSCorLib.dll. Исключение — класс *Semaphore*, который реализован в библиотеке System.dll. Иерархия классов выглядит следующим образом:

```
WaitHandle
   Mutex
   Semaphore
   EventWaitHandle
      AutoResetEvent
      ManualResetEvent
```

В базовом классе *WaitHandle* есть поле, содержащее Win32-описатель объекта ядра. Это поле инициализируется при конструировании класса, производного от *WaitHandle*. Кроме этого, класс *WaitHandle* предоставляет открытые методы, единые для всех производных неабстрактных классов. Приведенное ниже псевдоопределение класса описывает все интересные открытые методы класса *WaitHandle* (не показаны некоторые перегруженные версии отдельных методов):

```
public abstract class WaitHandle : MarshalByRefObject, IDisposable {
   // Код метода Close вызывает Win32-функцию CloseHandle.
   public virtual void Close();

   // Код метода WaitOne вызывает Win32-функцию WaitForSingleObjectEx.
   public virtual Boolean WaitOne();
   public virtual Boolean WaitOne(
      Int32 millisecondsTimeout, Boolean exitContext);

   // Код метода WaitAny вызывает Win32-функцию WaitForMultipleObjectsEx.
   public static Int32 WaitAny(WaitHandle[] waitHandles);
   public static Int32 WaitAny(WaitHandle[] waitHandles,
      Int32 millisecondsTimeout, Boolean exitContext);

   // Код метода WaitAll вызывает Win32-функцию WaitForMultipleObjectsEx.
   public static Boolean WaitAll(WaitHandle[] waitHandles);
   public static Boolean WaitAll(WaitHandle[] waitHandles,
      Int32 millisecondsTimeout, Boolean exitContext);

   // Код метода 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);

   public const Int32 WaitTimeout = 0x102;
}
```

По поводу этих методов есть несколько замечаний:

- Можно вызвать метод *Close* класса *WaitHandle* (или не имеющий параметров метод *Dispose* интерфейса *IDisposable*), чтобы закрыть соответствующий описатель объекта ядра. Внутри этих методов вызывается Win32-функция *CloseHandle*.
- Можно вызвать метод *WaitOne* класса *WaitHandle*, чтобы вызывающий поток ожидал освобождения соответствующего объекта ядра. Внутри метода вызывается Win32-функция *WaitForSingleObjectEx*. Если объект освободился, возвращается значение *true*. В случае тайм-аута возвращается *false*.

- Можно вызвать метод *WaitAny* класса *WaitHandle*, чтобы вызывающий поток ожидал освобождения одного из объектов ядра, указанных в параметре *WaitHandle[]*. Возвращаемое значение *Int32* является индексом элемента массива, соответствующего освободившемуся объекту ядра или *WaitHandle.WaitTimeout*, если за время тайм-аута ни один объект не освободился. Внутри этого метода вызывается Win32-функция *WaitForMultipleObjectsEx*, передающая в параметре *bWaitAll* значение *false*.
- Можно вызвать метод *WaitAll* класса *WaitHandle*, чтобы заставить вызывающий поток ожидать освобождения всех объектов ядра, указанных в параметре *WaitHandle[]*. Если все объекты освобождены, возвращается логическое значение *true*, в противном случае (не все объекты освобождены) — *false*. Внутри этого метода вызывается Win32-функция *WaitForMultipleObjectsEx*, передающая в параметре *bWaitAll* значение *true*.
- Статический метод *SignalAndWait* класса *WaitHandle* вызывается, чтобы атомарно освободить один объект ядра и ожидать освобождения другого объекта. Если объект освободился, возвращается значение *true*. В случае тайм-аута возвращается *false*. Внутри этого метода используется Win32-функция *SignalObjectAndWait*.

Версии *WaitOne*, *WaitAll* и *SignalAndWait*, которые не принимают параметр тайм-аута, должны прототипироваться как возвращающие *void*, а не *Boolean*. В противном случае эти методы будут возвращать только значение *true*, поскольку по умолчанию время ожидания не ограничено. При вызове любого из этих методов проверять возвращаемое значение не нужно.

Версии *WaitOne*, *WaitAll*, *WaitAny* и *SignalAndWait*, не принимающие параметр *exitContext*, подставляют в него значение по умолчанию — *false*. CLR позволяет разместить объект в собственном контексте, применив к классу методы *System.Runtime.Remotin.Contexts.ContextAttribute* или *System.Runtime.Remoting.Contexts.SynchronizationAttribute*. Контекст можно сравнить с мини-доменом AppDomain или мини-окружением для экземпляров класса. В любом случае эти атрибуты имеют отношение к удаленному доступу, а использование этих функций сейчас не рекомендуется. При вызове одного из методов класса *WaitHandle*, принимающего параметр *exitContext*, обычно передают значение *false*.

Классы *Mutex*, *Semaphore*, *AutoResetEvent* и *ManualResetEvent* являются производными от *WaitHandle*, поэтому наследуют поведение этого родительского класса. В этих классах появились некоторые дополнительные методы, о которых мы сейчас поговорим.

Во-первых, код конструкторов всех этих классов вызывает Win32-функции *CreateMutex*, *CreateSemaphore*, *CreateEvent* (передавая *false* в параметре *bManualReset*) или *CreateEvent* (передавая *true* в параметре *bManualReset*). Возвращаемый всеми вызовами описатель сохраняется в закрытом поле, определенном в базовом классе *WaitHandle*.

Во-вторых, в классах *Mutex*, *Semaphore* и *EventWaitHandle* есть статические методы *OpenExisting*, их внутренний код вызывает Win32-функции *OpenMutex*, *OpenSemaphore* или *OpenEvent*, передавая аргумент *string*, в котором указывается существующий именованный объект ядра. Возвращаемый всеми этими методами описатель сохраняется в новом объекте, который и возвращает метод *OpenExisting*.

Если объекта ядра с указанным именем не существует, генерируется исключение *WaitHandleCannotBeOpenedException*.

В-третьих, чтобы освободить объект ядра *Mutex* или *Semaphore*, нужно вызвать метод *ReleaseMutex* из класса *Mutex* или *Release* — из класса *Semaphore*. Чтобы установить или сбросить событие объекта ядра с автоматическим или ручным сбросом, нужно вызывать (унаследованные от *EventWaitHandle*) методы *Set* или *Reset* класса *AutoResetEvent* или *ManualResetEvent*. Заметьте: прототипы методов *Set* или *Reset* класса *EventWaitHandle* определяют в качестве возвращаемого типа *Boolean*, но они всегда возвращают значение *true*, поэтому нет необходимости писать код, проверяющий возвращаемое этими методами значение.

Вызов метода при освобождении одного объекта ядра

Исследуя производительность, специалисты Microsoft обнаружили, что во многих приложениях создают потоки, которые просто ожидают освобождения одного объекта ядра. При освобождении объекта поток направляет определенное уведомление другому потоку, после чего опять ожидает освобождения объекта. Некоторые разработчики даже пишут код, в котором несколько потоков ожидает освобождения объекта. Это очень нерациональная трата системных ресурсов. Так что, если у вас есть потоки, ожидающие освобождения одного объекта ядра, и в этот раз лучшим решением для повышения производительности приложения будет использование пула потоков.

Чтобы поток из пула потоков вызывал нужный метод обратного вызова при освобождении объекта ядра, нужно вызвать статический метод *RegisterWaitForSingleObject* класса *System.Threading.ThreadPool*. Есть несколько перегруженных версий этого метода, но все они очень похожи. Вот прототип одной из наиболее часто используемых перегруженных версий:

```
public static RegisterWaitHandle 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 указать любой производный от него класс. В частности, можно передать ссылку на объект *Semaphore*, *Mutex*, *AutoResetEvent* или *ManualResetEvent*. Второй параметр, *callback*, указывает метод, который поток из пула потоков должен вызывать. Вызываемый метод должен соответствовать делегату *System.Threading.WaitOrTimerCallback*, который определяется следующим образом:

```
public delegate void WaitOrTimerCallback(Object state, Boolean timedOut);
```

Третий параметр метода *RegisterWaitForSingleObject*, *state*, позволяет указать некоторые данные состояния, которые нужно передать в метод обратного вызова, когда поток из пула потоков вызывает его. Если данных нет, передайте *null*. Четвертый параметр, *millisecondsTimeoutInterval*, указывает пулу, как долго он должен ожидать освобождения объекта ядра. Чтобы задать неограниченное время ожидания, обычно передают *Timeout.Infinite* или -1. Если последний параметр, *executeOnlyOnce*, равен *true*, пул потоков выполнит метод обратного вызова только раз.

Если же *executeOnlyOnce* равен *false*, поток из пула потоков будет выполнять метод обратного вызова при каждом освобождении объекта ядра. Это исключительно удобно при ожидании освобождения объекта *AutoResetEvent*.

Методу обратного вызова передаются данные состояния и логическое значение *timedOut*. Если *timedOut* равен *false*, метод знает, что он вызван по причине освобождения объекта ядра. Если же значение *timedOut* равно *true*, метод знает, что он вызывается потому, что объект ядра не освободился 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 потоков, что нужно прекратить наблюдение за ожидаемым описателем, вызывают метод *Unregister* класса *RegisteredWaitHandle*:

```
public Boolean Unregister(WaitHandle waitObject);
```

Значение параметра *waitObject* указывает, как нужно вернуть уведомление о выполнении всех рабочих элементов в очереди зарегистрированного объекта ожидания. Если уведомление не требуется, передайте *null*. Если надо передать корректную ссылку на объект, производный от класса *WaitHandle*, пул потоков сообщит объекту о выполнении всех рабочих элементов.

Приведенный ниже пример кода показывает, как поток из пула потоков вызывает метод каждый раз, когда освобождается объект *AutoResetEvent*:

```
using System;
using System.Threading;

public static class Program {
   public static void Main() {
      // Создание объекта AutoResetEvent (изначально не освобожденного).
      AutoResetEvent are = new AutoResetEvent(false);

      // Сообщаем пулу потоков, что нужно ожидать AutoResetEvent.
      RegisteredWaitHandle rwh = ThreadPool.RegisterWaitForSingleObject(
         are,                // Ожидать этот объект AutoResetEvent.
         EventOperation,     // Выполнить обратный вызов этого метода.
         null,               // Передать null в качестве параметра EventOperation.
         5000,               // Ждать освобождения 5 секунд.
         false);             // Вызывать EventOperation при каждом освобождении.

      // Запуск цикла.
      Char operation;
      do {
         Console.WriteLine("S=Signal, Q=Quit?");
         operation = Char.ToUpper(Console.ReadKey(true).KeyChar);
         if (operation == 'S') {
            // Пользователь хочет освободить событие; устанавливаем событие.
            are.Set();
```

```
         }
   } while (operation != 'Q');

      // Сообщаем пулу потоков, что нужно прекратить ожидание события.
      rwh.Unregister(null);
   }

   // Этот метод вызывается при каждом освобождении события или
   // через 5 секунд после последнего освобождения/тайм-аута.
   private static void EventOperation(Object state, Boolean timedOut) {
      if (timedOut) {
         Console.WriteLine("Timedout while waiting for the AutoResetEvent.");
      }  else {
         Console.WriteLine("The AutoResetEvent became signaled.");
      }
   }
}
```