Алексей Дубовцев

bhv

Microsoft® .NET

- Архитектура и внутреннее устройство платформы .NET
- Недокументированные особенности
- Более 1000 примеров на языках C#, VB.NET, MC++, IL

Наиболее полное руководство

+CD

В ПОДЛИННИКЕ®

Алексей Дубовцев

Microsoft® .NET в подлиннике

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.06
ББК 32.973.018.2
Д79

Дубовцев А. В.

Д79 Microsoft .NET в подлиннике / Под ред. В. Е. Пышкина. — СПб.: БХВ-Петербург, 2004. — 704 с.: ил.

ISBN 5-94157-478-9

Рассмотрены теоретические основы и практические приемы программирования на платформе .NET с использованием популярных языков C#, VB.NET, MC++, IL. Описаны метаданные, общая система типов, сборки, архитектура доменов, атрибуты и др. На большом количестве простых и понятных примеров рассмотрены обработка исключений, делегаты и события, потоки и др. Дано подробное представление низкоуровневого взаимодействия с операционной системой из среды .NET. Прилагается компакт-диск, содержащий большое количество примеров на языках C#, VB.NET, MC++, IL.

Для программистов

УДК 681.3.06
ББК 32.973.018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Василий Плевалов*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.08.04.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 56,76.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-478-9

Содержание

Моим родителям

Речь верная красивой не бывает,
Красивые слова не убеждают.

Достоинство себя не в споре выражает,
Кто спорит — истиной не обладает.

Познанье истины — не есть образование,
А сумма сведений — еще не знание.

Кто тупо копит знания, ума не проявляет.
Мудрец, чем больше отдаёт, тем больше обретает.

Как Дао Неба — миру жизнь давать, не принося вреда,
Так Дао Мудрого — творить и не вступать в дебаты никогда.

Лао-цзы, "Дао де Цзин"

Мои искренние Благодарности

Воробьеву Дмитрию Анатольевичу — спасибо за то, что научили думать и указали путь.

Темнову Дмитрию Эдуардовичу — моему первому учителю, плюс как человеку, руками которого была написана моя первая программа — (print "Hello, World").

Силанову Виктору Алексеевичу — учителю, который приложил много личных усилий, для того чтобы сделать из меня профессионала.

Плещенкову Николаю Мироновичу — моему любимому школьному учителю.

Всей команде rsdn.ru — сайта, для которого я написал свою первую статью и с которого началась моя авторская работа.

Рыбакову Евгению — за то, что сподвигнул меня написать эту книгу.

Шишигину Игорю — за внимательность и понимание при подготовке книги.

Пышкину Евгению Валерьевичу — за отзывчивость и неоценимую профессиональную помощь.

Ложечкину Александру — за доверие и прекрасную аннотацию.

Шостенко Ирочке — за неоценимые советы по стилистике русского языка.

Романову Андрею и **Голубеву Дмитрию** — за те советы и замечания, которые они давали при подготовке книги.

Нанобашвили Вячеславу — за мудрые и жизненные советы.

Всему издательскому дому **BHV**, за понимание и веру.

Всему **преподавательскому составу** кафедры физической электроники факультета радиофизики Санкт-Петербургского государственного политехнического университета — за понимание и высококлассное профессиональное преподавание. В особенности Фотиади Александру Эпоминондовичу и Кораблеву Вадиму Васильевичу.

Особые благодарности: **Богачеву Филиппу**, **Титову Алексею**, **Вадиму Георгиевичу**, **Бучину Сереже**, **Ширину Мише, Лыкосову Сергею** за те инструменты и знания, которые они передали в мои руки.

Плюс теплые приветы **Диме Найдичу**, **Славе Ардентову**, **Ступину Вите**, **Антону Зеленскому**, **Саше Волкову**, **Коле Воробьеву** и всем остальным.

И наконец, привет моим **друзьям** и спасибо за то, что вы у меня есть:

Захаренкову Мише, Фазлееву Диме, Василевичу Диме, Шиповникову Андрею, Клищенко Саше, Сучкову Жоре, Сане Иванову, Афониной Ольге, Мартюшину Алексею, Кузнецову Андрею, Андрееву Юре.

Моим любимым **родственникам**:

Дубовцевым Валерию Федоровичу и **Вере Васильевне** — моим любимым дедушке и бабушке.

Топорской Оксане — моей любимой тетке.

Топорскому Сереже — моему лучшему дяде, который многому меня научил, всегда был внимателен и добр.

Топорской Сашке — самой моей любимой сестре на свете.

Селиверстову Игорю и всей его дружной семье — за любовь и понимание.

Селиверстову Сергею — за то, что научил работать с литературой, был добр и терпелив.

Персональный привет всей моей **Нижегородской семье**, **особенно тете Лизе**.

❤ **Дианке**...

Предисловие научного редактора

Перед читателем — написанная молодым автором книга, обобщающая опыт изучения им архитектуры и концептуальных элементов платформы Microsoft .NET Frameworks, обсуждение и использование которой стало едва ли не самым заметным явлением последних лет в области маркетинга средств разработки программного обеспечения. Следуя общеизвестному тезису "чтобы хорошо изучить какой-либо предмет или явление, напишите об этом книгу", автор провел впечатляющую работу по исследованию внутренних механизмов .NET, обеспечивающих такие решения, как поддержка компонентного программирования средствами языка C#, совместимость компонентов, разработанных на разных языках программирования, построение устойчивого, переносимого и безопасного кода, механизмы распространения кода с использованием сборок, организация взаимодействия управляемого и неуправляемого кода и т. д. С точки зрения содержания книги, весьма удачным представляется выбранное для книги название.

Несмотря на то, что книга посвящена в первую очередь рассмотрению аспектов *реализации* механизмов и концепций, лежащих в основе платформы .NET, и поэтому ориентирована на подготовленного читателя, отдельные разделы книги будут интересны и начинающим программистам. В частности, заслуживает быть отмеченным обсуждение фундаментального понятия "тип". Его подробное рассмотрение, вероятно, несколько выходит за рамки основной тематики книги, но при этом представляет собой весьма удачный пример понятного и наглядного объяснения трудных для понимания начинающих программистов концепций программирования. Примерами разделов, написанных на стыке общего знания и изучения механизмов и концепций .NET, могут также служить главы "Потоки" и "Архитектура доменов", от чтения которых удовольствие должны получить и опытные разработчики, и начинающие программисты.

В первой части книги обсуждаются фундаментальные концепции, лежащие в основе программирования на платформе .NET. В частности, подробно анализируются особенности исполнения управляемого кода, назначение

и реализация метаданных, используемых в ходе работы с управляемым кодом, организация и принципы работы исполняющей системы .NET. Подробно исследуются принципы построения общей системы типов и механизмы обеспечения межъязыковой интеграции в приложениях .NET.

Во второй части книги особое внимание автор уделяет новаторским элементам программирования, реализованным в рамках платформы .NET, таким как реализация механизма косвенных вызовов посредством делегатов; поддержка модели программирования, ориентированного на события встроенными средствами языка C#; реализация механизма использования информации времени разработки в ходе исполнения программы посредством специализированных атрибутов.

В третьей части детально обсуждаются аспекты работы с .NET в связи с управлением приложениями и механизмы реализации взаимодействия компонентов .NET. Особого внимания заслуживает изложение решений, лежащих в основе обеспечения взаимодействия с неуправляемым кодом.

Несмотря на то, что обозреваемая книга является первым столь объемным трудом автора, он предстает перед нами сложившимся специалистом со своим взглядом на вещи, открытым для конструктивного и благожелательного обсуждения отдельных разделов проделанной им большой работы. Неформальный и слегка ироничный стиль автора удачно оттеняет сложные для чтения разделы. В тех случаях, когда, по моему мнению, тот или иной момент заслуживал немного более строгого и "сухого" замечания, по согласию с автором и издательством, в текст добавлялось примечание научного редактора. Впрочем, во всей книге их всего около десятка.

Итак, вниманию читателя представляется достойное издание, которое, надеюсь, принесет заслуженную популярность его автору и станет удачным приобретением для читателей.

Пышкин Е. В.,
кандидат технических наук,
доцент кафедры автоматики
и вычислительной техники
Санкт-Петербургского государственного
политехнического университета

Глава 1

Общеязыковая среда исполнения

Общеязыковая среда исполнения .NET — новая мощная среда разработки от Microsoft, обеспечивающая уникальные возможности по межпрограммной и межплатформенной интеграции программных средств. Для ее реализации была разработана единая концепция использования управляемого кода. В главе приводятся сведения об основных принципах построения платформы .NET, рассматриваются внутреннее устройство среды исполнения, общие принципы работы виртуальных машин, а также природа и устройство управляемого кода. Прочитав эту главу, вы узнаете, для чего создавалась среда исполнения и какие задачи она способна решить.

1.1. Архитектура среды исполнения .NET

Общеязыковая среда исполнения (Common Language Runtime, CLR) — это общая независимая виртуальная программная среда для приложений, обеспечивающая эффективное и контролируемое исполнение пользовательского кода, а также предоставляющая разработчикам программных систем широкий спектр разнообразных сервисов. Она существенно облегчает разработку современных приложений, сочетая в себе многие преимущества технологии RAD (Rapid Application Development, Быстрая Разработка Приложений), с высокой производительностью и скоростью работы конечных приложений. Структурно общеязыковая среда исполнения .NET состоит из двух компонентов: виртуальной машины и общей библиотеки классов (рис. 1.1).

Виртуальная машина отвечает за исполнение пользовательского кода, а общая библиотека классов предоставляет разработчикам четко стандартизированный набор типов, предоставляющих самые разнообразные сервисы. Исходный язык программирования, на котором написано приложение для среды .NET, не имеет значения; от транслятора языка лишь требуется умение генерировать "управляемый" код, который понимает виртуальная машина .NET. Такой код работает только в среде CLR, что обеспечивает полный контроль над всеми процессами при исполнении приложения.

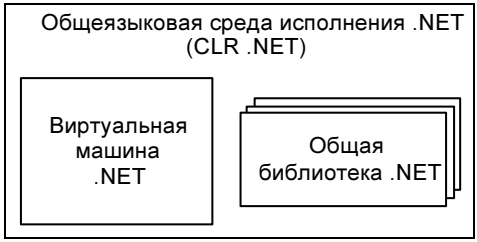


Рис. 1.1. Состав общеязыковой среды исполнения

Управляемый код — новое понятие, введенное разработчиками Microsoft, и которое поначалу вызывает затруднения для понимания у большинства программистов. Его особенностью является то, что он выполняется под постоянным надзором среды исполнения, которая контролирует все его шаги и опекает, как чуткий родитель своего ребенка. Именно из-за столь сильного контроля и надзора было решено назвать код управляемым. Он представляется в специальном формате, называемом .NET байт-код. Для визуального представления внутреннего .NET байт-кода разработан язык IL (Intermediate Language, Промежуточный Язык Microsoft). По сути дела, он находится на одном уровне с .NET байт-кодом, но это не сам байт-код, а лишь его текстовое представление. Для сравнения можно сказать, что IL и .NET байт-код соотносятся, как ассемблер и машинный код. Таким образом, все приложения поставляются для исполнения на виртуальной машине .NET в виде унифицированного .NET байт-кода.

Примечание научного редактора

По прочтении предшествующего текста у читателя может сложиться впечатление, что концепция управляемого кода как таковая является изобретением Microsoft. Это, разумеется, не так (хотя, пожалуй, стоит согласиться, что более или менее новым является сам термин "управляемый код").

Поэтому следует заметить, что сама идея создания кода, ориентированного на исполнение в рамках специальной среды, обусловлена необходимостью создания систем программирования, ориентированных на создание мобильного безопасного устойчивого кода. При этом под мобильностью кода понимается возможность выполнения одной и той же программы на разных вычислительных платформах. Под безопасностью кода понимается встроенная в язык защита от несанкционированного доступа к ресурсам компьютера, на котором исполняется приложение (в особенности, когда речь идет о сетевых приложениях). Устойчивость кода предполагает возможность обнаружения ошибок на возможно более ранних стадиях проектирования и исключение определенных классов ошибок (например, ошибок управления памятью и необрабатываемых исключительных ситуаций). Устойчивость кода предполагает также по возможности наиболее полный контроль корректности операций, совершаемых в процессе исполнения программы (преобразования типов, инициализация объектов, исключительные ситуации в связи с арифметическими вычислениями и т. д.).

Основной механизм, позволяющий обеспечить получение кода, обладающего такими свойствами, основывается на реализации специальной исполняющей системы, работающей не с кодом процессора, а с некоторым промежуточным кодом, который и является выходом компиляции исходного текста программы. Впечатляющий успех применения подобной организации обработки программы выпал на долю технологии Java. В основе решения проблем построения мобильного, безопасного и устойчивого программного обеспечения с помощью Java лежит трансляция исходного кода в промежуточное представление (байт-код) и его выполнение специальной исполнительной системой, называемой виртуальной машиной Java (Java Virtual Machine – JVM).

Код, разрабатываемый на языке, ориентированном на исполнение в контролирующей среде, называют управляемым, или контролируемым, кодом (managed code). Итак, по существу байт-код Java является примером управляемого кода. В этой терминологии, C/C++ генерирует неуправляемый код (unmanaged code), то есть код, ориентированный непосредственно на процессор.

Для создания низкоуровневого байт-кода из высокоуровневого языка, используются соответствующие компиляторы. Создавать приложения .NET возможно более чем на десяти различных языках программирования высокого уровня. При этом общие стандарты среды исполнения .NET, в лице общеязыковой спецификации CLS (Common Language Specification) и общей системы типов CTS (Common Type System), будут гарантировать интеграцию кода вне зависимости от исходного языка, использованного для создания приложения.

Интеграция кода подразумевает полную совместимость кода на всех уровнях: начиная от стандартов определения типов и заканчивая моделью поддержки асинхронных событий. К примеру, вы можете определить класс при помощи языка C#, а унаследовать его, используя Visual Basic .NET (VB .NET).

Особенностью системы .NET является использование оригинальной технологии интеграции кода, которая обеспечивает совместимость кода не на уровне исполняющего ядра (процессора, виртуальной машины .NET), а на уровне самой программной модели. Ранее, когда создавался код для процессоров с использованием ортодоксальных компиляторов, основной задачей программистов было обеспечение лишь односторонней совместимости с самим процессором, разработчики которого вводили собственные стандарты на машинный код, практически не думая о пользователях языков высокого уровня. В результате, сформированный машинный код представлял всего лишь последовательный набор команд для процессора, который предписывал ему совершить определенные действия.

.NET байт-код представляет собой не просто набор команд для виртуальной машины .NET. Он качественно отличается от машинного кода наличием полного описания внутренней структуры программы, начиная от локальной переменной и заканчивая классами и пространствами имен. Фактически, при классической компиляции в машинный код полностью теряется информация о внутреннем устройстве программы, которая по большому счету

и не нужна для ее исполнения. Зато такая информация жизненно необходима при решении задач межъязыковой и межпрограммной интеграции. При компиляции в управляемый .NET байт-код потерь этой информации не происходит и полное представление о внутренней организации программы сохраняется в специальных структурах, называемых метаданными.

Таким образом, технология управляемого байт-кода дает огромные преимущества в виде межъязыковой интеграции и кросс-платформенной переносимости кода. Но все преимущества подхода могут быть сведены к минимуму низкими показателями производительности из-за полной виртуализации исполнения байт-кода. Практическим подтверждением тому могут служить Java (классическая машина от Sun) и Microsoft Visual Basic .NET (ранних версий). Производительность при выполнении кода обеих систем оставляет желать много лучшего.

Существует два вида виртуальных машин — интерпретирующие и компилирующие. Интерпретирующие машины исполняют код, полностью эмулируя его работу. Фактически, они сами исполняют код вверенных им приложений. То есть на каждую инструкцию интерпретируемого байт-кода приходится N инструкций интерпретатора виртуальной машины. В итоге, происходит значительное снижение скорости выполнения программы. За что и не любят виртуальные машины такого типа.

Платформа .NET является виртуальной машиной принципиально другого типа. Это компилирующая виртуальная машина нового поколения. Подобные виртуальные машины не занимаются интерпретацией байт-кода, они компилируют его в машинный код платформы, на которой в данный момент происходит исполнение приложения (и всей виртуальной машины). При подобном подходе мы получаем на одну инструкцию байт-кода соответствующее количество машинных инструкций. Причем эти машинные инструкции будут соответствовать непосредственно исполняющейся программе, а не заранее скомпилированному коду виртуальной машины. При таком подходе конечный машинный код, исполняющийся на процессоре, получается более компактным и быстродействующим, чем обобщенный код интерпретирующей машины.

Примечание

Технологии компилируемых виртуальных машин разрабатывались и в России, в рамках проекта вычислительной системы "Эльбрус". В 2001 г. в статье Бориса Бабаяна "Основные Принципы Архитектуры E2K" **(http://www.mcst.ru/mcst/e2k.pdf)** подробно были описаны принципы функционирования такой технологии и рассмотрены преимущества ее применения. Команда Бабаяна разработала двоичный компилятор, который позволил запускать Windows и игровую программу Microsoft Flight Simulator на платформе Sun с весьма приличными показателями по скорости. Изучая виртуальную машину Microsoft .NET, никак не избавиться от ощущения того, что она построена по принципам, разработанным группой Бабаяна.

Для реализации подобного подхода в виртуальной машине требуется наличие очень важного компонента — компилятора времени исполнения JIT (Just-In-Time Compiler), который обеспечивает трансляцию .NET байт-кода в машинный прямо по ходу исполнения программы. Такие компиляторы обычно называют двоичными. По всем тестам на производительность, проведенным для платформы .NET, она практически ни в чем не уступает классическим компилирующим системам, а по некоторым — даже превосходит их. С подробными отчетами этих тестов можно ознакомиться на сайте **http://www. rsdn.ru**.

Среда исполнения .NET может базироваться на различных аппаратных (Intel, Pocket PC) и программных платформах (Windows, FreeBSD, MacOS, Linux). Для каждой поддерживаемой аппаратной и программной платформы должна существовать собственная реализация подобного компилятора и для каждой новой аппаратной платформы разработчикам придется создавать собственный JIT-компилятор, который будет переводить универсальный независимый .NET байт-код в конкретный машинный код. В этом заключается, пожалуй, основной недостаток этой технологии.

В общем виде внутреннее устройство виртуальной машины .NET можно представить следующим образом (рис. 1.2).

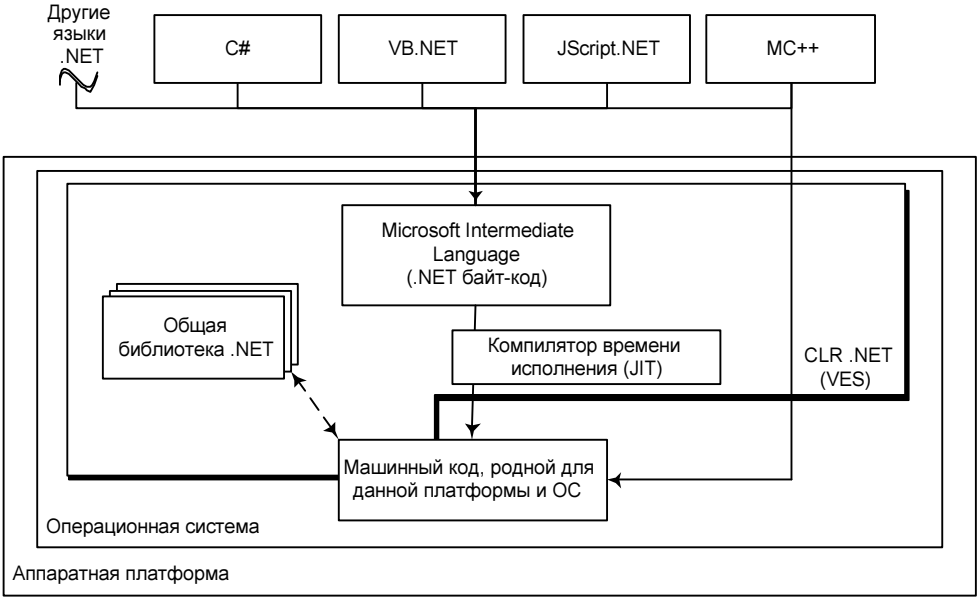


Рис. 1.2. Общая схема виртуальной исполняющей машины .NET

Все это вкупе носит название виртуальной исполняющей машины (VES, Virtual Execution System). Впоследствии, перед выходом платформы .NET "в свет" она была переименована в CLR. На самом верху системы находятся

языки высокого уровня. Программы, написанные на них, транслируются в байт-код платформы .NET, который может быть представлен в виде IL. Он же, во время исполнения программы, компилируется в машинный код аппаратной платформы. Но даже, несмотря на то, что исполнение программ происходит в чистом машинном коде, оно полностью контролируется со стороны виртуальной машины .NET. Достигается это за счет использования служебного контролирующего кода.

В итоге, мы получаем полное абстрагирование от уровня аппаратных и программных платформ. И, как следствие, полную унификацию предоставляемых сервисов, независимо от целевой платформы. К примеру, если предполагается в приложении .NET использовать потоки, то нас абсолютно не будут волновать детали реализации механизмов поддержки многопоточности в операционных системах, в среде которых будет исполняться наше приложение. С точки зрения разработчика, оно будет одинаково исполняться как на Windows системах, так и на FreeBSD.

Примечание

В действительности полной унификации предоставляемых сервисов для всех программных платформ, к сожалению, не наблюдается. Многие необходимые сервисы доступны только на платформах семейства Windows. Для примера, можно привести типы из пространства Windows.Forms, предназначенные для создания пользовательского интерфейса. Но это можно отнести скорее к нюансам маркетинговой политики компании, чем к принципиальным конструктивным недоработкам. Хотя необходимо отметить, что существуют реализации недостающих сервисов от сторонних разработчиков.

Таким образом, для программ .NET создается свой закрытый виртуальный мир, заглянуть за пределы которого им не разрешается. Однако нередко появляется необходимость доступа к специфичным программным и аппаратным ресурсам, которые могут быть не отражены в общей библиотеке классов .NET. Для разрешения этой проблемы разработчики .NET создали технологию исполнения совместного кода в рамках компилятора Managed Extensions for C++ (MC++, Управляемые Расширения для C++). Компилятор MC++ позволяет в одном файле комбинировать как управляемый, так и неуправляемый код. Задача выбора типа генерируемого кода возлагается полностью на программиста. Технология работы компилятора показана на рис. 1.3.

В листинге 1.1 приведен пример интеграции управляемого и неуправляемого кодов в рамках одной программы.

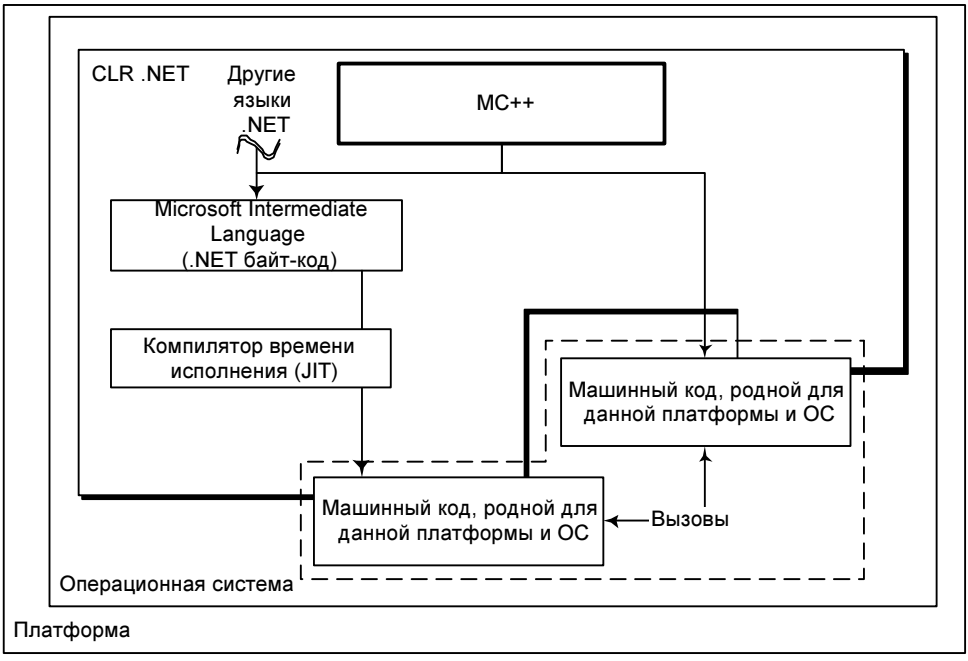


Рис. 1.3. Технология совместного кода, воплощенная в MC++

Листинг 1.1. Симбиоз управляемого и неуправляемого кода

```
/*
  Листинг 1.1
  File:   Some.cpp
  Author: Дубовцев Алексей
*/
// Подключим общую библиотеку классов .NET.
#using <mscorlib.dll>
// Подключим стандартную библиотеку.
#include <stdio.h>
// Директива компилятору на генерацию неуправляемого кода.
#pragma unmanaged
// Эта функция будет неуправляемой, она будет транслирована
// в чистый машинный код.
void UnmanagedFunction()
{
  printf("Hello, World, from UnmanadegFunction!");
```

```
}
// Прикажем компилятору вновь генерировать управляемый код.
#pragma managed
// Это управляемая функция, она будет транслирована в набор инструкций
// .NET байт-кода.
void ManagedFunction()
{
  // Прикажем компилятору использовать управляемую строку.
  System::Console::WriteLine(S"Hello, World, from ManagedFunction!");
  UnmanagedFunction();
}
// Эта функция также будет управляемой.
void main()
{
  ManagedFunction();
}
```

Такой тесный симбиоз управляемого и неуправляемого кода обеспечивает сочетание в одном приложении мощных и удобных средств .NET и наработанных ранее проектных решений. Помимо прямого взаимодействия с операционной системой из неуправляемого кода, среда исполнения предоставляет систему шлюзов, которые позволяют прямо из управляемого кода обращаться к сервисам операционной системы. При этом все обращения на нижний уровень будут контролироваться со стороны системы политики безопасности среды .NET, чего не скажешь о прямом использовании низкоуровневого платформенного кода, который потенциально может творить "что его душе угодно".

Для решения столь серьезных задач среда исполнения предоставляет разработчикам множество инструментов, важнейшими из которых являются:

- система автоматического управления памятью;
- прозрачная межъязыковая интеграция;
- автоматическое управление кодом на основе сборок;
- прозрачная межплатформенная совместимость;
- защита исполняемого кода;
- богатая, мощная и оптимизированная библиотека классов;
- сервисы взаимодействия с операционной системой;
- система автоматического контроля целостности кода;
- автоматическая система защиты переполнения буфера.

1.2. Общая библиотека классов

Общая библиотека классов .NET от Microsoft (Framework Class Library, FCL) мощна и многообразна и предоставляет пользователю огромный спектр сервисов. Перечислить их полностью не представляется возможным, приведем лишь некоторые из них (табл. 1.1).

Таблица 1.1. *Набор основных сервисов, предоставляемых через общую библиотеку классов*

Пространство Имен	Сервисы класса
`System`	Базовые типы, используемые всеми приложениями
`System.Collections`	Коллекции для хранения наборов объектов
`System.Diagnostics`	Сервисы для разработки и отладки приложений
`System.Drawing`	Сервисы для работы с двумерной графикой
`System.EnterpriseServices`	Поддержка транзакций, очередей компонентов, пулов объектов, JIT-активации и других серверных функций
`System.Globalization`	Поддержка региональных стандартов и системы локализации приложений
`System.IO`	Поддержка потокового ввода вывода, в том числе и файлового
`System.Managment`	Сервисы управления компьютерами при помощи WMI (Windows Management Instrumentation, ...)
`System.Net`	Поддержка сетевого взаимодействия
`System.Reflection`	Сервисы отражения, предоставляющие возможность динамического взаимодействия с программным кодом
`System.Resources`	Сервисы для работы с ресурсами
`System.Runtime.InteropServices`	Сервисы, предоставляющие шлюзы для взаимодействия с операционной системой
`System.Runtime.Remoting`	Сервисы, обеспечивающие удаленный доступ к типам
`System.Runtime.Serialization`	Сервисы, обеспечивающие сериализацию (упаковку) объектов в универсальные потоки данных, с возможностью их последующего восстановления

Таблица 1.1 *(окончание)*

Пространство Имен	Сервисы класса
`System.Security`	Сервисы поддержки защиты и безопасности программного кода
`System.Text`	Сервисы для работы с текстом
`System.Threading`	Сервисы поддержки многопоточного кода
`System.Xml`	Сервисы для работы с данными в формате XML
`System.Web.Services`	Средства, предназначенные для создания Web-сервисов XML
`System.Web.UI`	Сервисы для удобного создания Web-пользовательских интерфейсов (Web Forms)
`System.Windows.Forms`	Сервисы поддержки классических пользовательских интерфейсов
`System.ServiceProcess`	Средства поддержки работы с сервисами операционной системы, ответственные за взаимодействие со SCM (Service Control Manager, Менеджер управления сервисами)

Даже при первом рассмотрении набор предоставляемых возможностей впечатляет. А когда начинаешь знакомиться ближе, поражает стройность архитектуры библиотеки. Кажется, что в ней нет ничего лишнего и бесполезного. Общая библиотека классов жестко привязана к среде исполнения — она является ее неотъемлемой частью. В этом есть как свои плюсы, так и минусы. К плюсам можно отнести уменьшение размеров конечных приложений и оптимизацию библиотеки с повышением версии среды исполнения. К минусам относится возможная потеря совместимости сервисов общей библиотеки для различных версий среды исполнения.

С учетом требований к быстродействию кода общей библиотеки классов значительная ее часть написана на языке C++, который транслируется в чистый машинный код. Для этого была использована специальная технология шлюзов, обеспечивающая переход на уровень самой виртуальной машины. При вызове некоторых методов классов общей библиотеки, управление напрямую передается в виртуальную машину, где этот метод реализован в чистом машинном коде.

1.3. .NET байт-код и язык представления кода IL

Рассмотрим вопрос о соотношении между .NET байт-кодом и промежуточным языком IL. Здесь уместна аналогия из мира ортодоксального программирования, когда процессор обрабатывает двоичные машинные коды, а программист пишет и читает их на языке ассемблера. Виртуальная машина также занимается исполнением чистого .NET байт-кода и не имеет ни малейшего

понятия о IL, который служит для удобного представления этого кода в текстовом виде. Пример приложения на языке IL приведен в листинге 1.2.

Листинг 1.2. Пример приложения на языке IL

```
/*
       Листинг 1.2
       File:   Some.il
       Author: Дубовцев Алексей
*/
// Подключим общую библиотеку классов .NET.
.assembly extern mscorlib {}
// Зададим общее имя нашей сборки.
.assembly Some {}
// Это основная функция нашего приложения.
.method static public void GeneralMethod()
{
  // Указываем на то, что данная функция является точкой
  // входа в приложение.
  .entrypoint
  // Устанавливаем размер стека, для того чтобы поместить
  // туда ссылку на строку.
  .maxstack 1
  // Помещаем ссылку на строку в стек.
  ldstr "Hello, World!"
  // Выводим строку на консоль
  call void [mscorlib]System.Console::WriteLine(string)
  // Выходим из функции
  ret
}
```

Это простой текстовый файл, который, безусловно, не может исполняться виртуальной машиной .NET. Для этого необходимо представленный исходный код транслировать в .NET байт-код при помощи компилятора `ilasm`. Эта операция осуществляется при помощи следующей команды:

```
ilasm Some.il
```

После завершения компиляции формируется исполняемый .NET-файл, содержащий реальный байт-код. Именно он будет исполняться виртуальной машиной .NET, а отнюдь не программой на языке IL. IL — это лишь средство представления байт-кода в наглядном виде.

1.4. Исходный код CLI

Как известно, Microsoft обычно не предоставляет исходные тексты своих программных продуктов. Однако ни для кого не секрет, что в мире постоянно растет число приверженцев подхода Open Source. Неопровержимым свидетельством этого является развитие программы Shared Source, в рамках которой Microsoft распространяет исходные тексты CLI (Common Language Infrastructure, Общеязыковая Инфраструктура). В ней представлены исходные тексты всех основных сервисов платформы .NET, включая общую библиотеку классов, виртуальную машину, JIT-компилятор, C# компилятор, а также многое другое.

Предоставляемый исходный код мультиплатформен — он собирается на трех различных платформах: Windows, FreeBSD, MacOS. Для переноса на платформу FreeBSD использовалась фирменная технология Rotor от Microsoft, которая также поставляется в виде исходных текстов. В рамках данной технологии большая часть Windows API (Application Programming Interface — интерфейс прикладного программирования) была портирована на FreeBSD, с использованием ее стандартных внутрисистемных вызовов.

Глава 2

Метаданные

2.1. Краткий экскурс в историю

Перед разработчиками платформы .NET была поставлена весьма нетривиальная задача — обеспечить полную межъязыковую интеграцию в рамках среды. Ранее подобные цели ставились перед разработчиками COM (Component Object Model, Компонентная Объектная Модель), но им не удалось добиться желаемого результата. Формально, конечно, COM предоставляет все необходимые для межъязыковой интеграции сервисы. Свидетельством этого является возможность организации взаимодействия с платформой COM из различных языков, начиная от Ассемблера и заканчивая простейшими языками сценариев вроде VBScript. Однако, с пользовательской точки зрения, взаимодействие было организовано настолько неудобно и запутанно, что его вообще мало кто понимал. Главный просчёт разработчиков COM состоял в том, что основные заботы по межъязыковой интеграции были возложены на рядовых программистов. Именно они должны были сглаживать различия между языками, путём введения унифицированных интерфейсов, информация о которых предоставлялась через библиотеки типов.

Разработчики .NET решили пойти другим путём — они ввели систему метаданных, описывающих не только внешние интерфейсы, предоставляемые приложениями и компонентами, но также их полную внутреннюю структуру. Причем метаданные спроектированы как неотъемлемая часть любого управляемого приложения. Теперь больше не надо создавать дополнительные заголовочные файлы или внешние библиотеки типов. Вся информация предоставляется самими приложениями и хранится совместно с их кодом. Благодаря этому, стало возможным взаимодействие как через внешние, специально предоставленные интерфейсы, так и через прямое обращение к внутренним программным элементам. Теперь можно использовать типы других приложений как свои собственные — применять в качестве базовых, создавать их экземпляры, обращаться к их методам и т. д. Обобщая, можно сказать, что метаданные — это новый виток в эволюции технологий предоставления информации о типах и программах в целом.

Подобная модель взаимодействия существенно упрощает жизнь рядовым пользователям, потому что теперь им не нужно задумываться об описании внешних интерфейсов, которые не так уж и просты для понимания. Многим начинающим программистам совершенно непонятно, для чего, например, нужны заголовочные файлы в C/C++ подобных языках, или еще хуже — зачем необходимо создавать библиотеки типов для COM-компонентов. Теперь рядовые пользователи отстранены от необходимости решения вопроса, метаданные автоматически предоставят необходимую информацию как о внешних, так и о внутренних интерфейсах.

Все современные компиляторы, совместимые с платформой .NET, генерируют из текста программы не только исполняемый код, но и метаданные, описывающие программную модель приложения.

Взаимодействие же приложений осуществляется не на уровне исходных кодов, а на уровне чётко стандартизированных универсальных метаданных. Механизм взаимодействия показан на схеме (рис. 2.1).

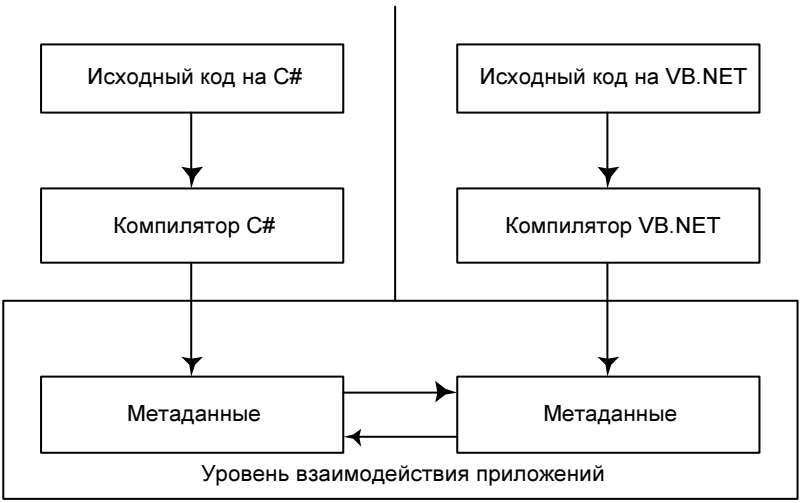


Рис. 2.1. Схема взаимодействия приложений на уровне метаданных

Фактически, метаданные являются унифицированной и чётко стандартизированной технологией описания исходного кода. Они являются выжимкой той, действительно необходимой информации, которая содержится в первозданном исходном коде приложения. Состав информации, содержащейся в метаданных, приведен в табл. 2.1.

Таблица 2.1. *Информация, содержащаяся в метаданных*

Объект	Информация для данного объекта
Сборка	Имя
	Версия
	Региональная информация
	Открытый криптографический ключ
	Экспортируемые типы
	Зависимости сборок
	Атрибуты защиты
Типы	Имя
	Вид доступа
	Иерархия наследования
	Описание полей, методов, свойств, событий, вложенных типов
	Дополнительная информация о коде
Атрибуты	Дополнительная пользовательская информация о типах

Как видно из таблицы, помимо жизненно необходимой информации, метаданные содержат, казалось бы, избыточное описание многих программных элементов. Но именно эта дополнительная информация позволяет наладить эффективное взаимодействие между различными языками. По сути дела, на уровне самой среды исполнения никаких языков и нет, там есть лишь метаданные и IL. Сама среда исполнения даже представления не имеет, на каком языке изначально было написано приложение. Точно так же, как процессор не имеет представления о языке, на котором разрабатывалась программа. Эта информация не нужна для исполнения приложений.

Напрашивается очевидный вопрос: почему при столь явной аналогии между процессором и виртуальной машиной .NET для обычных классических приложений не была организована столь мощная межъязыковая интеграция. Ответ оказывается прост — у таких приложений есть лишь линейный код, который приказывает процессору, что тот должен делать. При этом они (приложения) не предоставляют процессору абсолютно никакой информации о своём внутреннем устройстве. Код идёт в процессор линейным потоком, который лишь иногда меняет своё направление, под влиянием условных или безусловных переходов, но не более того. И лишь в исключительных случаях процессор может прервать исполнение программы и попытаться обратиться к одному из её сервисов. На уровне машинного кода нет ни функций, ни локальных переменных, ни тем более классов, тут лишь есть голый

код и ничего более. Именно поэтому столь сложно организовать действительно гибкое взаимодействие между программами, сам процессор не предоставляет для этого никаких сервисов.

Ситуация в .NET кардинально отличается. Здесь среда исполнения имеет полное представление о внутреннем устройстве программы, которое она получает из метаданных. Она знает, как устроена программа, и может предоставить сервисы для взаимодействия с другими программами. Причем эти сервисы работают полностью в автоматическом режиме, абсолютно прозрачно для программиста.

2.2. Практическое введение в систему метаданных

Для того чтобы разобраться с общей структурой информации, хранящейся в метаданных, сравним код обычного (неуправляемого) компилируемого приложения и код идентичного управляемого приложения, содержащего метаданные. На листинге 2.1 представлен простой пример программы на C++.

Листинг 2.1. Пример исследуемого приложения 1

```
/*
  Листинг 2.1
  File:   Some.cpp
  Author: Дубовцев Алексей
*/
// Подключим основной заголовочный файл Windows API.
#include <windows.h>
// Введём тестовый класс, который будем использовать в программе.
class CMessageBox
{
public:
  // Опишем функцию как статическую для того, чтобы не
  // создавать экземпляр класса.
  static void Show(LPTSTR szMessage)
  {
     // Выведем на экран приветствие.
     MessageBox(NULL,szMessage,NULL,0);
  }
};
```

```
// Точка входа в приложение.
void main()
{
  // Обратимся к статической функции нашего класса.
  CMessageBox::Show("Hello, World!");
}
```

В начале приложения объявляется один класс со статическим методом, затем в основной функции приложения производится обращение к функции класса, которая вызывает API-сервис операционной системы, выводящий на экран приветствие. В результате исполнения приложение выводит на экран диалоговое окно с надписью "Hello, World!". Но нас интересует не результат работы приложения, а его внутреннее устройство. Результат дизассемблирования приложения представлен на листинге 2.2.

Листинг 2.2. Ассемблерный код исследуемого приложения 1

```
.text:00401001      push    ebp
.text:00401001      mov     ebp, esp
.text:00401003      push    402008
.text:00401008      call    loc_401020
.text:0040100D      add     esp, 4
.text:00401010      xor     eax, eax
.text:00401012      pop     ebp
.text:00401013      retn
.text:00401020      loc_401020:
.text:00401020      push    ebp
.text:00401021      mov     ebp, esp
.text:00401023      push    0
.text:00401025      push    0
.text:00401027      mov     eax, [ebp+8]
.text:0040102A      push    eax
.text:0040102B      push    0
.text:0040102D      call    00402000        ;MessageBoxA
.text:00401033      pop     ebp
.text:00401034      retn
```

В результате компиляции получился красивый компактный код, на исполнение которого потребуются считанные такты процессора.

Теперь же рассмотрим аналогичную программу, разработанную с использованием технологии .NET. Хотя она написана на языке C#, её исходный код сильно напоминает предыдущее приложение (листинг 2.3).

Листинг 2.3. Пример исследуемого приложения 2 (технология .NET)

```
/*
  Листинг 2.3
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имён общей библиотеки классов.
using System;
// Подключим пространство имён, отвечающее за предоставление
// пользовательского интерфейса.
using System.Windows.Forms;
// Введём дополнительный экспериментальный класс.
class CMessageBox
{
  // Опишем статическую функцию.
  public static void Show(String str)
  {
    // Запросим для вывода на экран диалоговое окно.
    MessageBox.Show(str);
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Обратимся к описанному ранее классу.
    CMessageBox.Show("Hello, World!");
  }
}
```

Теперь исследуем внутреннее устройство управляемого приложения. Для этого воспользуемся специализированным дизассемблером `ildasm`, входящим

в комплект поставки Framework SDK. После открытия приложения мы увидим следующую картину (рис. 2.2).

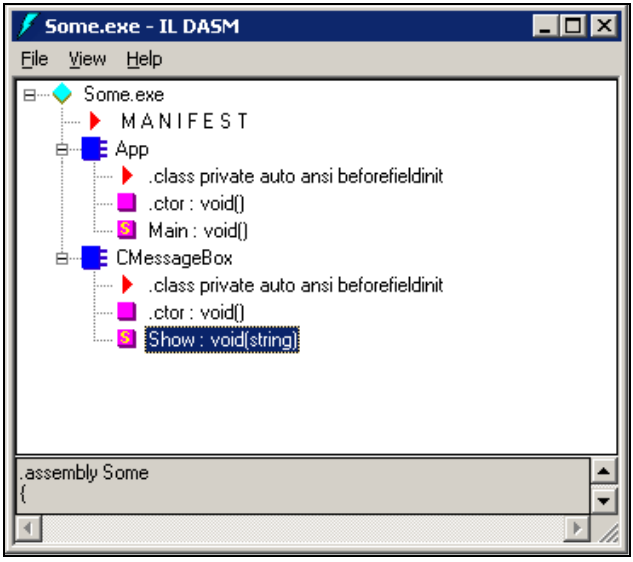


Рис. 2.2. Управляемое приложение 2, дизассемблированное программой `ildasm`

Оказывается, в управляемых приложениях, даже после компиляции, сохраняется информация об их внутреннем устройстве. Дизассемблер `ildasm` позволяет в интерактивном режиме исследовать код управляемых приложений. В главном окне отображается внутреннее устройство программных элементов приложения. Каждый узел дерева соответствует некоторому программному элементу, находящемуся внутри приложения. В табл. 2.2 приведены описания иконок, которые отображаются в узлах дерева. По ним вам будет легче ориентироваться внутри приложения.

Таблица 2.2. *Значение иконок в узлах дерева утилиты* `ildasm`

Иконка	Описание
	Дополнительная информация
	Пространство имён
	Класс
	Интерфейс
	Размерный тип

Таблица 2.2 *(окончание)*

Иконка	Описание
	Перечисление
	Метод
	Статический метод
	Поле
	Статическое поле
	Событие
	Свойство
	Манифест или информация о классе

При щелчке кнопкой мыши на некоторых элементах дерева, `ildasm` отобразит дополнительную информацию о них. Например, если активизировать функцию `Main` класса `App`, то на экране появится диалоговое окно (рис. 2.3), в котором будет отображён её код на языке IL.

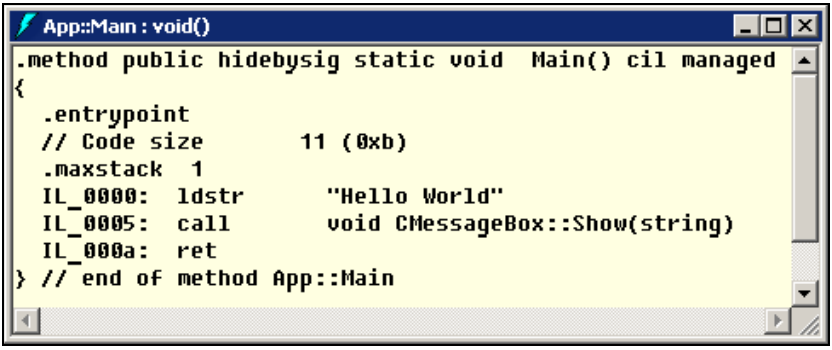


Рис. 2.3. Исходный код отдельной функции управляемого приложения 2 на языке IL

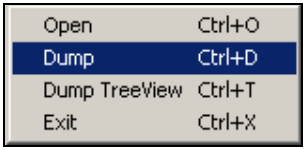


Рис. 2.4. Выбор функции восстановления исходного кода программы на языке IL

Конечно, IL-код, по сравнению с оригинальным исходным кодом приложения, выглядит более громоздким, тем не менее, он гораздо более информативен, чем чистый машинный код.

Интересно, что `ildasm` позволяет исследовать программу не только в интерактивном режиме, но даже создать полный исходный код программы на языке IL. Для этого необходимо выбрать пункт **Dump** в меню **File** (рис. 2.4).

После чего в появившемся диалоговом окне необходимо выбрать настройки согласно рис. 2.5. Здесь вы можете выбрать тип кодировки конечного файла с исходным IL-кодом, а также настроить вид этого файла — наличие дополнительных опций кодов, комментариев, номеров строк и т. п.

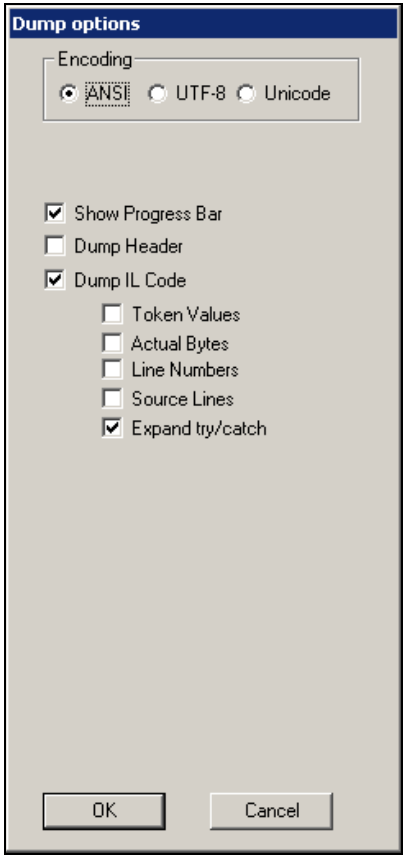


Рис. 2.5. Настройки вида восстанавливаемого IL-кода

После чего останется лишь выбрать файл для сохранения исходного кода приложения и приступить к его непосредственному изучению. В листинге 2.4 приведён исходный код тестового приложения 2 на языке IL.

Листинг 2.4. Дизассемблированный код приложения 2 на языке IL

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 1.1.4322.573
//  Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.


.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         //
.z\V.4..
  .ver 1:0:5000:0
}
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         //
.z\V.4..
  .ver 1:0:5000:0
}
.assembly Some
{
  // --- The following custom attribute is added automatically, do not
uncomment -------
  //  .custom instance void [mscor-
lib]System.Diagnostics.DebuggableAttribute::.ctor(bool,
  //
bool) = ( 01 00 00 01 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Some.exe
// MVID: {14D5E497-7586-4EE5-80E8-0FBB590C2C52}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x06c60000
//
// ============== CLASS STRUCTURE DECLARATION ==================
//
.class private auto ansi beforefieldinit CMessageBox
       extends [mscorlib]System.Object
```

```
{
} // end of class CMessageBox

.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
} // end of class App

// =============================================================

// =============== GLOBAL FIELDS AND METHODS ===================

// =============================================================

// =============== CLASS MEMBERS DECLARATION ===================
//   note that class flags, 'extends' and 'implements' clauses
//          are provided here for information only
.class private auto ansi beforefieldinit CMessageBox
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Show(string str) cil managed
  {
    // Code size       8 (0x8)
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       valuetype [Sys-
tem.Windows.Forms]System.Windows.Forms.DialogResult [Sys-
tem.Windows.Forms]System.Windows.Forms.MessageBox::Show(string)
    IL_0006:  pop
    IL_0007:  ret
  } // end of method CMessageBox::Show
  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
```

```
  } // end of method CMessageBox::.ctor

} // end of class CMessageBox

.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main() cil managed
  {
    .entrypoint
    // Code size       11 (0xb)
    .maxstack  1
    IL_0000:  ldstr      "Hello, World"
    IL_0005:  call       void CMessageBox::Show(string)
    IL_000a:  ret
  } // end of method App::Main
  .method public hidebysig specialname rtspecialname
         instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method App::.ctor

} // end of class App

// =============================================================
//*********** DISASSEMBLY COMPLETE ***********************
// WARNING: Created Win32 resource file C:\asdf.res
```

Пристально изучив листинг, начинаешь понимать устройство метаданных платформы .NET. Даже не имея опыта работы с языком IL, сравнивая этот листинг с ортодоксальным ассемблером, можно понять, насколько больше информации о внутреннем устройстве управляемой программы содержится в метаданных. Наличие этой информации позволяет обеспечить не только межъязыковую интеграцию, но и работу таких технологий, как сериализация, удалённое взаимодействие `.NET Remoting`, механизмы отражения `Reflection` и многих других.

Дизассемблированный IL-код обладает еще одним очень полезным свойством — его можно обратно собрать в полностью рабочее приложение, без опасений что-либо потерять или испортить. Сделать это можно при помощи компилятора `ilasm.exe`, передав ему в качестве первого параметра командной строки имя файла. Такого себе классические приложения позволить не могут. Ортодоксальное дизассемблирование считается процессом весьма и весьма неоднозначным. Даже самые лучшие и современные дизассемблеры не гарантируют, что после декомпиляции можно будет собрать приложение обратно, а тем более заставить его работать.

2.3. В заключение главы

Хотя сами метаданные являются неотъемлемой частью программных технологий .NET, однако они являются абсолютно прозрачными для обычных программистов верхнего уровня. В этом и состоит их цель — обеспечить "бесшумную" работу механизмов верхнего уровня. Хотя в большинстве случаев проблемным программистам не приходится явно работать с метаданными, знание о метаданных позволяет лучше разобраться в механизмах, обеспечивающих функционирование и взаимодействие управляемого кода. К тому же, вы всегда будете производить впечатление осведомлённого специалиста, от взгляда которого ничто не может ускользнуть.

Глава 3

Общая система типов

В главе рассмотрены основные типы, используемые в среде .NET, а также общие правила их применения.

3.1. Важнейшие определения

Общая система типов является фундаментом, на котором построено межъязыковое взаимодействие в среде .NET. Фактически, для всех платформ вводятся единые стандарты, которые обеспечивают межъязыковое взаимодействие на основе CTS (Common Type System, Общая система типов). В рамках этой системы определен набор всех абстракций, поддерживаемых CLR.NET.

CTS способствует решению следующих задач:

- организация межъязыкового взаимодействия;
- обеспечение высокой производительности исполнения кода;
- поддержка механизма защищенности типов (type safety);
- формирование единой объектно-ориентированной среды, используемой всеми языками, которые поддерживают стандарты общеязыковой спецификации (CLS, Common Language Specification).

CLS представляет собой набор правил, определяющих структуру и требования к языкам высокого уровня среды .NET. Этот документ, в основном, предназначен для создателей компиляторов. Он регламентирует многие аспекты разработки приложений, включая: создание новых типов, использование внешних типов, работу со сборками и многое другое. Если разработчик компилятора придерживался стандартов CLS, будет гарантирована совместимость программ, написанных на этом языке, с любыми другими, предназначенными для работы на платформе .NET.

В настоящей главе будут рассмотрены вопросы, относящиеся к введению и использованию новых типов. Хотя материал главы по большей части носит теоретический характер, его знание необходимо при программировании для

платформы .NET. Не понимая процессов, происходящих на каждом шагу создания программного кода, практически невозможно создавать действительно хорошие приложения.

3.2. Классификация типов среды .NET

Прежде чем приступить к детальному изучению CTS среды .NET, рассмотрим общую классификацию типов, используемых в ней. Их иерархическая структура представлена на рис. 3.1.

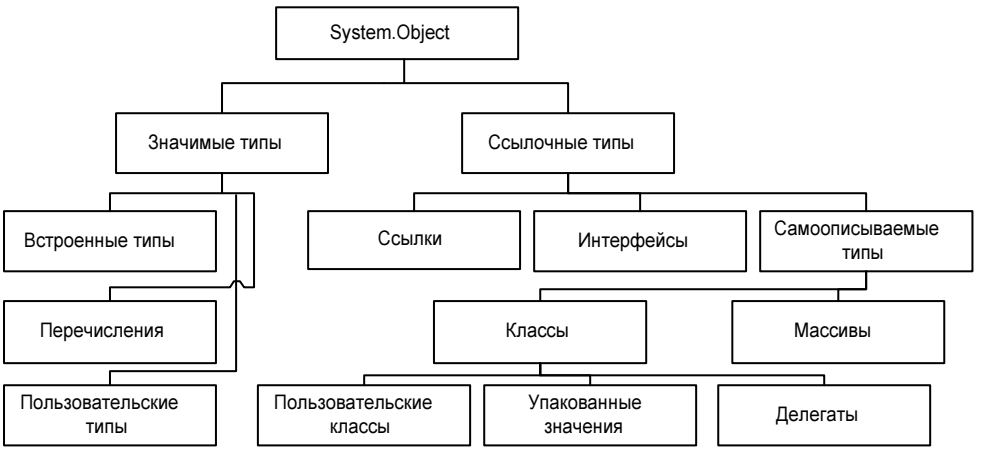


Рис. 3.1. Иерархия типов среды .NET

Корнем иерархии является класс `System.Object`, от которого прямо или косвенно унаследованы все типы среды .NET. Объявляя или используя любой тип, вы можете быть всегда уверены в том, что его самым дальним предком является `System.Object`.

Причины использования однокоренной иерархии типов

Использование однокоренной иерархии позволяет рассматривать любой тип как представитель класса `System.Object`. Такое обобщение очень удобно для создания коллекций (массивов, списков, и т. п.), универсальных алгоритмов, а также во многих других случаях в связи с конструированием обобщенных типов и алгоритмов. Ранее, для решения этой задачи, приходилось использовать преобразование к "бестиповым" указателям (`void*`), что было небезопасной операцией. Поскольку процесс преобразования подобных типов не контролируется со стороны компилятора и весьма вероятно возникновение ошибок, что и показала практика использования таких указателей.

Использование однокоренной иерархии типов (основанной на классе `System.Object`) позволяет провозгласить политику безопасности типов (`type safety`), гарантирующую, что случайная или намеренная подмена типов принципиально невозможна. Во-первых, теперь нигде не используется "бестиповый" подход, во-вторых, все операции над типами, включая их преобразование, строго контролируются самой средой .NET непосредственно во время исполнения программы. Если приложение попытается провести недопустимое преобразование типа, то немедленно будет вызвано исключение `InvalidCastException`. Приведем небольшой пример, демонстрирующий контроль безопасности преобразования типов. Здесь используется некий гипотетический класс `Some`, о котором необходимо сказать лишь то, что он является непосредственным потомком класса `System.Object`.

```
// Хотя в списке предков System. класс System.Object
// явно и не указывается, среда исполнения автоматически заботится
// об этом.
class Some
{
};
// Создадим новый экземпляр нашего класса.
Some s = new Some();
// Мы можем рассматривать его как объект типа Object.
// То есть мы провели обобщение, "усреднив" объект общего
// уровня типа Object.
Object obj = (Object)s;
// Впоследствии можно возвратиться к действительному типу объекта.
Some s2 = (Some)obj;
// Попробовав провести явно неверное преобразование типов,
// привести обычный объект Object к типу Some, мы
// получим исключение InvalidCastException, которое
// выдаст среда исполнения.
Some s3 = (Some)(new Object());
```

Таким образом, система обобществления типов на основе единого предка обеспечивает полный контроль над всеми операциями, проводимыми с типами. При этом проверка корректности операций осуществляется во время исполнения программы, и обойти его практически невозможно. Даже если изменить код программы, сгенерированной компилятором, на ход проверки во время ее исполнения это никоим образом не повлияет.

Два вида типов

Все типы, используемые в среде .NET, делятся на два вида: ссылочные (Reference Types) и размерные (`Value Types`).

Размерные типы. Экземпляры этих типов непосредственно содержат данные, преимущественно это локальные переменные и параметры методов, которые помещаются в стеке потока. Необходимо отметить, что такие экземпляры могут входить в состав более сложных ссылочных типов и, как следствие, располагаться в динамической области памяти. Если требуется воспользоваться ссылкой на подобные данные, приходится использовать операцию `boxing` (обертывание или упаковка).

Ссылочные типы. Для доступа к экземплярам подобных типов используются ссылки, которые указывают на реальные объекты, располагающиеся в динамической памяти приложения. В самих ссылках данные не хранятся, они лишь определяют область памяти, содержащую искомые данные. Причем для одного объекта может быть назначено несколько ссылок.

Далее будет рассмотрено более подробное описание аспектов, связанных с использованием и внутренним устройством типов .NET.

Данные и типы

Прежде чем переходить к детальному изучению конкретных типов (интерфейсов, классов), кратко опишем основные понятия теории типов.

Понятие типа является очень тонким вопросом, достаточно трудным для самостоятельного понимания. Поскольку с понятием типа неразрывно связаны данные, зачастую происходит путаница между ними. Данные являются представлением информационного объекта в памяти и при прямом рассмотрении их не несут никакой смысловой нагрузки, не дают ни малейшего представления о сути объекта. Это просто последовательность бит, хранящихся в памяти (рис. 3.2).

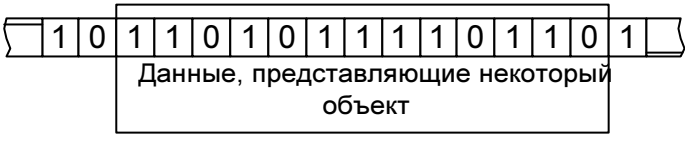


Рис. 3.2. Пример данных, представляющих некоторый объект

Тип представляет собой механизм, позволяющий осмысленно трактовать данные. Именно тип позволяет превратить ничего не значащую двоичную последовательность в число, в строку и в сложный пользовательский объект. Он определяет, каким образом программа будет работать с данными. Рассмотрим простой пример, демонстрирующий внутреннюю сущность типов и данных.

Опишем следующую структуру.

```
struct SomeStruct
{
  char FirstMemeber;
  int SecondMember;
};
```

Введение такого типа в программу никоим образом не повлияет на ее работу. Мы всего лишь определили правило для взаимодействия с определенной областью памяти.

Примечание

Такая структура будет занимать в памяти 5 байт без учета служебных данных. Один байт потребуется на переменную-член `FirstMember` типа `char`, плюс четыре байта на переменную-член `SecondMember` типа `int`.

При введении в программе переменной типа `SomeStruct`, под нее будут выделены ровно пять байт памяти, необходимых для размещения данных структуры. Для примера сравним два следующих определения.

```
SomeStruct val1;
char val2[5];
```

Оказывается, что с точки зрения использования памяти, они ничем не отличаются. В обоих случаях для их хранения будет выделено по пять байт и отличить в памяти структуру `SomeStruct` от массива `val2` не представляется никакой возможности (рис. 3.3).

Рис. 3.3. Память, необходимая для хранения данных различных объектов

Единственное различие между определенными нами объектами заключено в стартовых адресах, которые ответов на волнующий нас вопрос не дают. Для этих целей и предназначены типы, позволяющие получить осмысленный доступ к данным (рис. 3.4).

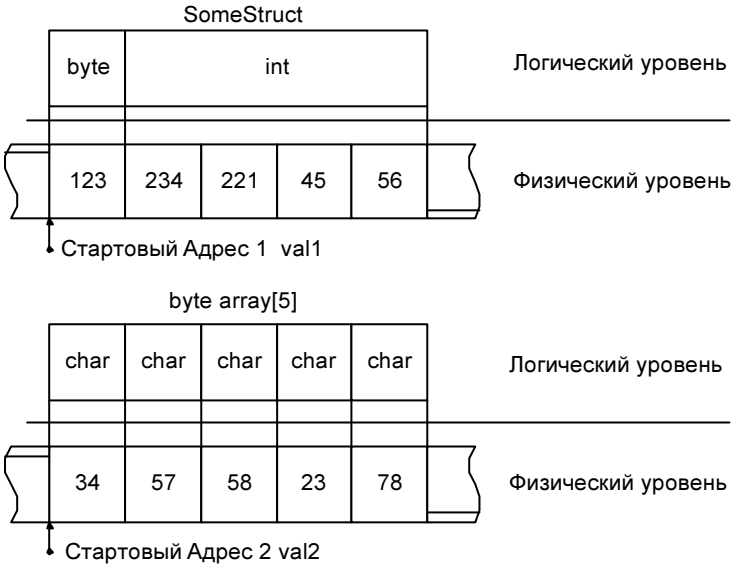


Рис. 3.4. Конкретизация доступа к данным на основе типов

Таким образом, при работе с типами выделяются два уровня обработки информации: логический и физический. Логический уровень определяет логику хранения данных. На этом уровне абстрактные биты превращаются в осмысленные строки, числа, пользовательские объекты, здесь властвуют типы. На физическом уровне производится непосредственная обработка двоичных данных.

В структуру данных также необходимо включить код, который будет обрабатывать данные в соответствии с правилами, определяемыми в типах. Такие правила управляют поведением кода по отношению к данным, являются инструментом, при помощи которого код обрабатывает данные.

3.3. Идентификация типов

Для идентификации конкретных типов в среде .NET применяются обычные символьные имена. Достаточно задать имя типа и можно его использовать в приложении. На самом деле, от пользователей высокоуровневых языков скрыт реальный механизм работы с именами типов. Внутри среды исполнения каждое имя типа должно быть задано в следующем формате.

```
[Имя_Сборки]Имя_Пространства_Имен.Имя_Типа
```

И никак иначе! В него обязательно должно входить имя сборки, в которой располагается тип, имя пространства имен, а также само имя типа. На верхнем уровне программного кода этого не видно. Здесь указывается только

имя типа, иногда дополняемое именем пространства имен. Разберем следующий пример.

```
using System;
class App
{
   public static void Main()
   {
     Console.WriteLine("Hello, World!");
   }
};
```

Тривиальное обращение к типу `Console` будет транслировано компилятором в следующую строку:

```
[mscorlib]System.Console::WriteLine("Hello, World!")
```

В приведенном примере `mscorlib` является именем сборки, в которой определяется тип `Console`.

Примечание

Сборка `mscorlib.dll` располагается в основном каталоге виртуальной машины .NET `%WINDIR%\Microsoft.NET\Framework\vX.X.XXX\` (где вместо `vX.X.XXX` указана версия виртуальной машины). Имя приведенной сборки является сокращением от Multilanguage Standard Common Object Runtime Library (Многоязыковая стандартная общая библиотека времени исполнения).

Кроме того, явным образом указывается пространство имен `System`, в котором располагается применяемый тип. Для идентификации типов контроль над пространствами имен на уровне среды исполнения не требуется. Сама среда исполнения даже допускает содержание точки в имени типа и рассматривает его персональное имя вместе с именем пространства имен как единое целое. Но следует отметить, что в метаданных для каждого типа указывается пространство имен, в котором он описан. Это необходимо для реализации высокоуровневых сервисов.

Обратимся к сугубо технической стороне реализации работы с именами типов. Хотя имена типов по-прежнему задаются в однобайтовой кодировке, среда исполнения не накладывает ограничений на употребляемые символы. Продемонстрируем это на примере, в котором используются имена типов на русском языке (листинг 3.1).

Листинг 3.1. Демонстрация региональных имен типов

```
/*
  Листинг 3.1
  File:   Some.cs
```

```
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class Приложение
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Объявим переменную с именем на русском языке.
    String раз_два_три = "Привет, мир!";
    // Выведем нашу переменную на печать.
    Console.WriteLine(раз_два_три);
  }
}
```

Пример успешно компилируется, и в результате его работы на консоль будет выведена строка:

```
Привет, мир!
```

С одной стороны, возможность использования региональных символов может показаться приятной. Особенно могут прельститься ею начинающие программисты, не сталкивавшиеся с проблемой региональных стандартов. Но, с другой стороны, она грозит большими неприятностями. Представьте себе, каково будет читать такую программу человеку, не имеющему на своем компьютере поддержки русского языка. Она будет выглядеть приблизительно так (листинг 3.2).

Листинг 3.2. Вид листинга на компьютере без поддержки региональных стандартов

```
/*
  T°╕к°¤є 3.2
  File:   Some.cs
  Author: -╟e■Єбїє L√ї·╕ї·
*/
using System;

// +╕ ¤■Є¤■· ·√E╕╕ и°√■Ўї¤°a
class жи°√■Ўї¤°ї
```

```
{
  // T■в·E Є┐■ЇE Є  и°√■Ўї¤°ї
  public static void Main()
  {
    // +e╖aЄ°№  їиї№ї¤¤╟ж °№a, ·■к■и■· ўEЇEЇ°№ ¤E ¤E▒ї№ √же°№■№
    // °  №■є╟вї№
    String e√E_e√E_e√E = "жи°Єїк №°и";
    // L кї їи╣ Є■┐ ■√╣ў╟ї№┐a ¤E▒ї·  їиї№ї¤¤■·
    Console.WriteLine(e√E_e√E_e√E);
  }
}
```

Мало того, что все комментарии и строки не читаются, так еще имена типов и переменных невозможно разобрать.

Среда исполнения не делает различий между символами, которые использовались для задания имен типов. Она их сравнивает побайтно, что позволяет задавать довольно интересные имена. Для демонстрации опишем метод с именем `Test 1"'#$%^*`, в котором есть все: пробелы, кавычки, специальные символы. Не подумайте, что это шутка, метод с таким именем действительно можно определить. Подтверждением тому служит результат декомпиляции на рис. 3.5.

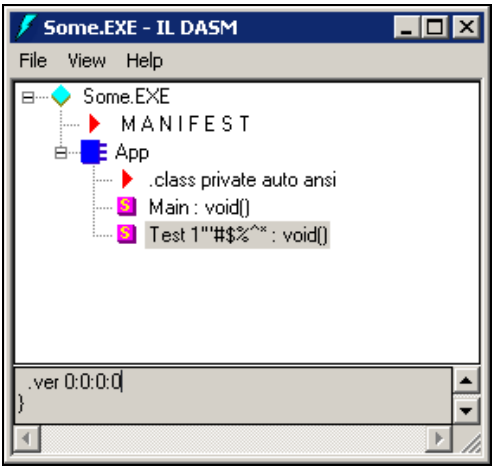


Рис. 3.5. Пример специального имени метода

Интересно взглянуть на листинг программы, приведем его. Из-за ограничений накладываемыми высокоуровневыми компиляторами, программа была разработана на языке IL.

Листинг 3.3. Специальное имя метода

```
/*
  Листинг 3.3
  File:   Some.il
  Author: Дубовцев Алексей
*/
// Объявим ссылку на общую библиотеку типов. Она располагается в сборке
// mscorlib.dll.
.assembly extern mscorlib {}
// Объявим имя нашей сборки.
.assembly Some
{
  // Зададим ее версию.
  .ver 0:0:0:0
}
// Зададим имя основного модуля нашего приложения.
.module Some.exe
// Это основной класс приложения.
.class App
  extends [mscorlib]System.Object
{
  // Точка входа в приложение.
  .method public static void Main() cil managed
  {
    // Модификатор определяет, что данная функция
    // является точкой входа приложения.
    .entrypoint
    // Определим максимальный размер стека для функции.
    .maxstack 1
    // Загрузим в стек ссылку на следующую строку.
    ldstr       "Hello, World!"
    // Вызовем метод Console.WriteLine.
    call        void [mscorlib]System.Console::WriteLine(string)
    // А теперь вызовем наш метод.
    call        void App::'Test 1\'\"#$%^*'()
    // Возвращаемся из функции.
    ret
```

```
  }
  // Объявим метод с нашим именем.
  .method public static void 'Test 1\'\"#$%^*'() cil managed
  {
    // Максимальный размер стека функции.
    .maxstack 1
    // Загружаем ссылку на строку.
    ldstr       "Hello, World from Test!"
    // Вызываем функцию, выводящую сообщение на консоль.
    call        void [mscorlib]System.Console::WriteLine(string)
    // Возвращаемся из функции
    ret
  }
}
```

Единственным ограничением подобных идентификаторов является невозможность их использования в языках высокого уровня. Но можно давать типам подобные имена вполне намеренно, чтобы ограничить доступ к ним. К примеру, любая программа, написанная на Jscript .NET, содержит объект `'JScript 0'`. В его имени есть пробел, вследствие чего он становится недоступным из языков высокого уровня. Необходимо отметить, что подобная защита не является гарантированной. Ее можно обойти при помощи технологии отражения, динамически связавшись с типом.

3.4. Размерные типы

В разделе будут рассмотрены размерные типы среды .NET. Эта ветвь иерархии типов является наиболее простой и включает в себя три подкатегории типов: встроенные, перечисления и пользовательские.

Встроенные типы

В .NET определена группа стандартных элементарных типов, поддерживаемых компиляторами и самой виртуальной машиной. Они используются для оперирования элементарными данными — числами, строками, объектами. Объединяя встроенные типы в классы или структуры, можно создавать более сложные пользовательские типы.

Все типы этой категории представлены соответствующими классами в основном пространстве имен `(System)` общей библиотеки классов. Для удобства работы программистов, компиляторы вводят собственные идентификаторы типов, которые впоследствии транслируются в обращения к настоящим типам.

В таблице 3.1 приведено соответствие собственных имен типов среды исполнения и ключевых слов языков, которые позволяют их использовать.

Таблица 3.1. *Соответствие имен встроенных типов .NET типам языков высокого уровня*

Категория	Имя класса	Описание	Типы данных в различных языках среды .NET			
			VB	C#	MC++	IL
Целые	Byte	8-битное беззнаковое целое	Byte	byte	Char	unsigned int8
	Sbyte[1]	8-битное знаковое целое	Sbyte[2]	sbyte	signed char	int8
	Int16	16-битное знаковое целое	Short	short	Short	int16
	Int32	32-битное знаковое целое	Integer	int	int или long	int32
	Int64	64-битное знаковое целое	Long	long	__int64	int64
	UInt16	16-битное беззнаковое целое	UInt16[2]	ushort	Unsigned short	unsigned int16
	UInt32	32-битное беззнаковое целое	UInt32[2]	uint	Unsigned int или unsigned long	unsigned int32
	UInt64	64-битное беззнаковое целое	UInt64[2]	ulong	Uinsigned __int64	unsigned int64
Вещественные	Single	Вещественное с плавающей точкой обычной точности (32 бита)	Single	float	Float	float32
	Double	Вещественное с плавающей точкой удвоенной точности (64 бита)	Double	double	Double	float64
Логические	Boolean	Булево число (true или false)	Boolean	bool	Bool	bool

Таблица 3.1 *(окончание)*

Категория	Имя класса	Описание	Типы данных в различных языках среды .NET			
			VB	**C#**	**MC++**	**IL**
Другие	`Char`	`Unicode`-символ (16 бит)	`Char`	`char`	`Wchar_t`	`Char`
	`Decimal`	96-битное десятичное значение	`Decimal`	`decimal`	`Decimal`	`valuetype [mscorlib]System.Decimal (*2)`
	`IntPtr`	Знаковое целое, размер которого зависит от разрядности платформы среды исполнения	`IntPtr`[2]	`IntPtr`[2]	`IntPtr`[2]	`native int`
	`UIntPtr`	Беззнаковое целое, размер которого зависит от разрядности платформы среды исполнения[1]	`UIntPtr`[2]	`UIntPtr`[2]	`UIntPtr`[2]	`native unsigned int`
Классы	`Object`	Прародитель всех объектов	`Object`	`object`	`Object*`	`object`
	`String`	Unicode-строка фиксированной длины	`String`	`string`	`String*`	`string`

[1] Тип несовместим с общеязыковой спецификацией CLS.

[2] Не является встроенным типом для этого языка высокого уровня, в нем нет специального ключевого слова, позволяющего использовать такой тип. Для того чтобы им воспользоваться, необходимо будет указывать полное имя его класса.

Использование ключевых слов языков программирования, а не имен классов общей библиотеки среды .NET, может ввести начинающих программистов в заблуждение. Например, применив тип `int`, можно предположить, что такой тип существует, хотя это всего лишь псевдоним для класса `Int32`. При компиляции программы, обращение `int` будет транслировано в обращение `System.Int32`. С одной стороны, это удобная возможность скрыть низкоуровневую действительность работы типов внутри платформы. Но, с другой

стороны, она уводит программистов от понимания процессов, происходящих в системе. Когда обычный программист сталкивается с классами, представляющими настоящие встроенные типы (к примеру, `Int32` или `String`), он нередко приходит в замешательство, потому что не может понять различия между ними (и которого попросту нет). К тому же, использование встроенных типов компилятора укрепляет языковой барьер — непосвященному программисту может показаться, что различные языки, совместимые с .NET, используют различные наборы элементарных типов.

Как уже отмечалось, каждый элементарный тип представлен соответствующим типом в общей библиотеке. На первый взгляд может показаться, что подобная архитектура может отрицательно сказаться на производительности приложений. Однако в реальной жизни этого не происходит, поскольку такие классы имеют непосредственную поддержку со стороны виртуальной машины .NET. А та, в свою очередь, знает все о каждом из этих классов и контролирует все операции, производимые над ними.

Пользовательские типы

Любой тип, прародителем которого является класс `System.ValueType`, является размерным. Таким образом, если унаследовать от него любой свой класс, он автоматически станет базовым. Но напрямую такую операцию выполнить не удастся, компиляторы высокоуровневых языков запрещают явное наследование от класса `System.ValueType`, мотивируя отказ тем, что он является служебным внутрисистемным классом. Для того чтобы обойти ограничение, придется воспользоваться специальными ключевыми словами, приведенными в табл. 3.2.

Таблица 3.2. *Ключевые слова для определения размерных типов*

Язык	C#	VB	MC++	JScript
Способ определения	`Struct`	`Structure` `End Structure`	`__value struct` или `__value class`	*1

*1 — программируя на языке Jscript, можно лишь использовать структуры, но нельзя определять собственные.

Если критически подходить к вопросу, то со стороны разработчиков платформы .NET было не совсем удачным решением предложить для определения размерных типов ключевое слово `Struct`. Оно никоим образом не отражает сущности размерных типов и может ввести неподготовленного программиста в заблуждение. На самом деле, среда исполнения не поддерживает структур, она работает только с классами. Структуры — это те же классы, но их прародителем обязательно будет тип `System.ValueType`.

Для примера приведем результат трансляции описания простой структуры на языке IL.

Исходный код.

```
struct SomeStruct
{
}
```

Компилированный код.

```
.class sealed SomeStruct
  extends [mscorlib]System.ValueType
{
}
```

Из примера видно, что это обычный класс. Правда, стоит отметить, что введением понятия структуры, разработчики платформы четко очертили различия между размерными и ссылочными типами для программистов на языках высокого уровня.

Внутренние механизмы работы типов

Напомним, что при оперировании с размерными типами используется их прямое представление в памяти, а для ссылочных типов применяются ссылки на область памяти, где располагаются данные объекта. То есть переменная размерного типа — это сами данные, а переменная ссылочного типа — указатель на данные, но не сами данные.

Поясним принципы работы размерных и ссылочных типов на примере (листинг 3.4). Программа содержит две функции `ValueDemo` и `RefDemo`, которые демонстрируют работу с размерными и ссылочными типами, соответственно. Алгоритм работы функций практически идентичен: обе они создают по две переменные, присваивают им одинаковые значения, а затем изменяют значение одной из них. В случае размерных типов, изменение одной переменной никоим образом не скажется на значении другой, поскольку при работе с подобными переменными используются сами данные. А в случае со ссылочными типами, изменение данных через одну ссылку, безусловно, отразится на значении, получаемом через другую ссылку.

Листинг 3.4. Демонстрация работы с размерными и ссылочными видами типов

```
/*
  Листинг 3.4
  File:   Some.cs
  Author: Дубовцев Алексей
*/
```

```
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем тестовый ссылочный тип.
class SomeClass
{
  // У него только один член в виде целочисленного
  // поля, имеющего по умолчанию значение 0.
  public int x = 0;
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main ()
  {
    // Эксперименты над размерными типами.
    ValueDemo();
    // Эксперименты над ссылочными типами.
    RefDemo();
  }
  // Функция демонстрирует работу с размерными типами.
  public static void ValueDemo()
  {
    // Введем две переменные размерного типа.
    int v1;
    int v2;
    // Первой переменной присваиваем значение 3.
    v1 = 3;
    // Второй переменной присваиваем значение первой,
    // при этом данные из первой переменной копируются во
    // вторую. Таким образом, вторая переменная теперь никак
    // не связана с первой.
    v2 = v1;
    // Для того чтобы доказать отсутствие связи между
    // переменными, изменим значение первой из них.
    v1 = 4;
    // И выведем на консоль значения обеих переменных,
    // чтобы убедиться, что они никак не связаны
    // между собой.
```

```
    Console.WriteLine("v1 = {0}\nv2 = {1}\n",v1,v2);
  }
  // Эта функции демонстрирует работу со ссылочными типами.
  public static void RefDemo()
  {
    // Определим две переменные ссылочного типа, которым
    // является наш класс.
    SomeClass r1;
    SomeClass r2;
    // Прошу обратить внимание, что при объявлении переменных
    // ссылочного типа создаются лишь пустые ссылки,
    // которые указывают в никуда.
    // Для того чтобы они заработали, необходимо еще создать
    // сам объект (данные) и присоединить его к ссылке.
    // Создадим экземпляр нашего класса и свяжем его с
    // первой переменной.
    r1 = new SomeClass();
    // Присвоим второй переменной значение первой, при этом будет
    // скопирована ссылка, то есть адрес той области
    // памяти, в которой хранится ранее созданный экземпляр
    // нашего класса.
    r2 = r1;
    // Установим значение единственного поля нашего экземпляра
    // через первую ссылку.
    // Поскольку экземпляр (данные) у нас один, это
    // изменение будет видно так же, как если мы осуществим доступ к нему
    // через вторую ссылку.
    //
    r1.x = 3;
    Console.WriteLine("r1.x = {0}\nr2.x = {1}",r1.x,r2.x);
  }
};
```

В результате работы программы на консоль будут выведены следующие строки:

```
v1 = 4
v2 = 3
r1.x = 3
r2.x = 3
```

Из распечатки сообщений программы следует, что переменные v1 и v2 независимы друг от друга, а ссылка r2 строго связана со ссылкой r1. Смысл заключается в том, что переменные размерных типов представляют сами данные, а ссылочные лишь указывают на них (рис. 3.6).

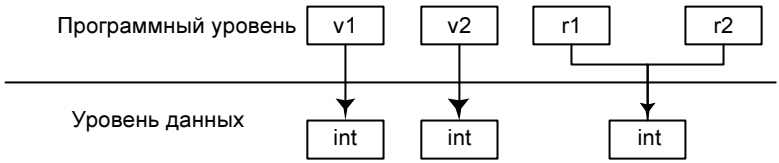


Рис. 3.6. Размерные (v1, v2) и ссылочные (r1, r2) переменные в программе 3.6

Таким образом, несколько ссылок могут указывать на одни и те же данные, а переменные размерных типов всегда имеют каждый свои собственные данные.

При работе с размерными типами зачастую приходится полностью копировать их данные, что не способствует увеличению скорости работы. В связи с этим на размерные типы наложено существенное ограничение — они не могут быть прародителями других классов. Это правило предотвращает разрастание пользовательских типов и предупреждает возникновение проблемы производительности при работе с ними.

Упаковка и распаковка

Экземпляры размерных типов представляют собой данные в чистом виде, что позволяет работать с ними напрямую, не используя указателей. Но иногда требуется получить ссылку, указывающую на размерный тип. К примеру, может понадобиться передать его по значению как один из параметров метода или же поместить в коллекции, которые позволяют работать только со ссылками. Для этого в среду .NET введены две операции: упаковка (`boxing`) и распаковка (`unboxing`). Первая позволяет получить ссылку, указывающую на размерный тип, вторая служит для противоположных целей. Операция упаковывания создает копию объекта размерного типа и ссылку необходимого типа, указывающую на скопированные данные. Доступ к данным размерного типа по ссылке производится по следующему алгоритму: сначала данные объекта копируются, и с копией связывается ссылка определенного типа. Эта операция называется упаковкой (рис. 3.7). Затем, после использования объекта, измененные данные копируются на место прежних, а ссылка уничтожается. Эта операция, в свою очередь, называется распаковкой.

Сверху на рисунке изображен исходный объект размерного типа. При помощи операции упаковки мы создаем из него два различных ссылочных объекта — клоны искомого. Особенно интересно то, что при этом мы можем указать произвольный тип ссылки, которая будет указывать на скопи-

рованные данные. Таким образом, мы получаем доступ к данным размерного объекта через ссылку, но при этом не стоит забывать, что это лишь копия объекта, а не он сам. Фактически, внося изменения в упакованный объект через ссылку, вы никоим образом не можете подействовать на исходный объект. Для этого придется воспользоваться операцией распаковки, она позволяет преобразовать ссылочный объект к размерному типу. Суть операции сводится к разрыву связи между данными и ссылкой, с последующим удалением второй.

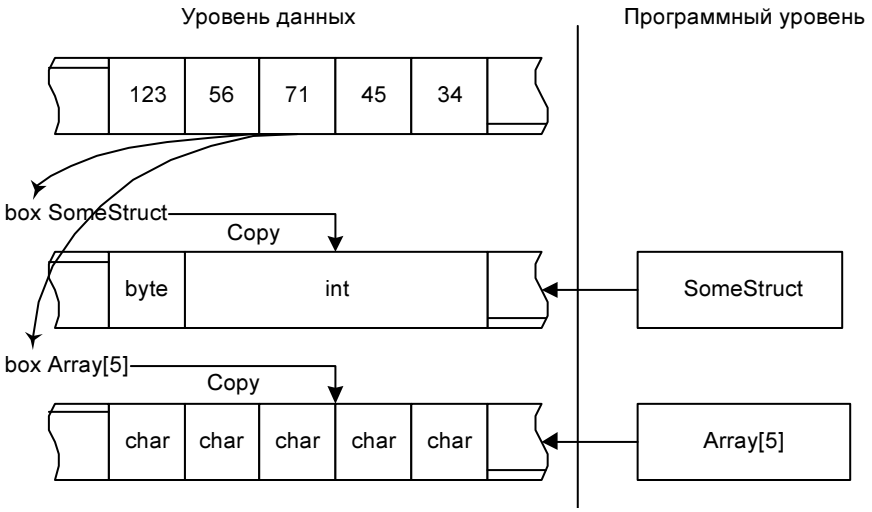


Рис. 3.7. Схема работы операции упаковки

Вверху рисунка изображен объект размерного типа. При помощи операции обертывания мы, по сути дела, создаем новый ссылочный объект — копию искомого. Самое интересное, что при этом можно указать произвольный тип ссылки, которая будет указывать на скопированные данные. Таким образом, мы получаем доступ к данным размерного объекта через ссылку, но при этом не стоит забывать, что этот объект является копией искомого и фактически не связан с ним. Внося изменения в упакованный объект размерного типа, вы не можете подействовать на искомый объект. Для этих целей служит операция распаковки, она позволяет преобразовать ссылочный объект к размерному типу. Суть операции заключается в разрыве связи между ссылкой и данными объекта (рис. 3.8).

После проведения операции распаковки возвращается чистый объект размерного типа, который можно использовать по своему усмотрению. В итоге, схема работы выглядит следующим образом: упаковал, использовал, распаковал обратно. Такие операции производятся всякий раз при преобразовании типов, передаче аргументов по ссылке, при возвращении значений функций.

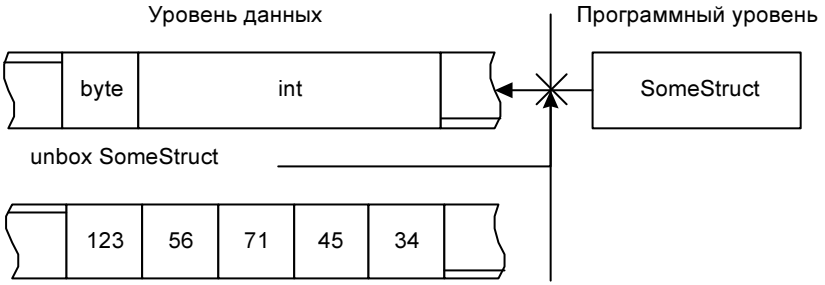


Рис. 3.8. Операция распаковки

Операции упаковки и распаковки составляют неотъемлемую часть работы в программной среде .NET. В большинстве высокоуровневых языков она полностью автоматизирована и является прозрачной для программиста, но в некоторых языках осуществлять операции придется вручную. К примеру, в языке MC++ программистам предоставлено специальное ключевое слово `__box`, которое отвечает за операцию упаковки. Операция распаковки производится при помощи специальной конструкции преобразования типов — `dynamic_cast<__box` ИмяТипа>(Ссылочный_Тип).

Продемонстрируем работу рассмотренных операций на примере (листинг 3.5). В программе специально для этого введен дополнительный размерный тип — структура `VStruct`. Экземпляр такой структуры должен быть передан в функцию `SomeFunction`, которая требует передачи по ссылке — это необходимо для изменения значения одного из полей объекта. Перед передачей размерного типа функции, производится его упаковка. После окончания работы функции, объект распаковывается и помещается на место использованного.

Листинг 3.5. Ручная распаковка и упаковка размерных типов

```
/*
  Листинг 3.5
  File:   Some.cpp
  Author: Дубовцев Алексей
*/
// Подключим основную сборку общей библиотеки классов.
#using <mscorlib.dll>
// Подключим основное пространство имен общей библиотеки классов.
using namespace System;
// Введем пользовательский размерный тип.
```

```
__value struct VStruct
{
  // Данное поле мы будем изменять через ссылку.
  int i;
};
// Функция, которая требует принимать размерный тип по ссылке.
void SomeFunction(VStruct* v)
{
  // Изменим значение поля объекта размерного типа.
  v->i = 33;
}
// Точка входа в приложение.
int main()
{
  // Объявим экземпляр размерного типа.
  VStruct value={10};
  // Выведем на консоль значение его единственного поля.
  Console::WriteLine(S"First Value = {0}",__box(value.i));
  // Произведем операцию упаковки объекта размерного типа,
  // в результате чего мы получим ссылку на новый упакованный
  // объект. Обратите внимание, что тип ссылки мы задаем самостоятельно.
  __box VStruct* pBoxedV = __box(value);
  // Передадим ссылку на новый упакованный объект.
  SomeFunction(pBoxedV);
  // Выведем на консоль значение поля упакованного объекта.
  Console::WriteLine(S"Boxed value = {0}",__box(pBoxedV->i));
  // Выведем на консоль значение поля искомого объекта.
  // Обратите внимание на то, что распаковка еще не проводилась.
  Console::WriteLine(S"Source value before unboxing = {0}",
__box(value.i));
  // Произведем распаковку объекта.
  value = *dynamic_cast<__box VStruct*>(pBoxedV);
  // Выведем на консоль значение поля искомого объекта после распаковки.
  Console::WriteLine(S"Source value after unboxing = {0}",
__box(value.i));
  return 0;
}
```

В результате работы программы на консоль будут выведены следующие строки:

```
First Value = 10
Boxed value = 33
Source value before unboxing = 10
Source value after unboxing = 33
```

Таким образом, пока не будет произведена распаковка объекта, его данные обновлены не будут.

Пример имеет еще один поучительный момент. Обратите внимание на способ передачи аргументов функции `WriteLine`. Она в качестве аргумента требует ссылку на объект `Object*`, и для того чтобы предоставить ссылку на поле размерного типа `int`, используемого нами в экспериментах, пришлось применить операцию упаковки.

3.5. Ссылочные типы

Ссылочные типы — наиболее распространенный вид типов в среде .NET. К ним относятся классы, делегаты, массивы, интерфейсы и ссылки. Все они подробно рассмотрены в настоящем разделе.

Классы и их применение

Классам в среде .NET отведена главенствующая роль. Связано это с тем, что платформа .NET изначально проектировалась как объектно-ориентированная среда, базирующаяся на классах. Среда .NET пронизана классами сверху донизу; большинство технологий, используемых в ней, построено на классах, начиная с общей библиотеки классов и заканчивая механизмами поддержки исключений.

Класс — это один из способов описания сгруппированных данных и набора операций для работы с ними. При этом подразумевается, что сами данные скрыты внутри объекта класса, а для работы с ними используются внешние механизмы, предоставляемые классом. Таким образом, соблюдается концепция черного ящика (инкапсуляция), позволяющая работать с данными, не имея ни малейшего представления об их внутреннем устройстве. Сами классы являются типами, то есть правилами, используемыми при работе с данными. Объявляя класс, вы вводите правило для работы с его полями, в которых содержатся данные. Рассмотрим на примере.

```
// Опишем класс (тип) — правило для работы с данными.
class SomeClass
{
  // Это данные класса, его поля.
```

```
  int         m_Value;
  // Одно из правил, позволяющее работать
  // с данными, представленными этим классом.
  void        DoIt()
  {
    // Взаимодействуем с данными.
    Console.WriteLine("Value of m_Value is {0}",m_Value);
  }
};
```

Условно все классы можно разделить на два уровня — уровень данных (члены) и уровень правил для работы с этими данными (методы).

Для того чтобы работать с классом, прежде всего необходимо создать его экземпляр. Сделать это можно следующим образом.

```
...
{
  // Создаем экземпляр класса в памяти.
  SomeClass sc = new SomeClass();
  // Изменяем поле (данные) только что созданного экземпляра.
  sc.m_Value = 10;
  // Вызываем метод класса применительно к данному экземпляру,
  // то есть обращаемся к внутриклассовому правилу, предназначенному
  // для работы с его данными.
  sc.DoIt();
}
...
```

Процесс создания экземпляра класса представляет собой не что иное, как выделение области памяти для размещения полей класса. После чего можно работать с экземпляром класса при помощи правил, определенных его типом. К примеру, получить доступ к одному из полей класса или вызвать один из его методов. При этом среда исполнения будет действовать согласно инструкциям, заложенным нами при описании типа этого класса.

Рассмотрим внутренний механизм работы классов. Начнем с создания экземпляра объекта. Сначала среда исполнения резервирует область памяти, необходимую для хранения полей объекта, затем создает ссылку, указывающую на данный участок памяти. Эта ссылка возвращается программе, запросившей создание объекта (рис. 3.9).

Ссылка является механизмом доступа к объекту. Она хранит в себе указатель на область памяти, в которой расположены данные созданного экземпляра, также в ней представлена информация об их типе. Таким образом,

известно, какой тип необходимо использовать для обработки данных. Все операции, производимые над экземпляром класса, можно разделить на встроенные и пользовательские. Логика встроенных операций однозначна, она заложена в ядро среды исполнения и не может быть изменена программистом. Примером такой операции служит обращение к полю объекта.

```
sc.m_Value = 10;
```

Встретив подобную операцию, среда исполнения обратится к типу ссылки sc, вычислит относительное положение данных `m_Value` внутри типа, далее используя адрес объекта, на который указывает ссылка sc, изменит значение необходимого поля (рис. 3.10).

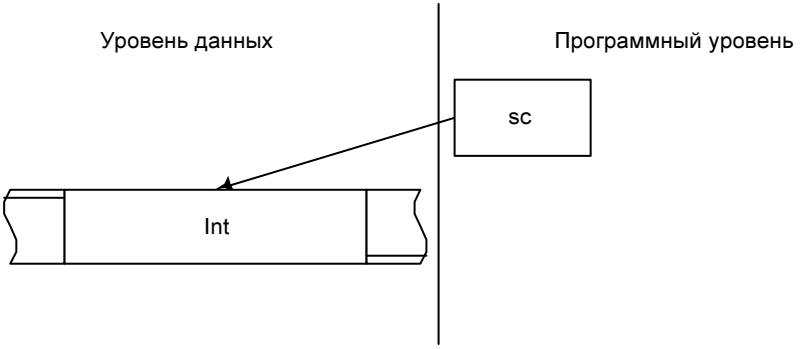


Рис. 3.9. Внутреннее устройство классов

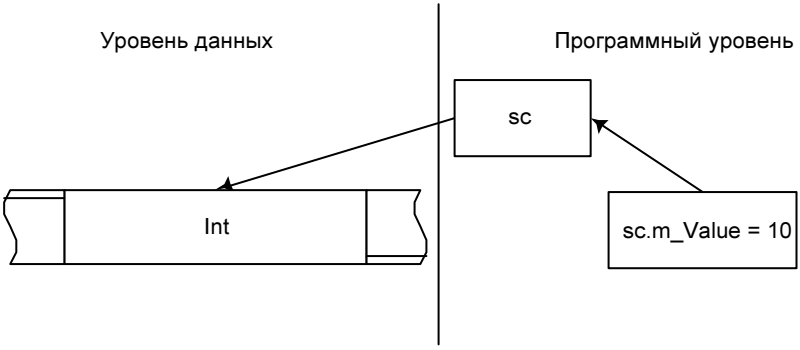


Рис. 3.10. Структура встроенной операции класса

Но что делать, если необходимо ввести собственные операции для работы с объектом. Для этого предусмотрена возможность введения пользовательских операций над объектами. В нашем случае такой операцией является метод `DoIt()`, общая структура которой представлена на рис. 3.11.

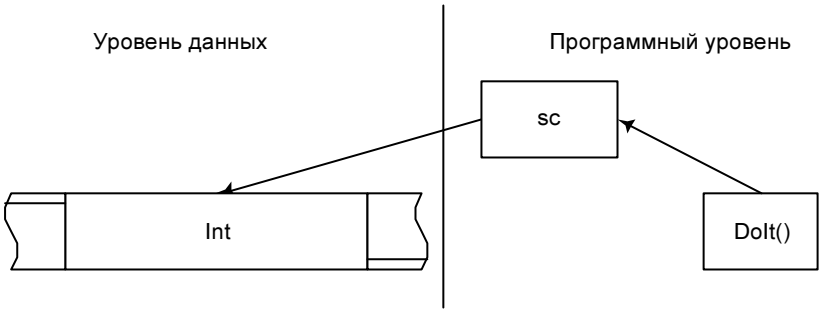


Рис. 3.11. Структура пользовательской операции класса

Механизм обращения к пользовательской операции достаточно прост. Вызов метода класса производится через ссылку `sc`. Компилятор, обнаружив подобный вызов, обращается к методу `DoIt` неявным образом, передавая ему ссылку на объект (`sc`). Фактически, методы классов не отличаются от обычных функций. Они лишь принимают дополнительный скрытый параметр, через который производится доступ к данным экземпляра класса. Этот параметр обладает идентификатором `this`, и его можно использовать из любого метода. Большинство программистов даже не подозревают о существовании такого параметра, поскольку компилятор автоматически подставляет его в нужных местах абсолютно прозрачно для пользователя языка.

Рассмотрим пример некоторого гипотетического метода.

```
// Метод принимает один скрытый параметр,
// отсутствующий в его прототипе.
void SomeMethod()
{
  // Установим значения полей класса.
  m_iMember = 5;
  m_strMember = "Hello, World!";
}
```

Результат после трансляции кода.

```
// Метод принимает один скрытый параметр,
// отсутствующий в его прототипе.
void SomeMethod()
{
  // Установим значения полей класса.
  this.m_iMember = 5;
  this.m_strMember = "Hello, World!";
}
```

Таким образом, пользовательская объектная модель является чисто искусственной и может быть реализована без специальных средств.

После знакомства с внутренним устройством классов, обратимся к их практическому использованию. Рассмотрим общие свойства классов. При описании любого класса можно использовать один или несколько спецификаторов, определяющих его поведение (табл. 3.3).

Таблица 3.3. *Спецификаторы, используемые при определении классов*

Спецификатор	Описание
`sealed`	Наследование от класса запрещено, то есть он не может являться базовым для других классов
`implements`	Означает, что класс реализует указанный интерфейс
`abstract`	Указывает на абстрактность класса, вследствие чего невозможно создание его экземпляров. Подобные классы могут быть полезны только как базовые для других
`inherits`	Делает возможным использование такого класса для работы с экземплярами, имеющими тип одного из его предков
`exported` или `not exported`	Указывает, будет ли класс виден за пределами сборки. Спецификатор применим только для классов верхнего уровня, то есть тех, которые не являются вложенными

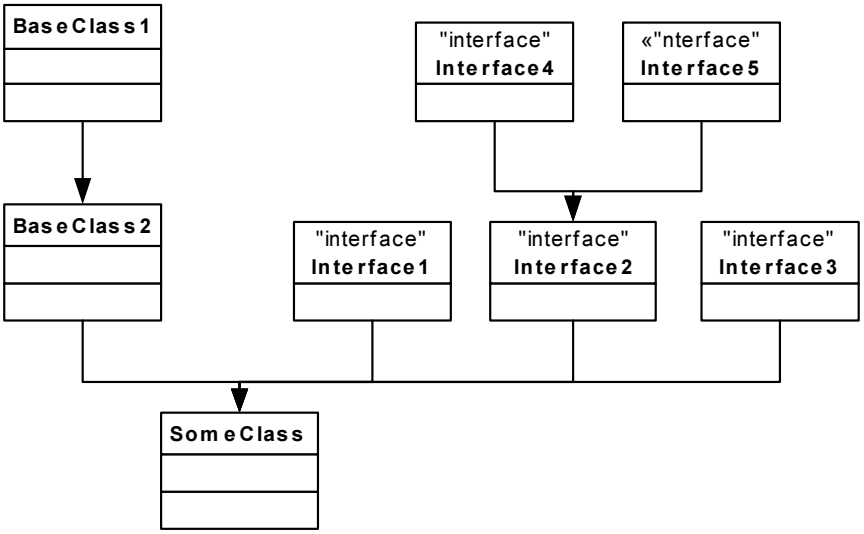


Рис. 3.12. Общая схема наследования классов в среде .NET

Названия спецификаторов в таблице приведены для внутреннего уровня среды исполнения. Способ их задания и названия для различных языков могут кардинально различаться. Некоторые языки вообще могут не поддерживать определение отдельных спецификаторов. За более подробной информацией необходимо обращаться к документации конкретного компилятора.

Механизмы наследования в среде .NET несколько отличаются от общепринятых в объектно-ориентированных средах. Для классов разрешено только линейное наследование и недопустимо множественное. Наряду с этим, существует возможность множественного наследования интерфейсов. Но вследствие того, что интерфейсы не содержат реализации своих членов, подобное наследование является вырожденным.

Общая схема наследования классов, используемая в среде .NET, представлена на рис. 3.12.

Таким образом, мы имеем линейную ветку наследования классов и иерархическую ветвь наследования интерфейсов.

Делегаты

Делегаты используются в среде .NET для создания ссылок на функции и методы. При этом важно понимать, что сами делегаты являются всего лишь типами, а их экземплярами являются ссылки на методы. Объявление делегата в программе, к примеру, таким образом

```
delegate void MyDelegate(int i);
```

еще не вводит ссылку на функцию. Мы лишь описываем тип. Для создания ссылки на функцию при помощи делегата, необходимо создать его экземпляр. Сделать это можно следующим образом:

```
new MyDelegate(DelegateFunction);
```

В результате проведения операции будет создана ссылка на функцию `DelegateFunction`

До появления среды .NET, делегатов не существовало, и приходилось использовать обычные указатели на функции. Работа с ними была крайне неудобна, а спектр их возможностей был ограничен. Указатели на функции могли работать лишь с обычными функциями. Они не поддерживали работу с методами классов и не могли одновременно ссылаться на несколько функций. Делегаты .NET лишены этих недостатков, они предоставляют все перечисленные сервисы (рис. 3.13).

Для работы с делегатами предназначен класс `System.Delegate` и все делегаты в среде .NET являются его потомками. Этот класс предоставляет сервисы для комбинирования делегатов, вызова присоединенных функций, а также для асинхронной работы с ними.

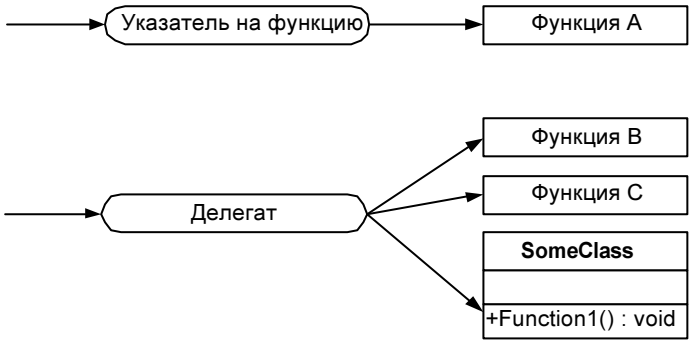


Рис. 3.13. Сравнение делегатов и классических указателей на функции

В большинство языков высокого уровня введены автоматизированные средства работы с таким классом. К примеру, для того чтобы скомбинировать несколько делегатов, вы можете воспользоваться оператором `+=`, вместо прямого обращения к методу `Combine` класса `System.Delegate`. Второй способ также правомерен. Более подробно о делегатах мы поговорим в *главе 8*.

Массивы

Массивы в .NET также претерпели существенное изменение, по сравнению с классическими средами. Теперь они имеют объектно-ориентированную природу. Любой массив, используемый в среде .NET, так или иначе, является потомком класса `System.Array`, который обеспечивает поддержку всех операций, производимых с массивами. Казалось бы, такое усложнение избыточно, и может привести к существенному снижению производительности. Однако взамен мы получаем несколько неоспоримых преимуществ, которые оправдывают такой подход. К ним относятся:

- оптимизация распределения элементов массива в памяти;
- полный контроль правомерности всех операций над массивами;
- стандартный набор базовых алгоритмов для работы с массивами.

Рассмотрим более подробно новые механизмы работы с массивами.

Оптимизация распределения элементов массива в памяти

Внутренняя организация массивов в среде .NET существенно отличается от классического подхода. Теперь элементы массива могут располагаться не только последовательно. Среда исполнения, из соображений производительности, вправе изменять положение отдельных элементов, при этом целостность самого массива не нарушается. Достигается это за счет отказа от использования адресной арифметики для обращения к элементам массива и введения новой технологии двойной индексации. Сначала, при помощи

индекса, вычисляется ссылка, и на ее основе определяется искомый элемент массива (рис. 3.14).

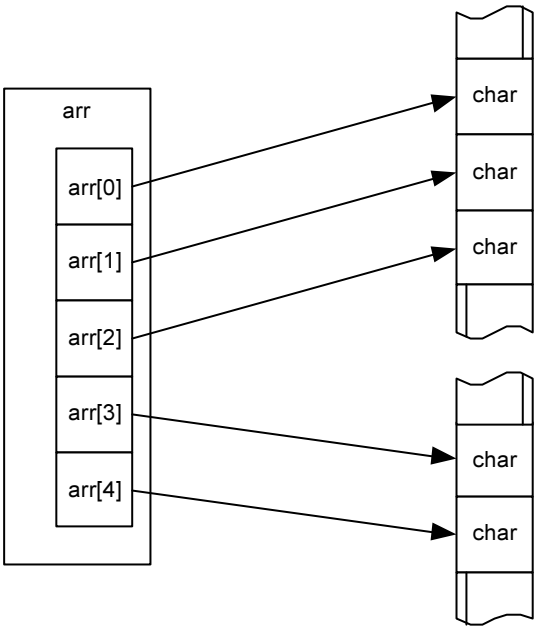


Рис. 3.14. Технология двойной индексации в массивах

Процесс индексации происходит автоматически, абсолютно прозрачно для программиста. Такая технология позволяет оптимизировать работу с памятью при использовании массивов и повышает производительность управляемых приложений.

Контроль правомерности операций над массивами

В ортодоксальных компилируемых системах массивы являются "ахиллесовой пятой". Большое количество ошибок происходит по вине неправильной работы с массивами. Основной проблемой является выход за границы массива. Такие факты ранее не контролировались вообще, предполагалось, что программист самостоятельно обязан заниматься этим вопросом. В результате чего, в большинстве случаев правильность индекса при работе с массивами не проверяется. Это достаточно опасно, поскольку, если индекс превысит размер массива, будет произведено чтение или запись данных, не имеющих к нему никакого отношения, уже за его пределами.

С введением платформы .NET Microsoft существенно ужесточила требования по безопасности кода, и пришлось пересмотреть механизмы работы с массивами. Любые операции по работе с массивами жестко контролируются

со стороны .NET. Среда отслеживает правильность индексов, типы передаваемых данных, значения элементов, записываемых в массив, и многое другое. Это стало возможным благодаря переходу на объектную основу. Теперь, когда каждый массив представлен как экземпляр системного класса `System.Array`, появилась возможность проверять все операции прямо через этот класс. Что и позволило повысить безопасность и надежность приложений .NET.

Базовые алгоритмы для работы с массивами

При работе с массивами программист неизбежно сталкивается с необходимостью использования стандартных операций: поиска, сортировки, замены элементов, вычисления реальной длины, копирования частей массива и многих других. Для решения таких задач ему приходилось либо самостоятельно разрабатывать необходимые алгоритмы, либо использовать уже готовые решения. Но никогда ранее такие средства не были интегрированы в саму платформу языка. Теперь Microsoft, наконец-то, осознала важность и необходимость этих алгоритмов в практике программирования и включила их поддержку в платформу .NET. В табл. 3.4 перечислен базовый набор алгоритмов для работы с массивами в платформе .NET.

Таблица 3.4. *Базовый набор алгоритмов для работы с массивами*

Функция или свойство класса `System.Array`	**Операция**
`Length`	Определение полного количества элементов в массиве
`Rank`	Вычисление размерности массива
`BinarySearch`	Бинарный поиск в отсортированном одномерном массиве[1]
`Copy`	Копирование одного массива в другой, с учетом приведения типов и применения операций упаковки и распаковки
`GetLength`	Вычисление количества элементов в заданной размерности массива
`IndexOf`	Определение индекса первого элемента, совпавшего с шаблоном, переданным в качестве параметра (поиск элемента по образцу)
`Initialize`	Инициализация элементов массива, при помощи вызова конструкторов по умолчанию для каждого элемента массива
`Reverse`	Переворачивает массив
`Sort`	Сортировка массива

[1] — метод работает значительно быстрее линейного алгоритма, за счет того, что при каждой итерации количество непроверенных элементов сокращается вдвое.

Все операции представлены соответствующими свойствами или методами класса `System.Array`, и если вы используете массив в среде .NET, то он обязательно окажется экземпляром этого класса. Алгоритмы, предоставляемые им, обеспечивают эффективное решение большинства задач повседневного программирования. При этом все операции являются абсолютно безопасными и постоянно контролируются со стороны среды .NET.

Интерфейсы

Интерфейс является наиболее интересным и сложным для понимания типом. В классическом определении тип служит для организации доступа к данным. Интерфейсы же предоставляют доступ непосредственно к коду. Если заглянуть во внутренние механизмы работы интерфейсов, там, безусловно, обнаружатся специализированные внутренние данные, позволяющие осуществить доступ к коду. Но внешне использование интерфейсов выглядит странно, поскольку, с одной стороны, мы как бы работаем с данными, а с другой — с кодом. Поначалу эта необычность сбивает с толку.

В классическом понимании теории типов, интерфейсы являются абстрактными классами, необходимыми для обобщения доступа к классам. По сути дела, программисту предоставляется механизм, позволяющий описать функциональность класса, не задумываясь о способах ее реализации. Вводимый интерфейс строго регламентирует набор методов, свойств, событий, поддерживаемых классом, реализующим такой интерфейс. При этом подробности их реализации внутри класса программисту знать нет необходимости.

В рамках среды .NET на интерфейсы наложен ряд ограничений.

- Все члены интерфейсов должны обязательно быть объявлены как общедоступные (`public`). В языках высокого уровня это обычно достигается за счет введения запрета на применение любых спецификаторов доступа. Компилятор при транслировании кода автоматически подставит спецификатор `public`.
- К членам интерфейсов не могут применяться какие-либо атрибуты защиты.
- Интерфейсы не могут иметь конструктора.

В завершение раздела заметим, что теперь интерфейсы являются полноправными элементами языка и самой платформы, в отличие от прежних, которые эмулировались при помощи обычных структур.

Ссылки

Ссылки в среде .NET представляют собой особый вид объектов, которые инкапсулируют (скрывают внутри) реальные указатели. Такой подход позволяет среде исполнения контролировать все операции, производимые над указателями, гарантируя тем самым безопасность их использования. Также

введение системы ссылок позволило организовать новую подсистему памяти, поддерживающую полностью автоматическую сборку мусора. Необходимо упомянуть о наличии обратной связи между ссылками и менеджером памяти среды исполнения. В отличие от простых указателей, которые по своей сути являлись просто числовыми переменными, ссылки являются динамическими объектами, которые отчитываются о своих действиях перед менеджером памяти среды исполнения. Зная обо всех ссылках, среда исполнения может их полностью контролировать и следить за безопасным и более гибким распределением памяти.

Среда исполнения выделяет две общедоступные и безопасные операции, которые могут быть совершены над любой ссылкой:

- получение значения по ссылке;
- запись значения по ссылке.

Обе операции контролируются со стороны среды, которая проверяет правильность передаваемых данных и их типов.

Помимо этого, существует рад небезопасных операций, к которым относится вся адресная арифметика с прямым доступом к памяти. Для того чтобы воспользоваться небезопасными операциями, придется обратиться за специальным разрешением к среде .NET. В языке C# для этого необходимо использовать ключевое слово `unsafe`. При использовании небезопасных операций, необходимо помнить, что среда исполнения может в любой момент отказать вам, исходя из настроек внутренней политики безопасности, определяемой администратором.

Таким образом, использование ссылок, с одной стороны, сильно ограничивает наши возможности в динамичности и творчестве при работе с памятью, но, с другой стороны, безопасность и отказоустойчивость наших приложений значительно повышается.

3.6. Использование типов в среде .NET

В разделе будут рассмотрены общие аспекты создания и использования собственных пользовательских типов. Система типов, применяемых в программировании для платформы .NET, описана в рамках стандарта CTS. Стандарт включает в себя группу правил, регламентирующих использование типов, а также работу с их членами. Всего в CTS существует четыре раздела:

- применение атрибутов;
- контроль доступа к типам;
- наследование типов;
- описание членов типов.

Рассмотрим подробнее эти пункты.

Применение атрибутов

Атрибут представляет собой объект специального вида, содержащий дополнительную пользовательскую информацию о типах. Он может прикрепляться прямо к типам. Впоследствии можно читать эту дополнительную информацию при помощи сервисов технологии отражения. Атрибуты могут применяться к любым типам среды .NET, а также к любым их членам, начиная от обычных полей и заканчивая значениями, возвращаемыми функциями. Существует возможность применять атрибуты к сборкам и модулям. Основной идеей, заложенной в концепцию атрибутов, является возможность прикрепления атрибута к типу, а не к объекту.

Контроль доступа к типам

Контроль доступа к типам в среде .NET организован на качественно новом мощном уровне; Microsoft ввела несколько новых уровней ограничения доступа к типам, не существовавших ранее.

Условно все операции контроля доступа можно разделить на две группы: внешнетиповый и внутритиповый.

Первая группа операций позволяет контролировать доступ к самим типам, работая на уровне сборок. В этой группе поддерживается два вида доступа к типам, они приведены в табл. 3.5.

Таблица 3.5. *Виды доступа к типам на уровне сборок*

Тип доступа	Описание
`Public`	Позволяет обращаться к типу извне сборки
`Assembly`	Позволяет обращаться к типу только из самой сборки, в которой он описан

Кроме этого, существует возможность ортодоксального контроля доступа на уровне самих типов, позволяющая определить политику доступа к членам типов. Виды доступа, предоставляемые средой исполнения для членов типов, включают в себя:

- `Private` — доступен только из самого типа, в который входит член с таким атрибутом защиты;
- `Family` — доступен из этого типа, а также из типов потомков (в языках высокого уровня более известен как `protected`);
- `Assembly` — доступен только из сборки, в которой описан данный тип;
- `Family-and-Assembly` — доступен только из сборки и классов потомков типа, в котором он описан (в отличие от вида `Family`, разрешает получить доступ к члену только из самой сборки);

- `Family-or-Assembly` — доступен только из этого типа или на уровне сборки;
- `Public` — общедоступен везде и всегда.

Существует два дополнительных правила, которые обязаны всегда выполняться.

- Члены внутри интерфейсов всегда должны иметь атрибут защиты `Public`.
- При перекрытии виртуальных методов, уровень доступа должен только понижаться, а не повышаться.

Хотя большинство высокоуровневых языков не позволяет изменять уровень доступа для перекрытых методов, принципиально это вполне возможно. На уровне IL-кода разрешается задать любые спецификаторы доступа, компилятор языка не отслеживает этот момент.

Кроме того, в среде .NET выделяется два уровня контроля доступа к типам: уровень компилятора (`Compiler Control`) и уровень среды исполнения. На уровне компилятора проверка происходит во время транслирования программы в IL-код. Примечательно, что CTS разрешает компиляторам вводить собственные уровни доступа, отличающиеся от принятых в .NET.

Второй уровень контроля доступа более интересен, поскольку он происходит во время исполнения программы. Подобной функциональности вы не встретите ни в одной из классических объектно-ориентированных сред, вроде языка C++. В этом языке контроль доступа происходит лишь на уровне компилятора, а во время исполнения программы существует возможность обхода механизмов контроля. В среде .NET получить доступ к закрытому методу во время исполнения программы не удастся. Продемонстрируем на примере. Учитывая, что попытка получения доступа к закрытому члену типа будет пресечена еще на стадии компилирования программы, придется немного схитрить. Для начала напишем программу, в которой производится вызов ничем не примечательного общедоступного статического метода некоторого класса. Код примера представлен в листинге 3.6.

Листинг 3.6. Обращение к общедоступному статическому методу (шаблон)

```
/*
  Листинг 3.6
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
```

```
// Введем дополнительный класс, над которым будем экспериментировать.
class Some
{
  // Общедоступный метод, который мы будем использовать.
  // вне класса
  public static void Hello()
  {
    Console.WriteLine("Hello, World!");
  }
}
// Основной класс приложения.
class App
{
   // Точка входа в приложение.
  public static void Main()
  {
    // Обратимся к методу нашего экспериментального класса.
    Some.Hello();
  }
}
```

Пример будет успешно компилироваться и работать, выводя на консоль строку "Hello, World!".

Модифицируем пример таким образом, чтобы метод `Hello` класса `Some` был закрытым. Для этого придется временно декомпилировать приложение при помощи утилиты `ildasm.exe`, после чего изменим спецификатор доступа метода `Hello` со значения `public` на значение `private` (листинг 3.7).

Листинг 3.7. Декомпилированная версия примера 3.7 с небольшими изменениями

```
/*
  Листинг 3.7
  File:   Some.il
  Author: Дубовцев Алексей
*/
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                          //
.z\V.4..
```

```
  .ver 1:0:5000:0
}
.assembly Some
{
  // --- The following custom attribute is added automatically, do not
uncomment -------
  //  .custom instance void [mscor-
lib]System.Diagnostics.DebuggableAttribute::.ctor(bool,
  //
bool) = ( 01 00 00 01 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Some.exe
// MVID: {1FF00D4C-93BA-4451-AC29-38737E442D4C}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x06b50000
//
// ============== CLASS STRUCTURE DECLARATION ==================
//
.class private auto ansi beforefieldinit Some
       extends [mscorlib]System.Object
{
} // end of class Some
.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
} // end of class App

.class private auto ansi beforefieldinit Some
       extends [mscorlib]System.Object
{
  // Обратите внимание, мы изменили спецификатор доступа к методу.
  .method private hidebysig static void  Hello() cil managed
  {
    // Code size       11 (0xb)
```

```
    .maxstack  1
    IL_0000:  ldstr      "Hello, World!"
    IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000a:  ret
  } // end of method Some::Hello

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method Some::.ctor
} // end of class Some
.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main() cil managed
  {
    .entrypoint
    // Code size       6 (0x6)
    .maxstack  0
    IL_0000:  call       void Some::Hello()
    IL_0005:  ret
  } // end of method App::Main
  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method App::.ctor
} // end of class App
```

При помощи компилятора `ilasm` соберем готовую программу и попытаемся запустить. В результате, увидим сообщение о произошедшем в приложении исключении `MethodAccessException`. На консоль будут выведены следующие строки, описывающие исключение.

```
Unhandled Exception: System.MethodAccessException: Some.Hello()
   at App.Main()
```

Наследование типов

Среда .NET допускает наследование для следующих типов:

- Классы
- Структуры
- Интерфейсы

При этом для классов и структур допускается только линейное наследование, а для интерфейсов — множественное. Множественное наследование предполагает возможность указания более одного предка, а при линейном наследовании можно указать лишь одного предка. При этом необходимо отметить, что для классов и структур можно комбинировать линейное наследование от классов с множественным от интерфейсов. Фактически, для класса можно объявить в качестве предка единственный класс или структуру, а также несколько интерфейсов.

Таким образом, будут соблюдены все необходимые правила.

Описание членов типов

Сами по себе типы были бы абсолютно бесполезны, если бы не поддерживали столь мощного механизма описания членов, благодаря которому и стала возможной организация всей объектной инфрастуктуры среды .NET.

Всего существует пять видов членов типов (табл. 3.6).

Таблица 3.6. *Виды членов типов*

Член типа	Описание
События	Член, позволяющий отслеживать изменение состояния объекта. Он предоставляет сервисы для подключения и отключения обработчиков события
Поля	Позволяют задать данные, характеризующие внутреннее состояние объекта. Поля являются внутренним ядром объекта, реально содержащим информацию о нем
Вложенные типы	Типы, описанные внутри других. По сути дела, здесь работает лишь механизм разграничения подпространств имен, других механизмов, связывающих вложенный тип с родительским, не существует

Таблица 3.6 *(окончание)*

Член типа	Описание
Методы	Методы позволяют определять в типах операции, которые можно совершать над ним извне, а также предназначенные для внутреннего использования из самого типа. Операторы, конструкторы и деструкторы являются особым видом методов
Свойства	Специальные методы, которые извне выглядят подобно полям. Введены для удобства обращения к данным объекта извне

Наиболее интересными, из представленных здесь членов, являются вложенные типы. С одной стороны, они являются полноправными типами, а с другой — членами родительского типа, который вправе контролировать доступ к ним. Приведем пример, демонстрирующий работу с вложенными типами (листинг 3.8). Объявим два типа, вложенных в класс `SomeClass`, один из которых будет закрытым (`nestedZ`), а второй общедоступным (`nestedM`). По своей сути оба этих класса ничем не отличаются друг от друга, их функциональность одинакова, они оба предоставляют каждый по одному методу `DoIT`, выводящему на консоль приветствие "Hello, World!". Однако доступ к классу `nestedZ` ограничен со стороны внешнего для него класса `SomeClass`, а для класса `nestedM` доступ разрешен, тем же самым классом `SomeClass`. Рассмотренный пример наглядно показывает двойственность свойств вложенных типов.

Листинг 3.8. Демонстрация работы с вложенными типами

```
/*
  Листинг 3.8
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем дополнительный экспериментальный класс.
class SomeClass
{
  // Объявим закрытый вложенный класс с именем nestedZ.
  protected class nestedZ
  {
    // Объявим метод, с тем чтобы проверить доступ к нему.
    // Его функциональность в примере не играет роли.
```

```
    public static void DoIt()
    {
      Console.WriteLine("Hello, World!");
    }
  };
  // Объявим общедоступный вложенный класс с именем nestedM.
  public class nestedM
  {
    // Объявим метод, с тем чтобы проверить доступ к нему.
    // Его функциональность в примере не играет роли.
    public static void DoIt()
    {
      Console.WriteLine("Hello, World!");
    }
  };
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Попробуем обратиться к методу закрытого вложенного типа.
    SomeClass.nestedZ.DoIt();
    // На предыдущую строку компилятор выдаст сообщение:
    // error CS0122: 'SomeClass.nestedZ' is inaccessible
    // due to its protection level.
    // Что означает:
    // ошибка CS0122: 'SomeClass.nestedZ' недоступен, в связи с
    // его уровнем защиты.
    // Обратимся к общедоступному методу. Что, кстати говоря,
    // нам прекрасно удастся.
    SomeClass.nestedM.DoIt();
  }
};
```

Как и предполагалось, обращение к закрытому вложенному типу не увенчалось успехом. При попытке компиляции примера из среды разработки Visual Studio, будет выдано сообщение об ошибке (рис. 3.15).

Обращение к общедоступному вложенному типу произойдет без проблем.

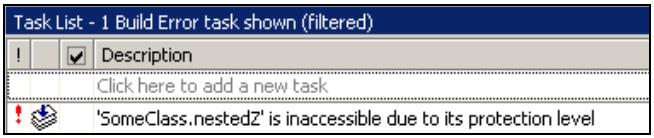


Рис. 3.15. Ошибка при обращении к закрытому вложенному типу-члену

Дополнительные спецификаторы, применяемые к членам типов

Помимо спецификаторов, предназначенных для контроля доступа к членам типов, существует группа спецификаторов, позволяющих задать дополнительные свойства для членов типов. В табл. 3.7 перечислены и кратко описаны все возможные спецификаторы, которые могут быть использованы при описании членов типов. Также в таблице указана область применимости для каждого из них.

Таблица 3.7. *Дополнительные спецификаторы, применяемые к членам типов*

Спецификатор	Применим к ...	Описание
`abstract`	методам, свойствам и событиям, типам	Позволяет указать, что реализации данного типа или члена не представлены, присутствует лишь его описание
`private`, `family`, `assembly`, `family and assembly`, `family or assembly`, `public`	любым членам типов	Определяет тип доступа `private` — только для использования из типа владельца данного члена `pamily` — доступен из семейства данного типа, то есть из самого типа, а так же из его потомков `pssembly` — доступен только внутри своей сборки `family and assembly` — доступен только из семейства этого типа, с условием, что доступ осуществляется внутри сборки описывающей тип `family or assembly` — доступен или из этой сборки, или из семейства этого типа `public` — общедоступен, ограничений на доступ нет
`final`	методам, свойствам и событиям	Виртуальный метод, который не может быть перекрыт типом-потомком
`initialize-only`	полям	Значение поля может быть задано единожды при инициализации типа и не может изменяться впоследствии

Таблица 3.7 *(окончание)*

Спецификатор	Применим к ...	Описание
`instance`	полям, методам, свойствам и событиям	Если метод не помечен `static` или `virtual`, то по умолчанию он является `instance`. Это означает, что для работы с членом типа необходима ссылка на объект, с которым будет производиться работа
`literal`	полям	Фиксированное значение, задаваемое только во время компиляции. Используется компиляторами большинства языков для определения констант
`newslot or override`	к любым членам типов	Определяет вид перекрытия члена в классе-потомке, при условии полного совпадения прототипа члена. `nwslot` — прячет унаследованные члены. При этом в базовом классе будет вызываться его член `override` — замещает определение на виртуальный метод. При этом в базовом классе будет вызываться новый член
`static`	полям, методам, свойствам и событиям	Будет создана единственная копия члена для всех экземпляров объекта. При этом для доступа к члену будет использоваться не объект, а напрямую его тип
`virtual`	методам, свойствам и событиям	Метод может быть замещен в типе-потомке. Теперь подобные методы могут быть как статическими, так и динамическими. В случае, если метод статический, подходящая его версия будет определяться прямо по ходу исполнения программы. Если же он динамический, при обращении к нему может понадобиться преобразование типов

Названия спецификаторов, представленные в таблице, определены в рамках стандарта CTS и используются в документации и на низком уровне среды исполнения. В языках высокого уровня имена спецификаторов могут значительно различаться. В табл. 3.8 приведены соответствия между общими и внутриязыковыми названиями спецификаторов.

Таблица 3.8. *Соответствие спецификаторов доступа для различных языков*

Спецификатор	C#	MC++	VB	JScript	IL
abstract	Abstract	__abstract	MustOver-ride	Abstract	Abstract
private	Private	Private	Private	Private	Private
family	pro-tected	Protected	Protected	protected	Family
assembly	Internal	public pri-vate:	Friend	Internal	Assembly
family and assembly	-	-	-	-	Famandas-sem
family or assembly	pro-tected internal	public pro-tected	Protected Friend	protected internal	Famorassem
public	Public	Public	Public	Public	Public
final	Sealed	__sealed	NotOver-ridable	final	Final
initialize-only					
literal	Const	Const	Const	const	Literal
newslot	New	-			
override	Override		Overrides	override	
static	Static	Static	Shared	static	Static
virtual	Virtual	Virtual	Overrida-ble	Автомати-чески	Virtual

Примечание

При использовании языка Jscript .NET все функции, определяемые в классах, являются виртуальными.

Перегрузка членов типов

Технология перегрузки пришла в .NET из C++-подобных языков. Она подразумевает возможность объявления двух или более членов типа с одинаковыми именами, но различными сигнатурами. К примеру, это могут быть два метода с одинаковыми именами, но принимающие разные по типу параметры.

```
void SomeMethod(int i);
void SomeMethod(string s);
void SomeMethod(int i, string s);
```

Технология перегрузки методов позволяет создавать очень гибкий код, автоматически подстраивающийся под нужды программиста. Используя перегруженные методы, программист больше не должен задумываться над выбором подходящей версии метода, компилятор самостоятельно выберет ее.

Наследование — перекрытие и сокрытие членов

В среде .NET технология перекрытия членов классов получила очень широкое развитие. Теперь, помимо уже ставшей стандартной технологии замещения (перекрытия) членов классов, появилась возможность сокрытия членов классов. При перекрытии виртуальных членов в классах-потомках, они будут замещаться для всей иерархии классов. Таким образом, если мы перекроем функцию в одном из классов-потомков, то и все базовые классы будут использовать новую версию функции. Новый вариант механизма перекрытия членов типа позволяет заместить функцию только для текущего типа и его потомков, базовые классы будут по-прежнему использовать свою версию метода.

Продемонстрируем работу технологии замещения и сокрытия на простом примере (листинг 3.9). Введем класс A с неким виртуальным методом `DoIt`. Затем унаследуем от него класс B, в котором также будет описан виртуальный метод `DoIt`. Введем еще класс C, который также будет являться потомком класса A.

Листинг 3.9. Демонстрация технологии перекрытия и сокрытия членов типов

```
/*
  Листинг 3.9
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем основной базовый класс, который будет являться
// базовым для остальных.
class A
{
  // Обратите внимание на следующую функцию. Сама по себе она не
  // представляет интереса, поскольку при ее описании не применялось
  // дополнительных спецификаторов. В этом случае
  // интересен код функции, который обращается к виртуальным
  // методам.
```

```
  public void Global()
  {
    // Вызываем метод DoIt, который является виртуальным, а
    // следовательно, этот вызов может ссылаться на функцию,
    // расположенную в классах-потомках, если те, конечно,
    // перекрывали данную функцию.
    DoIt();
  }
  // Объявим виртуальную функцию, которую могут перекрыть потомки
  // этого класса.
  public virtual void DoIt()
  {
    // Информируем пользователя о вызове
    // виртуального метода, причем именно из этого класса.
    Console.WriteLine("Do It from A");
  }
};
// Объявим первого потомка класса A.
class B : A
{
  // Объявим новую виртуальную функцию, которая, вопреки
  // логике, не перекроет метод DoIt предыдущего класса, поскольку
  // она объявлена с модификатором new. Это означает, что вызов
  // функции DoIt из класса потомка A, приведет к вызову его
  // собственной функции A::DoIt, объявленной в его классе. Если
  // же функция DoIt будет вызвана извне, то будет вызван именно
  // этот метод.
  public virtual new void DoIt()
  {
    Console.WriteLine("Do it from B");
  }
};
class C : A
{
  // А здесь мы перекроем метод DoIt для всей иерархии наследования
  public override void DoIt()
  {
    Console.WriteLine("Do it from C");
  }
```

```
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Для начала обратимся к классу B,
    // реализующего сокрытие функций.
    Console.WriteLine("Test B class");
    B b = new B();
    // Вызовем метод Global класса A, который, в свою очередь,
    // обратится к методу DoIt, своего же класса, поскольку
    // класс B остановил перекрытие при помощи объявления
    // новой функции с модификатором new.
    b.Global();
    // Обратимся извне к виртуальному методу.
    b.DoIt();
    // Теперь обратимся к классу C,
    // реализующего замещение функций.
    Console.WriteLine("\nTest C class");
    C c = new C();
    // Вызовем метод Global класса A, который, в свою очередь,
    // обратится к методу DoIt класса C, который заместил
    // его для всей иерархии классов.
    c.Global();
    // Обратимся извне к виртуальному методу.
    c.DoIt();
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки:

```
Test B class
Do It from A
Do it from B
Test C class
Do it from C
Do it from C
```

Такая возможность может быть весьма полезна, если потребуется изменить поведение класса, не меняя при этом его внешних интерфейсов и принципов работы базовых классов.

3.7. Корневой объект — *System.Object*

Платформа .NET является четко спроектированной и грамотно построенной объектно-ориентированной средой, в основе которой лежит базовый класс `System.Object`. Это фундамент всей системы типов и объектов среды .NET. Абсолютно любой объект среды .NET является потомком данного класса, даже если это и не указано явно. Этот объект введен для обобщения и централизации всей системы типов. Класс `System.Object` дает универсальность доступа к любым объектам среды .NET, что означает возможность обращения к абсолютно любому объекту через класс `System.Object`. Это может быть сложный пользовательский объект, число, строка или встроенный объект общей библиотеки классов. При первом знакомстве со средой .NET подобные конструкции поражают даже бывалых программистов.

```
5.ToString();
"Hello, World!".GetType();
```

При том, что они выглядят несколько неправдоподобно, в то же время они абсолютно корректны. В выражении число 5 и строка "Hello, World!" будут рассматриваться средой исполнения как объекты класса `System.Object`. А операторы `ToString` и `GetType` являются его методами. Продемонстрируем работоспособность этого кода на простейшем примере (листинг 3.10).

Листинг 3.10. Демонстрация работы с произвольными объектами через класс `Object`

```
/*
  Листинг 3.10
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
```

```
  {
    // Продемонстрируем обобщенность объектов внутри среды
    // через тип Object.
    Console.WriteLine(5.ToString());
    Console.WriteLine("Hello, World!".GetType().ToString());
  }
}
```

В результате работы примера на консоль будут выведены следующие строки:

```
5
System.String
```

Обратите внимание, для доступа к объектам использовался тип `Object`, но при этом сами объекты не меняли своего типа. Мы как бы рассматривали их через призму типа `Object`.

Опишем простой класс `Hello`, в котором определен один метод и одно поле — его персональные члены.

```
// Новый класс является прямым потомком типа Object, хотя и неявно.
class Hello
{
  void DoIt()
  {
    Console.WriteLine("Hello, World!");
  }
  int Data;
}
```

Введенный класс автоматически является потомком от класса `Object`. Таким образом, его экземпляры будут также автоматически принадлежать к объектам типа `Object`. В результате, через тип `Hello` можно получить доступ к полному объекту, а через тип `Object` лишь к части его методов (рис. 3.16).

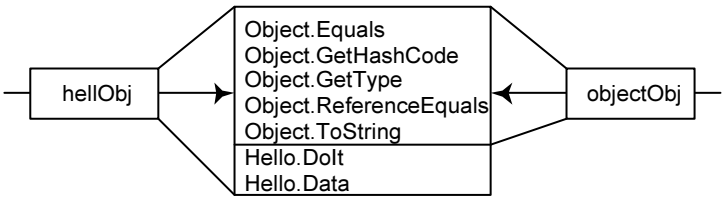


Рис. 3.16. Организация доступа к объекту через собственный тип и через класс `Object`

Разработчики среды .NET заложили в класс `Object` очень мощную функциональность, которая, к удивлению пользователей, представлена всего лишь несколькими методами (табл. 3.9).

Таблица 3.9. *Описание членов типа* `Object`

Член класса	Описание
`Equals`	Позволяет определить идентичность двух объектов
`GetHashCode`	Возвращает уникальный хеш-код объекта
`GetType`	Возвращает тип объекта
`ReferenceEquals`	Определяет, указывают ли ссылки на один и тот же объект
`ToString`	Возвращает строковое описание объекта
`MemberwiseClone`	Создает клон объекта — его полноправную копию

Метод *Equals*

Предназначен для универсального сравнения объектов.

```
public virtual bool Equals(
   object obj
);
public static bool Equals(
   object objA,
   object objB
);
```

Метод является членом класса `Object`, являющегося корневым для всей системы типов среды .NET, и поэтому может быть вызван абсолютно для всех объектов среды .NET. Именно благодаря этому методу стало возможным создание универсальных классов коллекций, которые широко используются в среде .NET. Работа многих коллекций невозможна без проведения операции сравнения над их элементами.

Примечание

Универсальной является коллекция, позволяющая хранить объекты любых типов. В C++ подобные коллекции реализовались либо при помощи шаблонов C++, либо при помощи "бестиповых" (void*) указателей. Ранее, при проектировании универсальных коллекций, программистам приходилось прибегать к хитроумным трюкам, которые все равно не давали необходимой гибкости.

Метод *GetHashCode*

Разработчики .NET решили, что полезно иметь простой механизм идентификации объектов. Было решено ввести хеширование объектов, которое базируется на специальных, универсальных для каждого объекта значениях.

```
public virtual int GetHashCode();
```

При помощи метода вы получаете некоторое целочисленное значение и можете быть уверены, что оно окажется универсальным для текущей сессии работы приложения. Необходимо отметить, что алгоритмы, встроенные в класс `Object`, не гарантируют уникальности значения хеша на протяжении всей работы приложения. Дело в том, что `Object` рассчитывает хеш, учитывая адрес объекта в памяти, а он может изменяться в процессе работы приложения. Этот эффект связан с процессом сборки мусора, который будет обсуждаться в *главе 10*. Поэтому не исключено, что хеш объекта изменится или же совпадет с другим. Поэтому Microsoft рекомендует использовать собственные механизмы генерации хеша, которые учитывали бы природу используемых объектов и их внутреннее содержание.

Метод *GetType*

```
public Type GetType();
```

Возвращает объект `Type`, предоставляющий доступ к информации о типах этого объекта. Метод незаменим при работе с технологией отражения, он позволяет наиболее быстро добраться до необходимой информации о типах.

Метод *ToString*

```
public virtual string ToString();
```

Человеку, предложившему ввести метод `ToString` в класс `Object`, можно смело поставить памятник. Использование метода беспрецедентно облегчает отладку приложений, поскольку позволяет получить строковую информацию о любом объекте среды .NET. Программисты по достоинству оценили эффективность работы с этим средством и широко используют его в практической работе.

Метод *MemberwiseClone*

Создает неполную копию объекта. Если в полях объекта присутствуют ссылки, то будут скопированы только их значения, но не сами объекты, на которые они указывают. Таким образом, копия объекта, созданного при помощи метода, может оказаться неполной. Отрицательным моментом метода является существенное снижение производительности приложения, вследствие активного использования технологии отражения.

```
protected object MemberwiseClone();
```

Обратите внимание, метод объявлен при использовании модификатора `protected`, что предотвращает его использование извне. Для разрешения копирования объекта внешним клиентам, предназначен интерфейс `ICloneable`, содержащий единственный метод `Clone`, который должен реализовать сам разработчик объекта.

```
object Clone();
```

В листинге 3.11 представлен пример объекта, поддерживающего клонирование.

Листинг 3.11. Пример объекта, поддерживающего клонирование

```
/*
  Листинг 3.11
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Пример объекта, поддерживающего клонирование.
class Some : ICloneable
{
  object ICloneable.Clone()
  {
    // Клонируем объект по полям
    base.MemberwiseClone();
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
  }
}
```

Если класс является сложным, тогда в нем необходимо переопределить собственную версию метода `MemberwiseClone`, в которой бы обеспечивалось корректное копирование объекта.

Глава 4

Сборки

В главе рассматривается архитектура и внутренние механизмы работы технологии сборок, являющейся фундаментом среды .NET. Вначале дается краткий экскурс в историю технологий распределения кода, который в дальнейшем позволит понять многие решения, положенные в основу кода платформы .NET.

Акцент сделан на подробном описании внутренних механизмов работы, понимание которых позволит в совершенстве овладеть большинством программных средств, предоставляемых платформой .NET. Будут рассмотрены как простые вопросы, например, создание строго именованных сборок, так и более сложные, вроде низкоуровневого механизма загрузки сборок.

4.1. Немного истории

Сначала ознакомимся с историей развития технологий распределения программного кода, предшествующих появлению сборок. После знакомства с ними станут понятны причины введения многих решений, нашедших себя в сборках, и кажущиеся с первого взгляда громоздкими и неуместными.

Первые шаги

С момента написания первых серьезных приложений перед программистами встала проблема разделения и повторного использования кода. Безусловно, самым тривиальным решением является банальное копирование текста исходного кода программы из старого проекта в новую программу. Несмотря на кажущуюся простоту подхода, он оказался неприемлем при разработке более или менее серьезных проектов. В большинстве случаев, использование подобного подхода приводило к путанице и возникновению нескончаемых потоков ошибок и нестыковок. Было очевидно, что проблема требовала более глубокого анализа и рассмотрения.

Первым шагом на пути решения задачи распределения кода стало появление процедурного программирования. Это было большим шагом вперед, по сравнению с линейным стилем написания программ по методу "сверху-вниз", присущим первым версиям языков программирования. Процедурный подход оказался достаточно успешным решением, и технология благополучно дожила до наших дней, став повсеместным стандартом де-факто. Без использования функций и процедур сейчас не обходится ни один язык программирования, начиная от простых языков сценариев, предназначенных для администрирования операционных систем, и заканчивая серьезными языками для разработки системных приложений.

Но одного процедурного подхода оказалось недостаточно, поскольку он позволял разбивать программу на логические части лишь в рамках одной языковой среды разработки. Это означает, что разработчик программы на языке Паскаль не имел возможности использовать функции, написанные на C++ или других языках, что порой было крайне необходимо. Более того, в те времена программисты даже и не мечтали об удобной межъязыковой интеграции. Это казалось фантастикой, существующей в несбыточных мечтах программистов. Попытки разработки систем интеграции, конечно, предпринимались, но они оказались неприемлемы для широкого использования, поскольку большинство из них создавались программистами, как говорится, "на коленке". Основная проблема заключалась в отсутствии общего стандарта.

Постепенно количество языков и сред разработки неудержимо увеличивалось. Создатели средств разработки понимали, что одного процедурного подхода для решения проблемы явно недостаточно. Вследствие чего, во многие компиляторы была введена возможность создания модулей. То есть код разбивался на отдельные блоки, предоставляющие внешний интерфейс доступа и скрывающие реализацию самого кода от пользователя. Однако такая возможность не могла претендовать на роль единого средства интеграции программного кода, поскольку не была общестандартизованной. Каждое средство разработки имело собственную технологию модулей, что ограничивало ее использование только этой средой. Требовалась технология, которая позволила бы взаимодействовать коду на уровне всей операционной системы, независимо от языка и среды разработки приложения.

Пионером на пути решения проблем глобальной интеграции и распределения кода стала технология программных прерываний. Она позволяла создавать и вызывать общесистемные сервисы из любых программ, написанных на любых языках. Каждая программа могла перекрыть любое из 255 доступных программных прерываний, предоставив, таким образом, общесистемный глобальный сервис.

Это был первый унифицированный и четко стандартизованный интерфейс, используемый для глобального взаимодействия между программами. Он позволяет осуществлять вызовы практически из любых языков программирования и поддерживается непосредственно операционной системой, совместно

с процессором, что делает его применение надежным и практичным. К основным недостаткам метода прерываний относятся:

- трудоемкая и непрозрачная для синтаксиса языков реализация вызовов;
- трудоемкость в обеспечении отказоустойчивости программ и обработки исключительных ситуаций;
- ограничение числа возможных сервисов, определяемое размером таблицы векторов прерываний;
- отсутствие встроенной системы безопасности;
- концептуальная несовместимость с синтаксисом и базовыми принципами языков высокого уровня (прерывания предполагают оперирование регистрами, понятия о которых в языках высокого уровня попросту не существует);
- отсутствие контроля типов и корректности данных;
- отсутствие наглядности действия механизма и простоты его использования.

Последний пункт стал, пожалуй, основным препятствием к повсеместному использованию прерываний в качестве основной технологии распределения программного кода. Было бы неверно говорить, что прерывания не получили распространения. Во времена операционной системы MS-DOS, они были наиболее популярным способом предоставления общедоступных программных сервисов. Эта технология не забыта и сегодня — она с успехом применяется в недрах ряда современных операционных систем. К примеру, в ОС Microsoft Windows 2000 все вызовы ядра проходят через программное прерывание под номером `2Eh`, хотя для программистов высокого уровня, использующих обычное Windows API, это не является явным. Рассматривая функцию API `ReadFile`, можно показать, что ее вызов приводит на определенном этапе к обращению к прерыванию `2Eh` с параметром `0A1h`, передаваемым в регистре `eax`. Прерывание `2Eh` является стандартным шлюзом для подавляющего большинства API-вызовов на уровень ядра, а параметр `0A1h` определяет искомую функцию, в данном случае `ReadFile`.

Примечание

В последних версиях современных операционных систем внутрисистемные вызовы предпочитают организовывать не через прерывания, а при помощи новой специализированной инструкции `syscall`. По своей природе `syscall` достаточно сильно напоминает прерывания, но имеет более высокие показатели производительности, а также автоматически повышает уровень защиты до нулевого кольца.

Динамические библиотеки

Следующим шагом на пути распределения программного кода явилось создание динамически подключаемых библиотек, знакомых читателю под широко известной аббревиатурой DLL (Dynamic Link Library). С появлением

этой технологии стало возможным создание внешних, полностью самостоятельных универсальных библиотек. Обращение к коду которых могло наглядно осуществляться из любых языков и средств разработки.

На технологии динамических библиотек построен весь программный интерфейс верхнего уровня операционных систем от Microsoft. Любое API, любой сервис, так или иначе, базируются на DLL. Основное преимущество этой технологии заключается в том, что в большинстве случаев она позволяет осуществить разделение кода прозрачно для программиста. Так, к примеру, для большинства программистов механизм вызова API-функций является скрытым. Они просто обращаются к нужным им функциям, не задумываясь о том, как это работает.

Примечание

Для DOS, а также для многих других операционных систем, существует поддержка технологии динамически подключаемых библиотек. Вот только реализация настоящего механизма будет несколько отличаться и носить характерные особенности этой операционной системы.

Принцип работы динамических библиотек несложен, если, конечно, не вдаваться в подробности внутренней реализации.

Сама библиотека содержит откомпилированный процессорный код и ряд служебных таблиц, описывающих содержащиеся в ней функции. По запросу программы или, что более вероятно, загрузчика исполняемых файлов Windows, код библиотеки копируется в ее адресное пространство, становясь с этого момента доступным для прямого вызова. После этого обращение к функции может быть осуществлено непосредственно при помощи инструкции `call`, `jmp` или путем прямого изменения регистра IP (здесь и далее будет рассматриваться только программы, ориентированные на архитектуру x86). На верхнем же уровне вызовы происходят как обычное обращение к функции, в привычной для языка лексике, а о низкоуровневых подробностях заботится сам компилятор.

Процесс загрузки динамической библиотеки схематично изображен на рис. 4.1.

Для максимального упрощения описания механизма работы динамических библиотек ничего не было сказано о переадресации функций, таблицах импорта и экспорта, проецировании кода, пересчета смещений, а также о многом другом, но в общих чертах все осталось верно. Для получения более полной информации о работе динамических библиотек, стоит обратиться к прекрасной книге Джефри Рихтера (Jeffrey Richter): "Programming Application for Microsoft Windows" или к документации Microsoft Platform SDK.

Динамические библиотеки во времена их разработки были прекрасным средством распределения кода и межъязыковой интеграции. Правда, при их использовании возникали незначительные проблемы с выбором формата

вызова функций, поскольку разные языки использовали разные форматы вызовов функций.

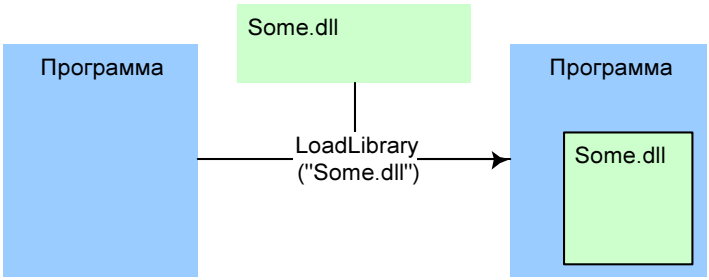


Рис. 4.1. Процесс загрузки динамической библиотеки

Примечание

Форматом вызова функции называется соглашение о способе передачи ее параметров и возвращении результата ее работы. Всего существует семь форматов вызовов: `stdcall`, `cdecl`, `fastcall`, `thiscall`, PASCAL, FORTRAN, SYSCALL. Причем последние три считаются безнадежно устаревшими и более официально не поддерживаются.

Но в итоге стандартом де-факто стал `stdcall`, который поддерживается всеми современными компиляторами и используется в подавляющем большинстве динамических библиотек, представляющих глобальные API-сервисы.

Для подключения функций из динамической библиотеки используется ее символьное имя, что является более наглядным, чем вызов прерывания по его номеру. К тому же, вызов функции может происходить в обычном синтаксисе языка.

Правда, в отличие от родных функций языка, при вызове динамических функций не проверяется информация о типах, а также корректность передаваемых параметров, поскольку эта информация не предоставляется библиотеками. Вследствие чего, при их неосторожном использовании, не исключено возникновение конфликтных ситуаций, а также неправильная работа программ. Правда, такие ситуации случаются довольно редко, поскольку, в основном, динамические библиотеки подключаются статически, на стадии сборки программ. А в этом процессе участвуют специальные библиотеки компоновки (в частности, `lib`-файлы), при помощи которых компилятор может проконтролировать соответствие типов функ-

ций и их параметров на стадии компиляции. Что существенно снижает риск некорректного использования функций из динамических библиотек, но важно отметить, что на стадии исполнения никаких проверок не происходит.

Помимо возможных неприятностей с вызовами, динамические библиотеки принесли в жизнь программистов куда более серьезную проблему, которую окрестили адом динамических библиотек (DLL Hell). Для того чтобы пояснить суть этой проблемы, необходимо в общих чертах обрисовать механизм подключения динамических библиотек.

Способы подключения динамических библиотек

Существует два способа подключения динамических библиотек: статический и динамический. Наиболее распространен статический тип подключения. Он подразумевает связывание с библиотекой во время компиляции приложения. Делается это при помощи добавления записи в таблицу импорта исполняемого файла. Такая запись представляет собой имя файла библиотеки, а также список используемых из нее функций. Перед началом исполнения программы, загрузчик Windows проверяет записи в таблице импорта, автоматически подгружает необходимые библиотеки и при помощи специального механизма связывает указанные функции со ссылками внутри исполняемого файла. Таким образом, загрузка библиотек происходит автоматически, без явного участия кода программы. А соответственно и программиста.

Наряду со статическим типом подключения динамических библиотек, существует возможность динамической загрузки библиотек. То есть код программы имеет возможность самостоятельной загрузки библиотек по ходу ее исполнения. Эта операция осуществляется при помощи API-функции, расположенной в файле `kernel32.dll`, и которая, в свою очередь, должна быть статически подключена к программе.

```
HMODULE LoadLibrary(
  // Имя файла, содержащего динамическую библиотеку.
  LPCTSTR lpFileName
);
```

Примечание

В принципе, программа может и не подключать библиотеку `kernel32.dll` статически. Все равно, она автоматически подгружается в адресное пространство приложения, поскольку используется загрузчиком исполняемых файлов Windows. Но в этом случае, для того чтобы использовать функции этой библиотеки, придется самостоятельно просканировать определенные области адресного

пространства. Такая технология в обычных программах не применяется, а используется лишь в специальных целях, для создания защит или вирусов.

После загрузки библиотеки в адресное пространство процесса, необходимо получить адрес функции. Наиболее просто сделать это можно, воспользовавшись следующей функцией.

```
FARPROC GetProcAddress(
  // Описатель загруженной библиотеки
  HMODULE hModule
  // Имя функции
  LPCSTR lpProcName
);
```

Для этого ей необходимо передать в качестве первого параметра описатель библиотеки, загруженной ранее при помощи функции `LoadLibrary`, а в качестве второго параметра — имя необходимой нам функции или же ее порядковый номер (ординал). В результате, мы получим адрес этой функции внутри адресного пространства процесса (ведь библиотека с функциями уже загружена внутрь). После чего можно обращаться к этой функции любыми, доступными нам, средствами, начиная от обычного вызова функции в стиле используемого языка программирования и заканчивая прямым вызовом команды `call` по полученному адресу.

Если взглянуть на оба способа подключения библиотек со стороны, то становится ясно, что фактически они идентичны. В обоих случаях, для загрузки библиотеки используется одна и та же функция `LoadLibrary`. Только в первом случае ее вызывает загрузчик исполняемых файлов Windows, а во втором — сама программа.

Теперь, когда мы полностью ознакомились с механизмом загрузки динамических библиотек, можно перейти к детальному рассмотрению проблемы, окрещенной адом динамических библиотек.

Отметим, что для загрузки динамической библиотеки используется лишь имя файла, в которой она расположена. Ранее Microsoft рекомендовала хранить все общие динамические библиотеки в едином хранилище — каталоге `system`, для систем класса 9x, и каталоге `system32`, для систем класса NT. Поиск динамических библиотек в этих каталогах производится автоматически, без участия программы.

Примечание

Эта рекомендация была продиктована желанием сэкономить место на жестких дисках пользователей. Дело в том, что большинство программ используют одни и те же, широко распространенные библиотеки, а их размещение в общем

централизованном хранилище позволяло избежать дублирования одинаковых файлов на жестких дисках пользователей, что, в свою очередь, освобождало бесценные байты для других программ.

Поскольку такое хранилище является общедоступным, то, нередко, возникала ситуация, когда библиотеки замещались совершенно другими, но с идентичными файловыми именами. Эта ситуация наверняка знакома читателю. Выражается она обычно в том, что при попытке запуска приложения, на экране появляется диалоговое окно, сообщающее о проблеме подключения динамической библиотеки (рис. 4.2).

Рис. 4.2. Не найдена необходимая функция в динамической библиотеке

В окне выведено сообщение о том, что в библиотеке `Some.dll` не была найдена функция `hello`. При этом обратите внимание, что сама библиотека была успешно найдена и загружена. В случае если загрузчик исполняемых файлов обнаружит несоответствие библиотек и выдаст подобное сообщение об ошибке, можно считать, что вы обошлись малой кровью. Все может оказаться намного хуже, к примеру, если будет загружена библиотека, которую вроде бы хотелось использовать, но только другой версии. То есть в ней будут все необходимые нам функции, только работать они будут совершенно по-иному. А поскольку в библиотеке не содержится никакой информации о прототипах функций и проверить правильность передаваемых в нее параметров нет никакой возможности, то, скорее всего, при обращении к ней возникнет исключительная ситуация.

Примечание

Прототипом функции называется информация о количестве и типе параметров, передаваемых в функцию, а также о типе значения, возвращаемого функцией.

А если это произойдет, то можно гарантированно говорить о том, что приложение аварийно завершит свою работу.

Итак, суть проблемы ада динамических библиотек заключается в подмене необходимых библиотек. Происходит это из-за отсутствия механизма устойчивой идентификации необходимых библиотек. Даже неискушенному программисту будет понятно, что идентификация лишь по имени файла не является надежной. Хотя бы с той точки зрения, что имена файлов могут случайно совпасть.

Позднее Microsoft разобралась, что гораздо безопаснее хранить библиотеки совместно с приложениями, и жертвы дисковым пространством вполне оправданы, перед лицом столь серьезной проблемы. Microsoft, конечно же, порекомендовала хранить библиотеки совместно с приложениями. Но, к сожалению, большинство программистов не только не знакомо с этой рекомендацией, но и вообще ничего не слышали о проблеме ада динамических библиотек и до сих пор размещают свои творения прямо в каталоге `system32 (system)`. Чтобы убедиться в правильности этого простого утверждения, просто загляните в этот каталог и посчитайте, сколько там содержится файлов с расширением `dll`. У себя я их насчитал 1205. Даже если предположить, что процентов шестьдесят из них принадлежит самой операционной системе, то количество потенциально опасных библиотек все равно остается внушительным.

Управление версиями динамических библиотек

Осознав всю серьезность проблемы, Microsoft разработала механизм управления версиями динамических библиотек. Сделано это было при помощи введения дополнительного ресурса VERSIONINFO, содержащего информацию о версии, а также специального набора функций Versioning API, предназначенного для управления им. Функции этого API:

```
GetFileVersionInfo
GetFileVersionInfoSize
VerFindFile
VerInstallFile
VerLanguageName
VerQueryValue
```

Программистам предлагалось самостоятельно управлять версиями необходимых им библиотек при помощи функций разработанного API. Но, как уже упоминалось выше, большинство программистов ничего не подозревало о проблемах ада динамических библиотек, и, как следствие, это API не было востребовано широкими массами. По-хорошему, API должно было действовать на уровне загрузчика исполняемых файлов Windows и работать в автоматическом режиме. Но это уже нельзя было реализовать, не потеряв совместимости с предыдущими версиями приложений.

Для поддержки старых приложений, в новых библиотеках приходится оставлять даже морально устаревшие функции. К примеру, в псевдоядре Windows, библиотеке `kernel32.dll`, существует множество функций, которые не используются современными приложениями, но оставлены из-за необходимости поддержки совместимости со старыми.

Примечание

Вы сможете найти список всех устаревших функций в Platform SDK на странице Obsolete Windows Programming Elements.

На сегодняшний момент проблема ада динамических библиотека практически решена полностью, при помощи введения требований совместного хранения динамических библиотек и исполняемых файлов непосредственно в каталогах программ, а также при участии сервисов защиты системных файлов System File Protection и Windows File Protection. Но вероятность возникновения проблем все же остается и она не такая уж и маленькая, как может показаться с первого взгляда.

Компонентный подход COM

Следующим этапом развития технологий распределения и интеграции кода стало появление компонентной объектной модели (COM, Component Object Model). Эта технология впервые была реализована в системе Windows 95.

Она уже позволяла разделять программы на отдельные независимые компоненты. Компоненты, в отличие от своих предыдущих собратьев, подключались уже не по имени файла, а при помощи специального глобального идентификатора (GUID) и были абсолютно независимы от приложения.

Примечание

GUID (globally unique identifier) — глобальный уникальный идентификатор. Он представляет собой 128-битное число, которое создается при помощи специального алгоритма, гарантирующего его полную уникальность. Алгоритм учитывает: время (с точностью до наносекунд) и географическое положение (номер сетевой карты). Такой хитрый способ генерации GUID позволяет свести практически до нуля вероятность создания одинаковых идентификаторов. Правда, нельзя не упомянуть о том, что прецеденты совпадения идентификаторов уже наблюдались, но это были единичные случаи и по этому поводу не стоит беспокоиться.

Каждому компоненту сопоставлялся свой уникальный идентификатор, по которому, в свою очередь, можно было получить полную информацию о нем в базе данных компонентов. В ней хранится информация по каждому из компонентов, начиная от имени файла, в котором расположен сам компонент, и заканчивая сетевыми настройками. Вы можете самостоятельно ознакомиться с содержанием базы данных COM, заглянув в раздел реестра, указанный в примечании.

Примечание

Основная база данных COM находится в реестре, в разделе `HKEY_CLASSES_ROOT`. А записи GUID-идентификаторов хранятся в подключе `HKEY_CLASSES_ROOT\Clsid`. Также дополнительную информацию о компонентах можно узнать здесь: `HKCR\Interface`, `HKCR\TypeLib`.

Для усиления политики управления версиями, идентификаторы были введены не только на уровне имен самих компонентов, но и на уровне функций, экспортируемых ими. Правда, теперь компоненты экспортировали не отдельные функции, а целые группы, объединенные в интерфейсы. Каждому такому интерфейсу сопоставляется уникальный 128-битный GUID-идентификатор, который исключает возможность случайного использования другого интерфейса. Казалось бы, такая мощная система разделения кода и управлениями версиями была неуязвима, но оказалось, что она не так уж и надежна. Проблемы пришли сразу же с двух сторон. Во-первых, технология COM все же базировалась на динамических библиотеках, в которых размещались сами компоненты. А с библиотеками зачастую происходила путаница, связанная с совпадением имен файлов, в которых они расположены. Но это была еще не самая страшная проблема. Основная неприятность крылась в базе данных COM, расположенной в реестре. Работа с этой базой предполагалась напрямую, без использования специального дополнительного API, которое обязательно бы следовало ввести. А поскольку раздел реестра, в котором хранится база данных COM, является общедоступным, то он, после продолжительной эксплуатации Windows, в большинстве случаев приходил в нерабочее состояние. С этой ситуацией знакомы большинство из пользователей, — когда после установки большого количества программ на компьютер, он приходил в неработоспособное состояние, проявляющее себя в очень странных конфликтах и неадекватном поведении программ.

4.2. Новое решение проблемы надежности

При проектировании платформы .NET перед ее разработчиками была поставлена задача: создать новую технологию, которая разом решила бы все проблемы, существовавшие в предыдущих технологиях распределения программного кода. И похоже, на сей раз Microsoft удалось решить поставленную задачу. И даже если не полностью, то в достаточном на ближайшее будущее объеме.

В основе новой технологии лежат сборки (assembly), которые являются наименьшими строительными блоками .NET, призванными обеспечить безопасное разделение кода в управляемых приложениях.

Что такое сборки?

Сборки — это наименьшие строительные блоки-"кирпичики", из которых построена вся платформа .NET. На них возложены следующие задачи.

- Сборки обеспечивают хранение кода приложений и библиотек среды .NET.

- Сборки определяют границы безопасности. Для них могут быть установлены атрибуты защиты, определяющие политику безопасности для кода, расположенного внутри них.
- Сборки разделяют границы типов. Имя, используемое при работе с любым типом, неявно включает в себя имя сборки, в которой он расположен. Благодаря этому, два типа с одинаковыми именами, расположенные в разных сборках, не идентичны.
- Помимо задачи хранения кода, сборки используются для хранения вспомогательных ресурсов, в качестве которых может выступать информация любого типа: картинки, текстовые файлы или другая произвольная пользовательская информация.
- Сборки являются основным механизмом, обеспечивающим поддержку политики версий для приложений .NET. Отличие версий может существовать только на уровне самих сборок, но не ниже. То есть любые программные элементы внутри сборок уже будут иметь общую версию родительской сборки или же вообще не будут иметь версии. На самом деле это означает одно и то же.
- Сборки поддерживают возможность распределения кода приложений. Они (сборки), как впрочем и их отдельные части, могут находиться на удаленных серверах.
- Сборки ответственны за работу технологии "прямого запуска" (Side-By-Side Execution), которая позволяет в одном адресном пространстве одновременно использовать две одинаковые сборки разных версий.
- Сборки обеспечивают легкую и быструю инсталляцию приложений .NET.

Виды сборок

Сборки разделяются на два вида: статические и динамические. Статические сборки располагаются на диске в файлах формата PE (Portable Executable) и могут содержать заранее откомпилированный IL-код, а также дополнительные ресурсы. Динамические сборки располагаются прямо в памяти и не имеют дискового файла. Хотя они вполне могут быть сохранены на диск после исполнения. Такие сборки создаются во время исполнения программы при помощи специальных средств общей библиотеки классов (FCL), названных технологией отражения (reflection).

Далее мы будем рассматривать только статические сборки. Но все сказанное далее будет справедливо и для динамических сборок, за тем лишь исключением, что большинство описанных здесь операций придется проводить при помощи технологии отражения.

Статические сборки

Статические сборки располагаются в файлах формата PE. Что на первый взгляд удивительно, поскольку формат PE эксплуатируется уже много лет и на сегодняшний день является морально устаревшим. Однако на самом деле оказывается, что формат PE служит лишь оболочкой для реальных сборок .NET. То есть сборки .NET вкладываются в файл формата PE (рис. 4.3).

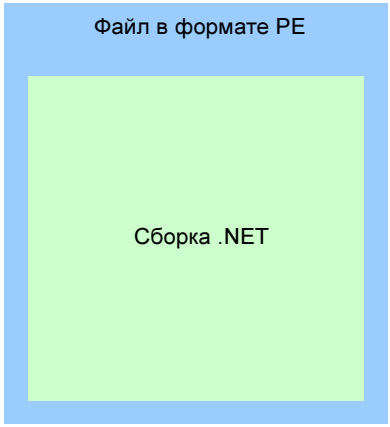


Рис. 4.3. Статическая сборка, вложенная в файл формата PE

Конечно, можно было создать для сборок новый формат файлов, как, к примеру, поступила фирма Sun при разработке языка Java.

Примечание

Откомпилированные Java-приложения распространяются в файлах специального формата с расширением `class`. Этот формат является полностью независимым от используемой платформы и может исполняться на любой системе под управлением виртуальной машины Java.

На первый взгляд, Microsoft поступила более чем странно, используя для хранения сборок файлы формата PE и тем самым, казалось бы, ограничив переносимость своих приложений классом операционных систем Windows. В других операционных системах поддержка формата PE не является встроенной. Но такое решение оказалось весьма продуманным и оправданным. На самом деле, использование формата PE нисколько не сужает круг платформ и не влияет на переносимость приложений .NET. Дело в том, что сборки загружаются и исполняются под управлением виртуальной машины .NET и формат файлов никакого значения не имеет. В чем можно прекрасно убедиться на примере Java.

Но зато выбор этого формата позволил Microsoft создать беспрецедентную по мощности систему взаимодействия .NET-кода с родными сервисами

операционных систем класса Windows. В большинстве случаев приложения .NET воспринимаются системой как родные, именно за счет PE формата файлов, в которых они размещаются. Вызовы между системой и такими приложениями идут при помощи стандартных интерфейсов, предоставляемых форматом PE: загрузка динамических библиотек, функции импорта и экспорта.

Перейдем к рассмотрению самих файлов статических сборок, которые вложены внутрь PE файлов. Они включают в себя четыре основных элемента:

- манифест, описывающий сборку;
- метаданные, описывающие типы, входящие в сборку;
- собственно откомпилированный код сборки в виде .NET-байт кода, который может быть представлен в удобочитаемом виде при помощи языка IL;
- произвольные пользовательские ресурсы.

Из вышеперечисленных элементов только манифест является обязательным, остальные элементы могут включаться по желанию разработчика.

Чаще всего сборки располагаются в одном дисковом файле. Но существует возможность создания многофайловых сборок. Их содержимое будет представлено в виде нескольких дисковых файлов, но, тем не менее, они будут рассматриваться средой исполнения как одна сборка.

Наиболее распространены однофайловые сборки, поскольку их создание более просто и по умолчанию используются в среде разработки Visual Studio .NET. На рис. 4.4 приведено схематичное изображение однофайловой сборки, которая включает в себя все элементы.

Рис. 4.4. Состав однофайловой сборки

Под крышей одной сборки можно объединить несколько файлов, в которых размещены различные элементы сборки (рис. 4.5). Из рисунка видно,

что за пределы сборки можно выносить не только ресурсы, но и код, который будет представлен в виде модулей.

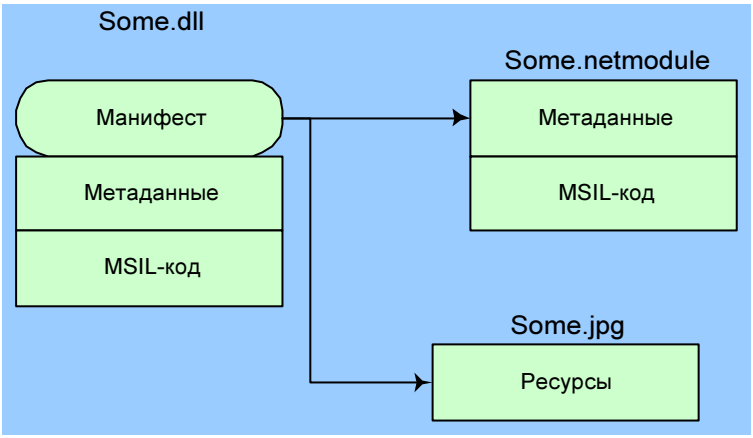


Рис. 4.5. Пример многофайловой сборки

Многофайловые сборки обладают тремя основными преимуществами перед однофайловыми.

1. В состав одной сборки можно включить модули, код которых разработан на различных языках программирования. Более подробно об этом рассказано в *подразделе "Модули" настоящей главы.*
2. При запуске приложения через сеть очень удобно поместить код сборки и ресурсы в разные файлы. Таким образом, файлы будут подгружаться не все сразу, а по мере необходимости, при непосредственном обращении к ним, что существенно снизит трафик при первоначальной загрузке приложения. При повторном запуске приложения этот фактор не будет существенным, поскольку все данные уже будут записаны в кэш.
3. Также удобно выносить ресурсы за пределы сборки во время отладки и разработки приложения, поскольку для их редактирования не придется прибегать к специальным средствам, извлекающим их из сборок. Ресурсы будут доступны в виде обычных дисковых файлов, которые можно будет редактировать привычным образом. При этом, чтобы увидеть внесенные изменения, приложение будет не нужно перекомпилировать.

Однако использование многофайловых сборок гораздо сложнее, чем однофайловых. И для их создания придется прибегнуть к специальным инструментам среды .NET, которые недоступны из среды IDE Visual Studio .NET.

Манифест

Каждая сборка, будь она динамической или статической, обязана иметь манифест, который всесторонне ее описывает. Манифест это что-то вроде личного паспорта каждой сборки. Он отвечает за хранение следующих данных:

- информация о сборке (имя, версия, контрольная сумма, открытый криптографический ключ);
- список файлов, составляющих сборку, в случае, если она является многофайловой;
- список сборок, необходимых для функционирования этой сборки, зачастую называемый списком зависимостей (dependence list);
- связи типов, экспортируемых сборкой, с их внутренней реализацией.

Информацию о манифесте можно просмотреть, используя IL-листинг сборки (он может быть получен при помощи утилиты `ildasm` с параметром `/out:имяфайла.il`).

Манифест, в подавляющем большинстве случаев, располагается в начале кода. Можно использовать утилиту `ildasm` в интерактивном режиме. В этом случае, необходимо открыть ветку `MANIFEST` в главном окне утилиты (рис. 4.6).

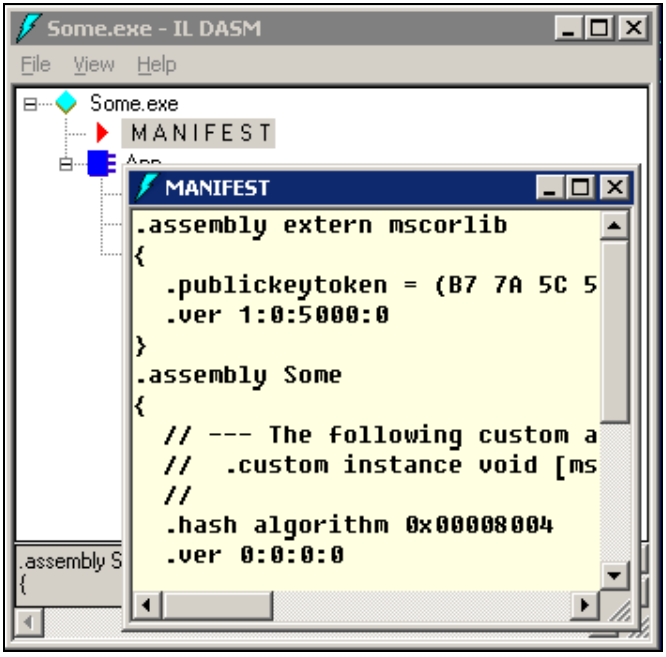


Рис. 4.6. Просмотр манифеста в утилите `ildasm`

Приведем пример манифеста (листинг 4.1).

Листинг 4.1. Пример манифеста

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 1.1.4322.573
//  Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.


.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )             // .z\V.4..
  .ver 1:0:5000:0
}
.assembly Some
{
  // --- The following custom attribute is added automatically, do not
uncomment -------
  //  .custom instance void [mscor-
lib]System.Diagnostics.DebuggableAttribute::.ctor(bool,
  // bool) = (01 00 00 01 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Some.exe
.module extern Some2.netmodule


// MVID: {AE5CD969-464A-4D97-93AB-AFF02E963A76}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
.mresource public prc.nlp
{
  .file prc.nlp
}
```

Директивы манифеста описаны в табл. 4.1.

Таблица 4.1. *Директивы манифеста*

Директива IL	Описание
`.assembly extern <имя сборки>`	Указывает внешнюю сборку, необходимую для работы текущей; в нашем случае это ссылка на FCL (Framework Class Library, общую библиотеку классов), которая преимущественно располагается в сборке `mscorlib`
`.publickeytoken <хеш>`	Определяет маркер открытого ключа подключаемой сборки для ее точной идентификации
`.ver <версия>`	Определяет необходимую версию подключаемой сборки
`.culture`	Указывает региональную принадлежность сборки
`.assembly <имя сборки>`	Указывает имя сборки
`.hash algorithm`	Определяет алгоритм, по которому рассчитывался маркер открытого ключа сборки
`.ver 0:0:0:0`	Определяет версию сборки
`.public`	Открытый ключ, встраиваемый в сборку
`.culture`	Указывает региональную принадлежность сборки
`.module <имя файла>`	Указывает имя основного модуля этой сборки
`.module extern <имя файла>`	Указывает имя внешнего модуля, входящего в сборку, на который есть ссылка в коде этой сборки (модуля)
`.imagebase <адрес>`	Указывает адрес, по которому следует загружать сборку в адресное пространство приложения
`.subsystem <константа>`	Указывает на тип приложения; в нашем случае число 3 указывает на то, что приложение консольное
`.file alignment <512>`	Определяет выравнивание, использовавшееся компилятором при создании этого файла
`.corflags <константа>`	Зарезервированная директива
`.mresource public <название ресурса>`	Указывает на подключение к сборке общедоступного ресурса с заданным именем

Таблица 4.1 *(окончание)*

Директива IL	Описание
`.file <имя файла>`	Определяет имя файла, в котором располагается ресурс
`.file <имя файла>`	Подключает внешний модуль, на который может не быть явных ссылок в коде этой сборки
`.entrypoint`	Указывает на то, что точка входа в приложение будет располагаться в этом модуле

Все эти элементы вкупе называются "метаданными сборки". Такое имя они получили за то, что описывают сборку. В рамках проекта .NET вспомогательную информацию, описывающую свойства чего-либо, принято называть метаданными. Все перечисленные элементы могут задаваться программистом, при помощи специальных программных атрибутов, ключей компилятора (настроек среды IDE Visual Studio) или же специальных утилит командной строки.

В таблицу не попал один элемент манифеста — MVID. Он даже был закрыт символами комментария в тексте IL-кода. Этот элемент достаточно важен, и поэтому рассмотрим его немного подробнее.

MVID расшифровывается как Module Version Identifier (Идентификатор Версии Модуля). Это уникальный 128-битный идентификатор, который автоматически генерируется при каждом построении сборки (модуля) и добавляется в его манифест. Он используется только во внутренних целях среды исполнения. Сами программисты с ним никогда не сталкиваются. Именно поэтому он и был закрыт символами комментария в манифесте. Поскольку он генерируется автоматически, то в языке IL не предусмотрено директивы для его указания.

При каждой загрузке любого модуля, среда исполнения использует MVID для проверки того, не был ли он уже загружен (MVID уникален глобально, поскольку это обычный GUID-идентификатор). Если модуль оказывается уже подгруженным, то среда исполнения просто устанавливает необходимые связи с ним, избегая прямого обращения к файлу, существенно увеличивая скорость доступа к его коду.

Модули

Модули в среде .NET используются для хранения кода. Код может храниться только в модулях, и нигде более. "Но как же так? — скажет внимательный читатель. — В предыдущем разделе говорилось, что код приложений может храниться в сборках".

В большинстве случаев приложения распространяются в виде однофайловых сборок, которые неявно включают модуль, содержащий код. Именно поэтому кажется, что код располагается в сборках. Но вполне возможна ситуация, когда сборка не будет иметь вложенного модуля, а значит, ни байта кода внутри себя. Фактически, сборки используются в среде .NET лишь для объединения модулей и ресурсных файлов в единые сущности, которыми более удобно управлять.

Внимание

Очень важно с самого начала осознать, что сборки это не файлы. Это не более чем абстракция в виде специальных пакетов, предназначенных для хранения и объединения файлов.

Сборки могут иметь версию, уникальную подпись на основе криптографического ключа, а также атрибуты безопасности, чего не имеют модули и файлы ресурсов. Можно предположить, что модули не могут использоваться автономно как самостоятельные единицы. Такое предположение по большей части верно, поскольку отдельное использование модулей возможно лишь вручную, при помощи технологии отражения. В подавляющем большинстве случаев они используются в составе сборок.

Независимо от того, входит модуль в состав основного файла сборки или представлен в виде отдельного дискового файла, он подгружается средой исполнения автоматически и абсолютно не виден извне. Таким образом, использование модулей абсолютно прозрачно для программистов. Даже больше — большинство рядовых программистов, пишущих для платформы .NET, никогда не узнает о модулях, так как их использование настолько скрыто, что самому догадаться об их существовании просто невозможно.

Помимо обычных модулей, по умолчанию входящих в однофайловые сборки, существует возможность создания внешних модулей. Такие модули, чаще всего, подключают к приложению при помощи технологии многофайловых сборок. Проиллюстрируем вышесказанное примером. Сначала создадим сам модуль, код которого затем включим в состав нашего приложения (листинг 4.2).

Листинг 4.2. Пример простейшего модуля

```
/*
  Листинг 4.2
  File:   Mod.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
```

```
// Единственный класс нашего модуля.
class Mod
{
  // Метод, демонстрирующий работу модуля.
  public static void DoIt()
  {
    // Выведем на консоль приветствие.
    Console.WriteLine("Hello World, from Module!");
  }
};
```

Код модуля-примера предельно прост и не содержит ничего лишнего. В нем находится класс `Mod` с единственным методом `DoIt`, который информирует о своем вызове выводом приветствия на консоль. Хотелось бы обратить внимание читателей на особенности компиляции этого листинга.

Примечание

Скомпилировать этот листинг в модуль при помощи IDE Visual Studio .NET, вам не удастся, по той причине, что эта среда не поддерживает работу с модулями и многофайловыми сборками.

Для того чтобы превратить этот листинг в модуль, нам придется воспользоваться компилятором командной строки `csc`, вызвав его при помощи следующей команды.

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

Ключ `/t:module` указывает на то, что из исходного кода, расположенного в файле `Mod.cs`, необходимо создать модуль. Далее необходимо создать многофайловую сборку, которая будет использовать этот модуль. Для начала напишем ее код (листинг 4.3).

Листинг 4.3. Многофайловая сборка, использующая модуль

```
/*
  Листинг 4.3
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Основной класс нашего приложения.
```

```
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Сообщим пользователю о начале работы нашего приложения.
    Console.WriteLine("Enter to main application");
    // Проверим модуль
    TestModule();
  }
  // Здесь состоится обращение к коду модуля.
  public static void TestModule()
  {
    // Вызовем функцию из внешнего модуля.
    Mod.DoIt();
  }
};
```

Использование кода внешних модулей настолько просто, что может показаться, будто класс `Mod` входит в пространство `System` общей библиотеки классов среды .NET. Такой вывод можно сделать из-за того, что в самом листинге никаких упоминаний о внешних модулях вы не найдете. Обращение к коду внешних модулей, входящих в сборку, не требует использования специальных средств и является прозрачным для программиста.

Для того чтобы скомпилировать код в сборку, воспользуемся компилятором `csc`, которому надо указать на необходимость подключения созданного модуля при помощи ключа `/addmodule`.

```
csc /addmodule:Mod.netmodule Some.cs
```

В итоге, получится два файла: `Some.exe` (файл сборки) и `Mod.netmodule` (внешний модуль). Фактически, в созданном приложении присутствует два модуля, хоть это и не является явным. Первый входит в состав файла `Some.exe` и является основным модулем приложения, поскольку содержит точку входа. А второй модуль — внешний и располагается в файле `Mod.netmodule`.

Примечание

Для того чтобы увидеть, как подключается внешний модуль к сборке, можно просмотреть ее манифест одним из способов, описанных в предыдущем подразделе. В нем вы обязательно найдете директиву `.module extern Mod.netmodule`, которая отвечает за его подключение.

В результате работы исполняемого файла приложения (`Some.exe`) на консоль будут выведены следующие строки.

```
Enter to main application
Hello World, from Module!
```

После запуска приложения управление передается функции `Main`, первая строка которой выводит на консоль сообщение, информирующее о запуске приложения. Код, транслируемый компилятором из этой строки, располагается в основном модуле приложения, следовательно, он будет доступен всегда и выполнится в любом случае. Затем произойдет вызов функции `TestModule`, которая уже будет непосредственно обращаться к коду модуля. Перед исполнением этой функции она будет передана JIT-компилятору, который откомпилирует ее из .NET байт-кода в процессорный (машинный) код. При предварительном сканировании кода метода `TestModule`, JIT-компилятор обнаружит, что функция обращается к коду внешнего модуля `Mod.netmodule`. После чего JIT-компилятор отдаст приказ на загрузку этого модуля среде исполнения .NET. Таким образом, модули загружаются непосредственно перед обращением к их коду. Соответственно, если приложение ни разу не обратится к коду внешнего модуля, он не будет загружен, хотя остальной код программы будет выполнен. Это легко проверить, удалив файл `Mod.netmodule` из каталога нашего приложения. На консоль будет выведена лишь первая строка, после чего среда исполнения сообщит о том, что необходимый для дальнейшего исполнения модуль найден не был.

```
Enter to main application
Unhandled Exception: System.IO.FileNotFoundException: File or assembly
name Mod.netmodule, or one of its dependencies, was not f
ound.
File name: "Mod.netmodule"
   at App.TestModule()
   at App.Main()
```

Сканирование кода на предмет использования внешних модулей происходит лишь в рамках одной функции, непосредственно перед ее вызовом. Сканирования всего кода приложения не происходит. Соответственно, обращения к разным модулям лучше помещать в разные функции, а не разбивать условными ветвлениями внутри одной функции. Такой подход повышает вероятность того, что приложение сможет обойтись без загрузки отдельных модулей.

У модулей есть одно очень полезное, но не документированное свойство. Они одновременно могут входить в состав нескольких сборок. То есть, можно создать модуль, поместив в него общий для нескольких сборок участок кода, а затем подключить его ко всем этим сборкам (рис. 4.7).

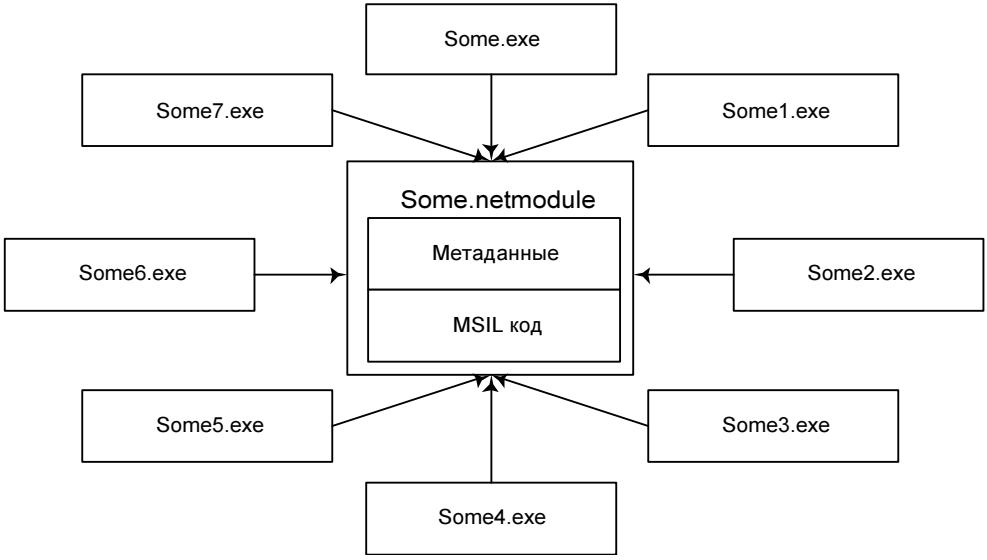


Рис. 4.7. Использование одного модуля несколькими сборками

Это очень удобно, поскольку, изменив код только одного модуля, подвергаются изменениям все, зависимые от него сборки. Такого же результата можно добиться подключением дополнительных сборок. Но использование модулей имеет существенное преимущество. Код, помещенный в модули, входит в состав сборки, которая его подключает, следовательно, будет иметь общие с ней атрибуты безопасности и конфигурационные настройки. Таким образом, код модуля `Some.netmodule` будет автоматически иметь атрибуты и привилегии той сборки, в состав которой он входит, а в случае помещения общего кода в отдельные сборки, такие параметры придется настраивать вручную. Таким образом, обобществление кода между приложениями при помощи модулей является гораздо более удобным методом, чем при использовании сборок.

"Пустые" сборки

Обязательным элементом сборки является лишь манифест, остальные элементы могут присутствовать опционально. Для примера создадим сборку, которая будет отвечать этому условию. Назовем такие сборки "пустыми", и они обычно предназначены для объединения файлов приложения, сами они не содержат ни кода, ни ресурсов.

К сожалению, обычными средствами создать "пустую" сборку и модуль, который будет подключаться при помощи сборки, не удастся. Поэтому придется воспользоваться низкоуровневым языком IL.

Сначала создадим модуль, а затем подключим его к пустой сборке. Он будет кардинально отличаться от обычных модулей, тем, что в нем будет располагаться точка входа приложения. Еще раз обратим ваше внимание — точка входа в приложение будет располагаться не в основном исполняемом файле, а во внешнем модуле! Код такого модуля приведен в листинге 4.4.

Листинг 4.4. Внешний модуль с точкой входа

```
/*
  Листинг  4.4
  File:   Module.il
  Author: Дубовцев Алексей
*/
// Устанавливаем связь с общей библиотекой классов FCL,
// она понадобится нам для вывода текста на консоль.
.assembly extern mscorlib {}
// Определим имя нашего модуля.
.module Module.mod
// Это точка входа в будущую программу,
// она будет располагаться в этом модуле.
.method public static void  EntryFunction()
{
  // Указываем на то, что эта функция будет точкой входа
  // в приложение.
  .entrypoint
  // Выделим в стеке одну ячейку. Позднее туда загрузим
  // ссылку на строку, выводимую на консоль.
  .maxstack  1
  // Помещаем в стек ссылку на строку,
  // которая будет передана в качестве параметра функции,
  // вызываемой далее.
  ldstr      "Hello, World, from module!"
  // Выведем строку на консоль.
  call       void [mscorlib]System.Console::WriteLine(string)
  //Выходим из функции.
  ret
}
```

Имя функции, точки входа в приложение, может быть произвольным, а не только `Main`. Главное, чтобы у функции был указан атрибут `.entrypoint`.

В остальном код модуля тривиален и не содержит ничего сложного. Для того чтобы скомпилировать исходный текст, необходимо воспользоваться компилятором IL-кода со следующими параметрами:

```
ilasm Module.il /DLL /out=Module.mod
```

В результате формируется модуль в виде файла под именем `Module.mod`.

Теперь необходимо создать "пустую" сборку, которая будет использовать этот модуль. Код сборки будет написан также на языке IL, поскольку только в этом языке присутствует возможность непосредственной работы с манифестом. Поскольку наша сборка будет представлять собой один лишь манифест, то язык IL окажется незаменим. Код "пустой" сборки представлен в листинге 4.5.

Листинг 4.5. Пример "пустой" сборки

```
/*
  Листинг 4.5
  File:   Some.il
  Author: Дубовцев Алексей
*/
// Определяем имя сборки.
.assembly Some {}
// Подключаем внешний модуль.
.file Module.mod
  // Указываем компилятору, что точка входа в приложение
  // будет располагаться в этом модуле.
  .entrypoint
```

Из исходного кода создадим сборку при помощи компилятора `ilasm`, используя следующую командную строку.

```
ilasm Some.il
```

Дополнительных параметров, кроме имени файла, указывать не надо. Результатом компиляции программы будет создание файла `Some.exe`. Это "пустая" сборка, являющаяся исполняемым файлом приложения. В результате ее запуска на консоль будет выведена строка.

```
Hello, World, from module!
```

Необходимо отметить факт, что в самом файле `Some.exe` не содержится ни байта кода. Если внешний модуль, в котором располагается код приложения, окажется недоступным, то среда исполнения сообщит об ошибке выводом диалогового окна (рис. 4.8).

Рис. 4.8. Среда исполнения не может найти точку входа в управляемое приложение

Функция `_CorExeMain` является внутренней точкой входа в любое приложение .NET.

Таким образом, "пустые" сборки используются в качестве связующего звена между реальными элементами сборки. Их целесообразно применять для поставки приложений в минимальной конфигурации, что очень удобно в локальных сетях с большим количеством пользователей. Можно поместить код и файлы ресурсов в сети. А само приложение поставлять в виде маленькой "пустой" сборки и конфигурационного файла, необходимого для указания местоположения остальных частей приложения в сети.

Сборки со строгими именами

Структура строго именованных сборок

Сборки могут использоваться в двух режимах: персональном и совместном. Персональный режим предполагает размещение сборок в одном каталоге с приложением. Связывание с ними будет осуществляться лишь при помощи их символьных имен. Поскольку сборка будет доступна только этому приложению, то конфликт имен или коллизии другого рода практически исключены.

В совместном режиме сборки помещаются в глобальное хранилище (GAC, Global Assembly Cache), из которого они доступны любым приложениям. Здесь уже будет недостаточно использования только имени для связывания со сборкой, необходим более устойчивый механизм подключения, который бы гарантировал уникальность и точность связывания. С этой целью для сборок были введены строгие имена. Помимо обычного символьного имени, версии и региональной информации, строгое имя требует подписание сборки открытым криптографическим ключом, что гарантирует полную уникальность ее имени (идентификации).

Примечание

Обратите внимание на то, что сборка содержит открытый ключ и строгое имя — маркер открытого ключа. Это два разных понятия, которые будут раскрыты далее.

Само строгое имя состоит из символьного имени сборки, версии, региональной информации, а также маркера открытого ключа. Примеры строгих имен приведены ниже.

```
Some, Version=1.2.0.0, Culture=neutral, PublicKeyToken=672eb6fe2413d741
Some, Version=1.2.0.0, Culture=ru-RU, PublicKeyToken=672eb6fe2413d741
Some, Version=1.2.0.0, Culture=ru-RU, PublicKeyToken=324cf587987a87dc
```

Обратите внимание на то, что при совпадении символьных имен, полные имена могут не совпадать из-за различий региональных идентификаторов, номеров версий, а также маркеров открытых ключей.

Примечание

Маркер открытого ключа представляет собой 64-битное число, которое является хешем (контрольной суммой) открытого ключа. Маркер был введен для того, чтобы сократить запись строгого имени сборки. С маркером ключа работать гораздо удобнее, поскольку он намного меньше, чем открытый ключ.

Правда, могут возникнуть сомнения по поводу надежности маркера, по сравнению с реальным ключом. Но по этому поводу можно не беспокоиться, маркер по умолчанию рассчитывается при помощи алгоритма SHA-1 (Secure Hash Algorithm), возможностей которого с лихвой хватает для обычных приложений.

Если вы хотите подробнее ознакомиться с теорией открытых ключей, хешами, а также другими криптографическими алгоритмами, то вам стоит прочитать книгу Брюса Шнайдера "Прикладная криптография".

В самой сборке помещен открытый ключ, а не маркер. А ссылка на сборку со строгим именем производится именно по маркеру открытого ключа. Убедиться в этом можно, обратившись к манифесту любого файла, использующего сборку со строгим именем. Примером такой сборки является `mscorlib` — основная сборка FCL (общей библиотеки классов). Запись о связи с ней присутствует в манифесте практически любой сборки и выглядит следующим образом.

```
.assembly extern mscorlib
{
   // Это маркер открытого ключа сборки.
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                 // .z\V.4..
  .ver 1:0:5000:0
}
```

Выбор маркера, а не полного ключа, для связи со сборкой, продиктован требованиями уменьшения размера конечного файла сборки. Сборка может ссылаться на множество других строгих сборок, поэтому связывание при помощи полных открытых ключей было бы весьма накладно. Размер открытого ключа превышает один Кбайт. Следовательно, если использовать для

связи со сборкой полный открытый ключ, то с каждой подключаемой сборкой размер файла будет увеличиваться более чем на Кбайт.

При запросе на загрузку сборки со строгим именем, загрузчик среды исполнения ищет ее, основываясь на символьном имени, версии, региональной информации, а также маркере открытого ключа, который обязательно должен совпадать с маркером (хешем) открытого ключа подключаемой сборки.

Примечание

Опасаться того, что расчет маркеров (читай: контрольных сумм) открытых ключей при поиске необходимых сборок может замедлить процесс загрузки приложения, не стоит. Маркеры для всех сборок, располагающихся в глобальном хранилище, рассчитываются заранее. Таким образом, при загрузке необходимо лишь сравнить два восьмибайтных числа.

Такой подход полностью исключает возможность использования чужой сборки, вместо необходимой. Если, конечно, разные разработчики не будут пользоваться одной и той же парой ключей при создании своих сборок. Случайное создание идентичной пары ключей полностью исключено!

Помимо участия в строгом имени сборки, открытый ключ используется также для ее подписания. При создании сборки компилятор рассчитывает контрольные суммы всех файлов, входящих в сборку, затем формирует базовый файл сборки, в манифесте которого будут указаны рассчитанные значения и полный открытый ключ сборки. После чего вычисляет контрольную сумму базового файла итоговой сборки, которую, при помощи пары ключей (открытого и закрытого), преобразует в цифровую подпись RSA. Эта подпись помещается в специальную секцию PE файла, не участвующую при подсчете общей контрольной суммы, что позволяет провести процесс в обратном порядке. То есть подсчитать контрольную сумму сборки и при помощи сохраненной цифровой подписи и открытого ключа проверить целостность файла.

Такой механизм позволяет предотвратить несанкционированное изменение сборки — ее кода или любых, входящих в ее состав файлов.

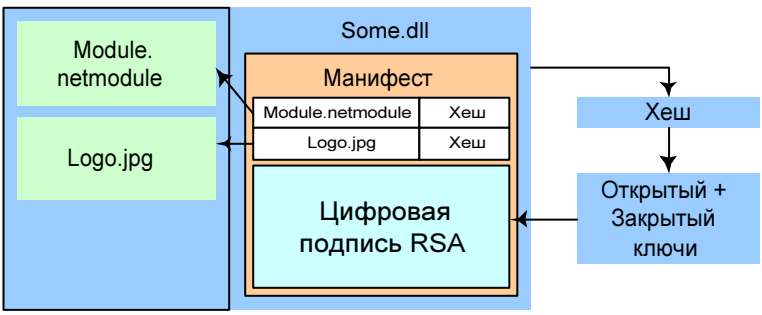


Рис. 4.9. Использование цифровой подписи для защиты сборки

Создание строго именованных сборок

Сначала необходимо создать пару криптографических ключей, которыми будет подписываться сборка. Ключи создаются с помощью утилиты `sn.exe`, с использованием параметра `-k`.

```
sn -k keys.snk
```

В результате выполнения команды в текущем каталоге будет создан файл, в котором располагается пара криптографических ключей: открытый и закрытый. При помощи все той же утилиты `sn` можно просмотреть открытый ключ и узнать его маркер.

```
sn -tp keys.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

Public key is
0702000000240000525341320004000001000100
0834b28783aa71a3b1f6188a895c51051989aca
bf8a340953dca9d3e1ff6e84573ad49cfc3f494
9cf6deb0e1cc6c907e5fad12f245eaef3a1d026
1ea78e8ea15095d256f700031d063316a428d05
182874924a31123b8f536264c4924b3f17c6471
44ed9a13c149fd5bdb90c1cfaa175f098c38a5d
9beb7af662578b76c8d69d7c309a0de521db83a
cd547a432ddc8c8d3ab0140352761e7321f39cd
b43e79e5fff2337bf5041dcfb5d0aeca4db7241
b00346bbd161603fa5759c5c84d00a0df441db6
453c5fce7878b7a911135345d687fa6361b4bac
a491d879223ad1804bfca7908cee00eeb9c190c
3d631368be9a3b90e35cd01c160485fea3e2805
8df5e101697c6e65ad1ca575e2a183a93a57644
f8df5d3671f6ac2726a2c8ee5079e704714dabe
ebbb36d49b4f59debdb9eb82de515fed1c285ef
df84f1fc6794753bbc17b46c58eb7c7ff9436ca
3d9ccb00bfe93bff46b80b589a7e39012e96fd7
1f8e60e2ca94e8918e9ee53887e75991c1bdfc0
171ad759c39b9f11e96ce17c1e722a13c26967a
e9caefb8f913a3f8fabb18f3585b2396de5dc76
3594f027e3915d7ee8a0779226c22dfc59261b2
d54b1195def8a76be314748698146841cb76c3b
```

```
0193b3aeede8ae48bdb4d6abab0d64d87ae1723
f043fa90411c637189e54e323cfcc536b7d3aa4
9344858efe293b29255e4961b2b633e711f74e9
af2f2b1189c2857e8b311d569e263bc6a58ef3c
ac91c6a0c914d2adcfdbf0bf8d5fe2b1526ba14
11fcd72b3619f084ae2348e4d0ec3fc224a6185
213d6c92d36e5b4e01c844

Public key token is 60582cc2df93bf17
```

Если параметр `-tp` заменить параметром `-t`, то на консоль будет выведен лишь маркер открытого ключа.

Предупреждение

Помните, что закрытый ключ должен быть доступен только вам. Поскольку, попав в чужие руки, он может быть использован для подписания сборки от вашего имени и позволит выдавать себя посторонним лицам за вас.

Располагая парой ключей, можно перейти к созданию строго именованной сборки. Для этого, при помощи атрибута `AssemblyKeyFile`, укажем компилятору, что сборку необходимо подписать при помощи пары ключей, располагающихся в файле `Keys.snk`. В исходном коде применение этого атрибута будет выглядеть следующим образом.

```
// В общем виде.
[assembly: AssemblyKeyFile("Имя файла с парой ключей")]
// В нашем случае.
[assembly: AssemblyKeyFile("Keys.snk")]
```

При автоматическом создании проекта в среде разработки VisualStudio .NET этот атрибут помещается в файл `AssemblyInfo.cs`. Фактически, местоположение атрибута принципиального значения не имеет.

Получившаяся строго именованная сборка, по сути дела, будет являться библиотекой, предоставляющей некий сервис, доступный любым приложениям. Код сборки представлен на листинге 4.6.

Листинг 4.6. Пример строго именованной сборки

```
/*
  Листинг 4.6
  File:   Strong.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
```

```
using System;
// В этом пространстве имен находится атрибут AssemblyKeyFile,
// используемый для указания имени файла с криптографическими ключами.
using System.Reflection;
// В этом пространстве имен находится класс MessageBox, позволяющий
// вывести на экран диалоговое окно.
using System.Windows.Forms;
// Зададим имя файла с парой криптографических ключей, которые будут
// использованы для подписания сборки.
[assembly: AssemblyKeyFile("TeskKey.snk")]
// Это пространство имен отделит типы нашей сборки от остальных.
namespace nmXTest
{
  // Тестовый класс.
  public class XTest
  {
    // Функция, сообщающая пользователю о своем вызове при помощи
    // диалогового окна.
    public static void Test()
    {
      MessageBox.Show("Hello, World, from test strong named assembly!");
    }
  };
}
```

Скомпилировать этот файл нужно при помощи следующей команды.

```
csc /target:library Some.cs
```

Параметр команды `/target:library` указывает на необходимость создания библиотеки. То есть классы этой сборки можно будет использовать из других приложений. В результате компиляции формируется файл `Strong.dll`. Это и есть строго именованная однофайловая сборка.

В использовании атрибута `AssemblyKeyFile` есть одна особенность — он обязательно должен располагаться в глобальном пространстве имен сборки. Для того чтобы убедиться в этом, будем действовать от противного. Поместим его в пространство имен `nmXTest` и попробуем откомпилировать сборку. Компилятор обязательно выдаст сообщение об ошибке.

```
Strong.cs(28,1): error CS0657: 'assembly' is not a valid attribute loca-
tion for this declaration. Valid attribute locations for
        this declaration are 'type'
```

Других особых требований на использование этого атрибута не накладывается.

Для использования полученной сборки в совместном режиме, необходимо поместить ее в глобальное хранилище сборок. Что такое глобальное хранилище сборок, как работать с ним, рассмотрим в следующем разделе.

4.3. Глобальное хранилище сборок (GAC)

Аббревиатура GAC расшифровывается как Global Assembly Cache, что в дословном переводе на русский язык означает: Глобальный Кэш Сборок. В книге будет использоваться термин хранилище, вместо термина кэш. Он более точно отвечает реальному предназначению GAC, так как кэш — это все-таки не постоянное, а временное место хранения, используемое для оптимизации доступа к данным. Для примера можно привести дисковый или процессорный кэш.

GAC предназначен для постоянного хранения совместно используемых сборок. В его задачу входит обеспечение безопасного хранения строго именованных сборок, а также предотвращение коллизий при их использовании.

С этой задачей GAC справляется на удивление элегантно. При всей красоте своей архитектуры, GAC на сегодняшний день является наиболее мощным подходом к хранению совместно используемого кода.

Сам GAC представляет собой группу обычных файловых каталогов, расположенных на диске компьютера по следующему адресу.

```
%WINDIR%\assembly\
```

При просмотре содержимого этого каталога с помощью проводника (Explorer), можно увидеть список всех проинсталлированных строго именованных сборок. Пример экрана представлен на рис. 4.10.

Но знайте, этот список не соответствует реальному строению GAC. Он формируется при помощи расширения оболочки проводника Windows.

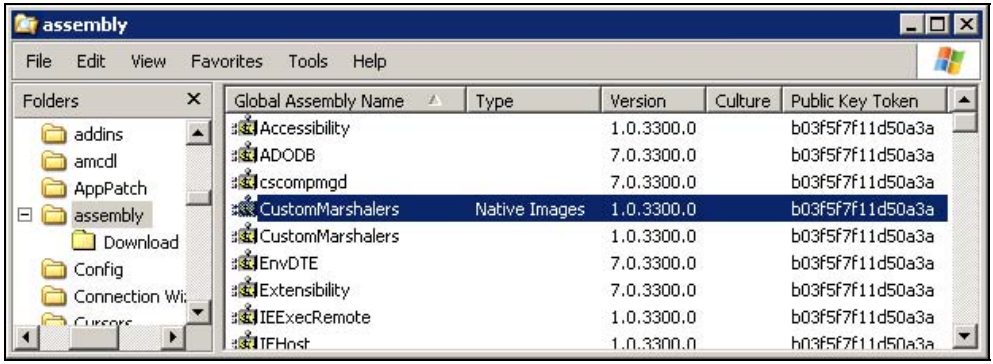


Рис. 4.10. Расширение оболочки, предназначенное для просмотра GAC

Расширение оболочки для управления сборками

Расширение оболочки GAC предназначено для облегчения процесса управления сборками простыми пользователями, а также администраторами среды исполнения .NET. Работая с оболочкой, нет необходимости знать реальную архитектуру GAC. При открытии папки `assembly`, расширение оболочки автоматически собирает информацию о всех установленных сборках и выводит их в удобном для просмотра виде.

Список состоит из четырех колонок, которые предоставляют практически всю необходимую информацию о каждой из сборок.

- `Global Assembly Name` — основное символьное имя сборки, совпадающее с именем ее файла без расширения (к примеру, у сборки `Strong` может быть имя файла `Strong.dll` или `Strong.exe`).
- `Type` — в этой колонке для сборок, откомпилированных в код платформы, указана строка `Native Images`.
- `Version` — в этой колонке указана версия сборки.
- `Culture` — региональная зависимость сборки.
- `Public Key Token` — маркер открытого ключа.

Если информации окажется недостаточно, то можно щелкнуть правой кнопкой мыши на интересующей сборке и из контекстного меню выбрать пункт **Properties**.

На вкладке **General** расположено поле **CodeBase**, которое указывает на место, откуда производилась ее установка в GAC. В случае использования обычных сборок, это поле не представляет интереса. А если сборка перед использованием скачивалась из сети, можно узнать точный URL-адрес ее базового местоположения.

В левой часть дерева каталогов, под папкой `assembly` находится папка `Download`, в которой отображается содержимое кэша закачанных из сети сборок. Фактически, эта папка ничего общего с GAC не имеет, поскольку хранится отдельно от него, в учетных записях, персональных для каждого пользователя. При изучении этой папки и пригодится поле **CodeBase** закладки **General**, поскольку позволит узнать, откуда были скачаны сборки. Скорее всего, на вашем компьютере папка Download будет пустовать. Удивляться этому не стоит, так как до сих пор использование возможности хранения сборок в сети используется мало. Происходит это по двум причинам: во-первых, об этой возможности мало кто знает, во-вторых, распределенные сетевые приложения пока не пользуются большой популярностью.

На закладке **Version**, вы сможете узнать исчерпывающую информацию о версии продукта, включающую: название компании, внутреннее имя, региональную информацию, базовое имя файла, полное имя продукта и его версию.

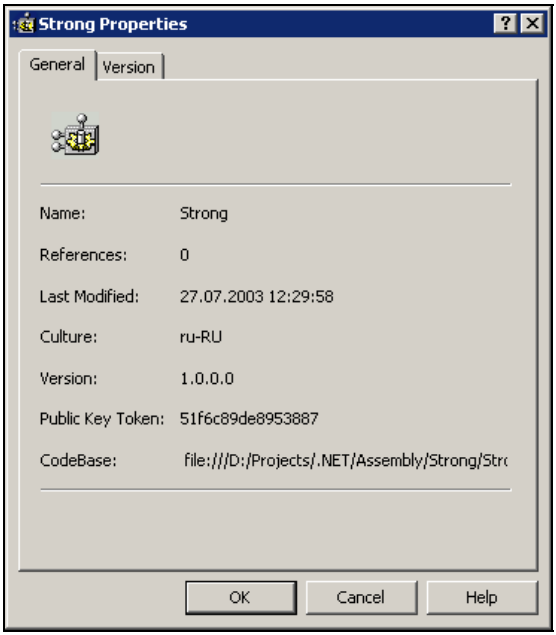


Рис. 4.11. Вкладка **General** окна **Properties** информации о строго именованной сборке

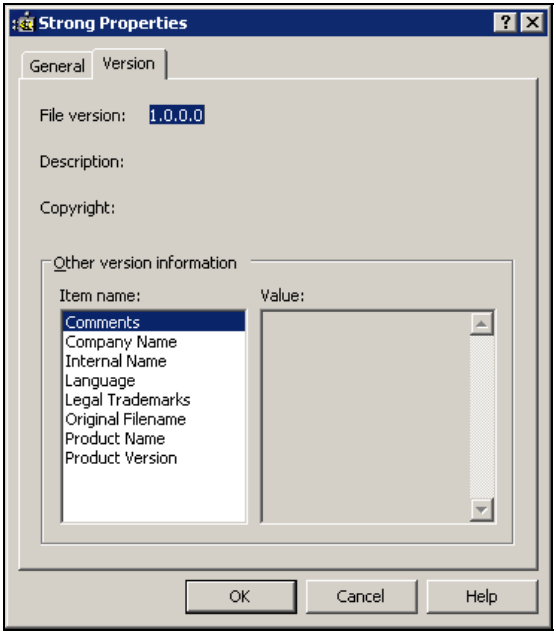


Рис. 4.12. Вкладка **Version** окна **Properties** информации о строго именованной сборке

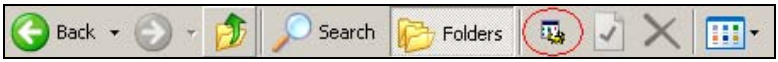


Рис. 4.13. Панель инструментов проводника при просмотре глобального хранилища сборок

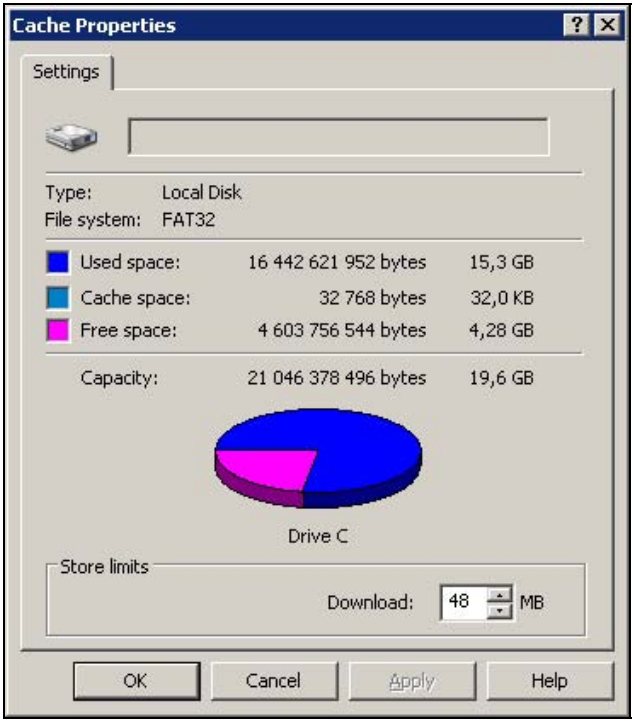


Рис. 4.14. Окно свойств глобального хранилища сборок

Расширение оболочки добавляет на панель инструментов малоприметную кнопку, которая связана с пунктом меню **Tools->Cashe Settings** (рис. 4.13).

Выбрав пункт меню **Tools->Cashe Settings** или нажав кнопку панели инструментов, вы откроете окно, при помощи которого можно просмотреть общую статистику о глобальном хранилище сборок (рис. 4.14).

В поле **Cache space** отражен общий размер скачанных из сети сборок, в нижней части окна, при помощи переключателя **Download**, можно установить максимальный размер хранилища скачиваемых сборок.

Напоследок можно отметить, что расширение оболочки представляет собой COM компонент, который располагается в файле `Shfusion.dll`.

Реальное строение GAC

Изучить реальную структуру GAC можно несколькими способами.

- При помощи проводника, отключив расширение оболочки.
- При помощи стандартных средств командной строки (`cd`, `dir`).
- При помощи файловых менеджеров сторонних производителей, не поддерживающих расширение оболочки проводника.

Расширение оболочки подключается к проводнику при помощи файла `Desktop.ini`, располагающегося в каталоге `assembly`. Найдя этот файл в каталоге, проводник автоматически вызывает расширение оболочки. Следовательно, убрав или переименовав этот файл, можно отключить расширение. Для этого следует выполнить следующие три команды.

```
// Переходим в каталог GAC.
cd %windir%\assembly
// Снимаем атрибуты "системный" и "скрытый" с файла Desktop.ini.
attrib -s -h desktop.ini
// Переименовываем файл для отключения расширения оболочки.
ren Desktop.ini Desktop.ini.disabled
```

После отключения расширения структура каталогов GAC будет отображаться проводником в истинном виде (рис. 4.15).

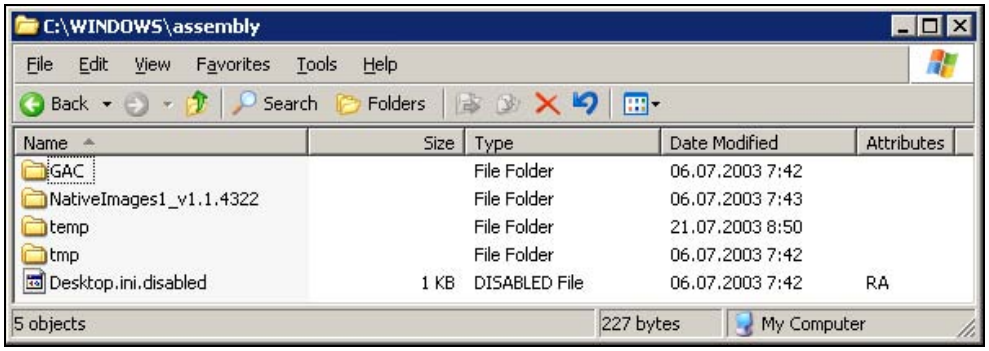


Рис. 4.15. Реальное строение GAC без расширения оболочки

Если понадобится снова включить расширение оболочки, необходимо присвоить файлу прежнее имя. При этом следует учесть, что Windows может автоматически создать файл `Desktop.ini`, который сначала надо будет удалить. Описанные операции можно выполнить при помощи следующих команд.

```
// Снимаем с файла атрибуты.
attrib -s -h -r desktop.ini
```

```
// Удаляем созданный Windows файл.
del desktop.ini
// Восстанавливаем исходное имя файла.
ren desktop.ini.disabled desktop.ini
```

В каталоге `assembly` располагаются четыре вложенные папки.

- `GAC` — это и есть главное хранилище сборок, здесь располагаются совместно используемые сборки.
- `Native1_v1.1.4322` — сюда помещаются версии сборок, откомпилированные в машинный код. У каждой среды исполнения имеется своя версия JIT-компилятора, который производит транслирование сборок в машинный код. Следовательно, для разных версий виртуальной машины результаты компиляции в машинный код могут различаться. Для того чтобы разделить сборки, откомпилированные разными JIT, в конце имени этого каталога указывается версия среды исполнения (JIT). Структура каталога совпадает со структурой каталога `GAC`.

`temp` и `tmp` — здесь располагаются временные файлы, используемые средой исполнения. Особого интереса они не представляют.

Структура папок `GAC` разделена на два вложенных уровня. На верхнем уровне представлены папки, отвечающие символьным именам сборок. Ниже приведен фрагмент содержимого каталога `GAC`, полученного при помощи команды `dir`.

```
C:\dir %windir%\assembly\GAC
 Volume in drive C is HELLO
 Volume Serial Number is A0FA-F69F

 Directory of C:\WINDOWS\assembly\GAC
<DIR>          .
<DIR>          ..
<DIR>          Microsoft.Vsa.Vb.CodeDOMProcessor
<DIR>          Microsoft_VsaVb
<DIR>          Microsoft.Vsa
. . .
<DIR>          Microsoft.VisualStudio.VCProject
<DIR>          stdole
<DIR>          Microsoft.VisualStudio.VSHelp
<DIR>          VSLangProj
<DIR>          Strong
               0 File(s)              0 bytes
              81 Dir(s)     971 382 784 bytes free
```

Возникает вопрос, а что происходит со сборками, имеющими одинаковые символьные имена. Не произойдет ли конфликта при их размещении в папках. Для того чтобы ответить на этот вопрос, необходимо спуститься на один уровень ниже и заглянуть в одну из папок, перечисленных ранее. Для этих целей выберем папки `System` и `Strong` и рассмотрим их содержимое.

```
C:\>dir %windir%\assembly\gac\system
 Volume in drive C is HELLO
 Volume Serial Number is A0FA-F69F
 Directory of C:\WINDOWS\assembly\gac\system
06.07.2003  07:43    <DIR>          .
06.07.2003  07:43    <DIR>          ..
06.07.2003  07:43    <DIR>          1.0.5000.0__b77a5c561934e089
               0 File(s)              0 bytes
               3 Dir(s)     971 296 768 bytes free
C:\>dir %windir%\assembly\gac\strong
 Volume in drive C is HELLO
 Volume Serial Number is A0FA-F69F
 Directory of C:\WINDOWS\assembly\gac\strong
27.07.2003  12:29    <DIR>          .
27.07.2003  12:29    <DIR>          ..
27.07.2003  12:29    <DIR>          1.0.0.0_ru-RU_51f6c89de8953887
               0 File(s)              0 bytes
               3 Dir(s)     971 296 768 bytes free
```

На этом уровне происходит более детализированное разделение хранимых сборок. Здесь каталоги уже содержат информацию о версии, региональной принадлежности, а также маркер открытого ключа. Имена каталогов на этом уровне формируются по такому правилу:

```
ВерсияСборки_РегиональныйИдентификатор_МаркерОткрытогоКлюча.
```

Вся эта информация гарантирует уникальность конечного пути хранения сборки и исключает возможные конфликты. Внутри этих папок уже непосредственно содержатся искомые сборки.

```
C:\>dir %windir%\assembly\gac\strong\1.0.0.0_ru-RU_51f6c89de8953887
 Volume in drive C is HELLO
 Volume Serial Number is A0FA-F69F
 Directory of C:\WINDOWS\assembly\gac\strong\1.0.0.0_ru-
RU_51f6c89de8953887
27.07.2003  12:29    <DIR>          .
27.07.2003  12:29    <DIR>          ..
27.07.2003  12:29             3 584 Strong.dll
```

```
27.07.2003  13:06               212 __AssemblyInfo__.ini
               2 File(s)          3 796 bytes
               2 Dir(s)     970 403 840 bytes free
```

Помимо самой сборки, в этом каталоге располагается файл `__AssemblyInfo__.ini`. Он содержит общую информацию о строгом имени сборки, а также о начальном местоположении. Этот файл используется средой исполнения в целях оптимизации загрузки, а также расширением оболочки для хранения информации поля CodeBase.

Из сказанного можно заключить, что GAC является самоописываемой файловой структурой. Вся необходимая информация заключена в именах папок. Дополнительные источники хранения информации, вроде реестра или каталогов Active Directory, не используются. При всей мощи хранилища GAC, его элегантная простота является революционной, ничего подобного ранее не существовало!

Инсталляция сборок в GAC

Всего существует четыре способа инсталляции сборок в GAC:

- при помощи Windows Installer (MSI) начиная со второй версии (используется для развертывания сборок на компьютерах пользователей);
- при помощи утилиты `gacutil.exe`;
- при помощи расширения оболочки проводника;
- вручную, непосредственно создав нужные каталоги и скопировав туда файлы сборок.

Но все они в итоге сводятся к созданию необходимых подкаталогов в каталоге %windir%\assembly\GAC и копированию в них файлов.

Наиболее простой способ инсталляции сборок — использовать расширение оболочки проводника. Для инсталляции необходимо перетащить значок устанавливаемой строго именованной сборки в каталог %windir%\assembly. Для удаления ее из GAC, необходимо щелкнуть правой кнопкой мыши на значке необходимой сборки и выбрать из контекстного меню пункт **Delete**.

Программистам и разработчикам Microsoft предлагает использовать утилиту `gacutil.exe`. Для работы с ней необходимо уметь пользоваться двумя основными ключами.

`/I <ИмяФайлаСборки> — Инсталлирует сборку в GAC.`

`/U <СтрогоеИмяСборки> — Удаляет сборку с заданным строгим именем из GAC.`

`/U <СимвольноеИмяСборки> — Удаляет все сборки с данным символьным именем из GAC.`

Для того чтобы протестировать работу этой утилиты, инсталлируем в GAC сборку `Strong.dll`, созданную в предыдущем разделе. Для этого воспользуемся следующей командой.

```
gacutil /i Strong.dll
```

Если все было сделано правильно, утилита должна сообщить об успехе инсталляции сборки при помощи следующего сообщения.

```
Assembly successfully added to the cache
```

Прокомментируем работу ключа `/U`. Если в качестве параметра ключа указать только символьное имя сборки, то все сборки с таким символьным именем, независимо от других параметров строгого имени, будут удалены. Фактически будет удален каталог %windir%\assembly\GAC\заданное_символьное_имя_сборки и все его содержимое. Подобного режима использования ключа стоит избегать, так как в этом каталоге могут оказаться сборки других производителей. Ведь совпадения на уровне символьных имен сборок вполне допускаются. Более предпочтительно использовать полное строгое имя сборки такого вида.

```
Strong,Version=1.0.0.0,Culture=ru-RU,PublicKeyToken=51f6c89de8953887
```

В этом случае будет удалена только необходимая сборка. И вы будете полностью застрахованы от нечаянного удаления чужих сборок.

Рассмотрим пример инсталляции сборки `Strong` вручную. Сначала создадим папку на верхнем уровне дерева каталогов GAC, соответствующую символьному имени сборки, то есть `Strong`. Сделать это можно при помощи любого стороннего файлового менеджера или даже при помощи самого проводника, отключив расширение оболочки. В нашем примере, для простоты и наглядности, будут использованы утилиты командной строки.

Для создания папки `Strong` необходимо выполнить следующую команду.

```
mkdir %windir%\assembly\GAC\Strong
```

Далее необходимо создать каталог второго уровня, который будет содержать в своем имени более детальную информацию о сборке. Для этого нужно знать ее версию, региональную информацию, а также маркер открытого ключа.

Примечание

Маркер открытого ключа произвольной сборки можно узнать при помощи утилиты `sn.exe`, запустив ее со следующими параметрами:

```
sn.exe -t <ИмяФайлаСборки>.
```

Каталогу необходимо присвоить имя по следующему правилу: `ВерсияСборки_РегиональныйИдентификатор_МаркерОткрытогоКлюча`. Это будет выглядеть следующим образом.

```
mkdir %windir%\assembly\GAC\Strong\1.0.0.0_ru-RU_51f6c89de8953887\
```

Далее необходимо скопировать в созданный каталог нашу сборку.

```
copy Strong.dll %windir%\assembly\gac\Strong\1.0.0.0_ru-RU_51f6c89de8953887
```

Помимо самой сборки, во всех каталогах, созданных стандартными средствами, содержится файл __AssemblyInfo__.ini. Он используется только для оптимизации поиска информации. Если среда исполнения его не найдет, то информация будет загружена напрямую из сборки. Этот файл создавать не будем, все будет работать и без него.

Деинсталляция сборки вручную производится простым удалением соответствующих каталогов и их содержимого.

Таким образом, инсталляция приложений .NET сводится лишь к простому копированию файлов, что делает систему крайне устойчивой и надежной.

Проверка целостности сборки при инсталляции в GAC

Любая сборка, инсталлируемая в GAC, обязана иметь цифровую RSA-подпись, удостоверяющую ее целостность. Если после последней компиляции сборки ее содержимое случайно или преднамеренно было изменено, то проверка подписи сборки перед ее инсталляцией не пройдет. Это легко показать на примере. Возьмем уже созданную и подписанную сборку Strong.dll. Дизассемблируем ее при помощи утилиты ildasm.exe, используя следующую команду.

```
Ildasm.exe Strong.dll /out:Strong.il
```

Изменим содержимое сборки, поменяв строку, выводимую этой сборкой на консоль.

```
Было
IL_0000:  ldstr      "Hello, World, from test strong named assembly!"
Стало
IL_0000:  ldstr      "He-he, I try to change you!"
```

После чего скомпилируем сборку при помощи этой команды, предварительно удалив настоящую сборку Strong.dll

```
ilasm.exe /dll Strong.il
```

В результате должна получиться сборка Strong.dll, весьма похожая на настоящую, за исключением цифровой подписи RSA. Эта сборка просто не могла быть подписана, поскольку для этого требуется закрытая часть криптографического ключа, которого у нас не было.

Попытаемся проинсталлировать сборку в GAC.

```
gacutil.exe /i Strong.dll
```

Операция не будет исполнена, поскольку не пройдет процедура проверки цифровой подписи. Сообщение об этом событии выдаст утилита gacutil.

```
Failure adding assembly to the cache: Strong name signature could not be
verified.  Was the assembly built delay-signed?
```

Использование строго именованных сборок

Этот подраздел будет завершающим в теме строго именованных сборок. Мы научимся использовать сборки со строгим именем в совместном режиме. Будет показан наиболее простой способ использования строго именованных сборок — при помощи среды разработки Visual Studio .NET. После этого, для окончательного закрепления материала, то же самое будет проделано при помощи утилит командной строки.

Внимание

Здесь, в качестве совместной, будет использована сборка `Strong`, созданная ранее. Прежде чем создавать приложение, использующее совместную сборку, убедитесь в том, что она проинсталлирована в GAC. Лучше сделать это заранее, еще до начала создания проекта.

Во-первых, нам понадобится приложение, к которому мы будем подключать строго именованную сборку. Создадим простое оконное приложение. Сделать это можно при помощи мастера, который доступен из меню: **File->New Project**.

После создания проекта необходимо добавить ссылку на сборку, которая будет использоваться в нем. Для этого необходимо воспользоваться окном **Add Reference**, которое может быть вызвано из основного меню **Project->Add Reference** или из контекстного меню элемента **Reference**, закладки **Solution Explorer** (рис. 4.16).

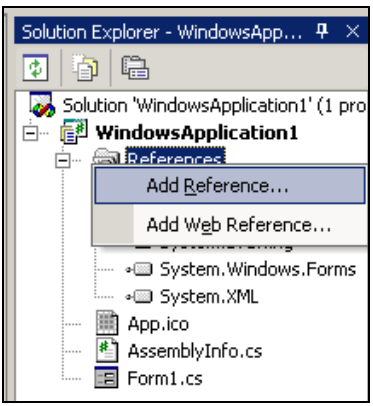


Рис. 4.16. Добавление ссылки на сборку

В окне **Add Reference** необходимо выбрать вкладку **.NET** (рис. 4.17). На ней расположен список, в котором перечислено множество сборок. Среди них должна находиться сборка `Strong`, проинсталлированная в GAC. Однако ее там не окажется. Потому, что на самом деле в этом списке перечислены сборки не из GAC, а расположенные в специальных служебных каталогах. Конкретное местоположение этих сборок указано в колонке **Path**.

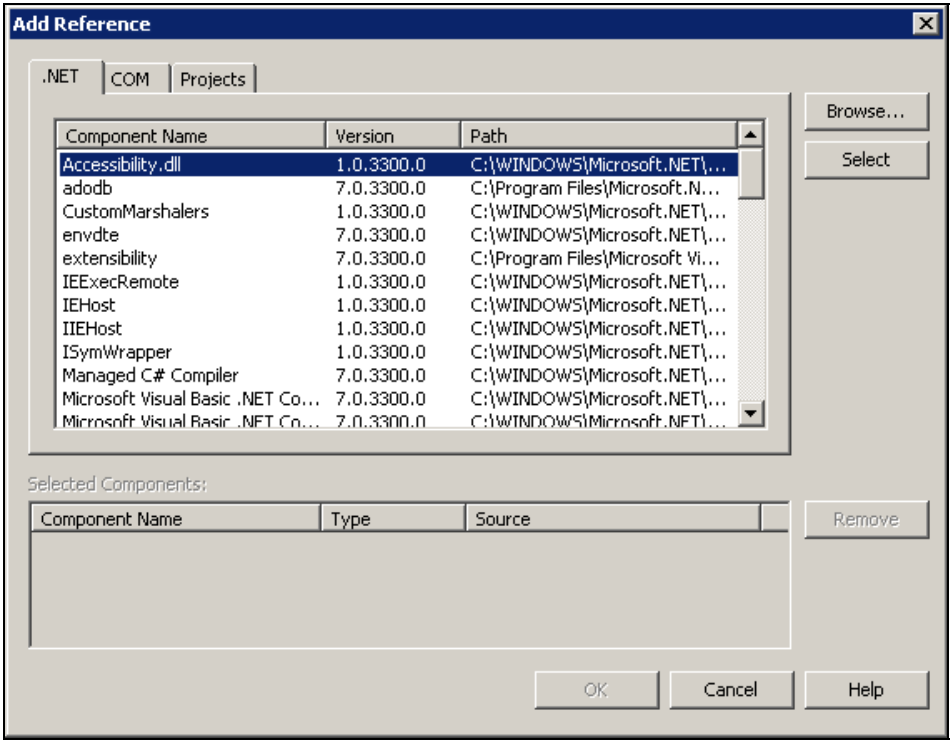


Рис. 4.17. Выбор сборки для установления связей

Для того чтобы добавить связь со сборкой, нужно воспользоваться кнопкой **Browse**, расположенной в правой части окна.

Примечание

Если необходимо часто подключать сборки к проекту, то пользоваться кнопкой **Browse** крайне неудобно. Можно заставить IDE Visual Studio .NET отображать в списке сборки из указанного вами каталога. Для этого необходимо добавить в ветвь реестра HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\..NETFramework\AssemblyFolders подветвь с любым именем и со значением по умолчанию, соответствующим необходимому каталогу. Вы можете сделать это, модифицировав файл реестра (листинг 4.7).

Листинг 4.7. Файл реестра, добавляющий пользовательский каталог в окно добавления сборок

```
REGEDIT4
;
;       Листинг 4.7
```

```
;       File:   Exe.reg
;       Author: Дубовцев Алексей
;
; После запуска сценария, Visual Studio.NET начнет отображать
; сборки из каталога c:\MyAssemlby.
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\AssemblyFolders\MyAsm]
@="C:\\MyAssembly"
```

Здесь необходимо пояснить суть операции добавления связи со сборкой в среде Visual Studio .NET. При разработке приложения сборка необходима на двух этапах:

- при редактировании исходного кода приложения среда разработки Visual Studio .NET использует метаданные сборки в служебных целях;
- при компилировании приложения метаданные сборки необходимы компилятору.

Но при этом вовсе не обязательно, чтобы сборка была инсталлирована в GAC. Такое требование необходимо лишь на стадии запуска приложения. Самой же Visual Studio .NET можно указать и копию сборки, лишь бы метаданные и версии сборок совпадали. Хотя, если вы заранее поместите сборку в GAC, то в дальнейшем избежите многих проблем.

После добавления ссылки на сборку, можно будет исследовать ее содержимое при помощи Object Browser, который можно вызвать при помощи меню **View->Other Windows->Object Browser**. В дереве объектов серыми прямоугольниками обозначены сборки, установленные в проекте, среди них должна присутствовать сборка `Strong` (рис. 4.18).

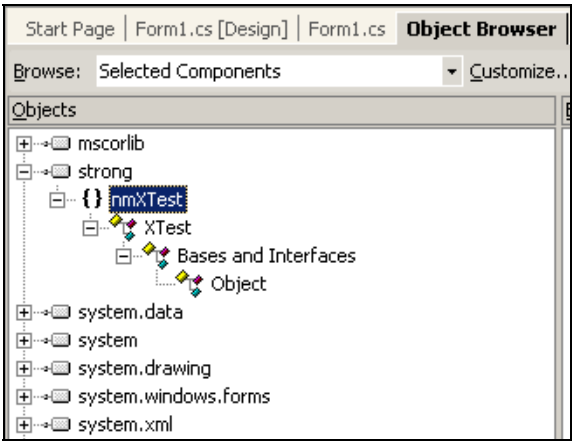


Рис. 4.18. Исследование внутреннего строения сборки при помощи `Object Browser`

Заключительным этапом в создании приложения будет добавление кода, использующего строго именованную сборку. Он будет состоять из одной строки.

```
nmXTest.XTest.Test();
```

Этот код следует добавить в то место, откуда он будет гарантированно вызван. Такими элементами могут быть: конструктор главной формы, обработчик нажатия кнопки и т. п. Это необходимо, чтобы быть уверенным в том, что код будет действительно вызван. Полный код приложения вы сможете найти в листинге 4.9 на прилагаемом к книге компакт-диске.

Во время набора исходного кода, вы сможете насладиться технологией IntelliSence, подсказывающей содержание типов, прямо по ходу редактирования (рис. 4.19).

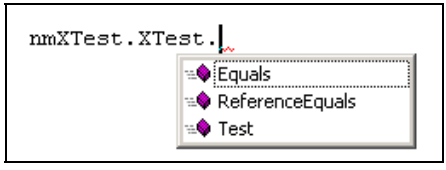


Рис. 4.19. Технология IntelliSence, подсказывающая программисту в ходе редактирования

На этом создание программы закончено, теперь осталось лишь откомпилировать и запустить ее. В результате запуска приложения, на экране появится диалоговое окно с надписью: "Hello, World, from test strong named assembly!" (рис. 4.20).

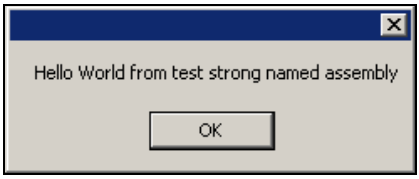


Рис. 4.20. Результат работы кода, расположенного в строго именованной сборке

Код, выводящий диалоговое окно, содержится в сборке `Strong`, которая располагается в GAC. Мы создали первый общий сервис, расположенный в совместной сборке, и его можно использовать в любых своих приложениях.

Для того чтобы убедиться в том, что диалоговое окно действительно выводится сборкой, расположенной в GAC, проведем эксперимент. Удалим сборку из GAC, после чего попытаемся запустить приложение.

В результате будет получено сообщение о том, что приложение было остановлено по причине произошедшего необработанного исключения (рис. 4.21).

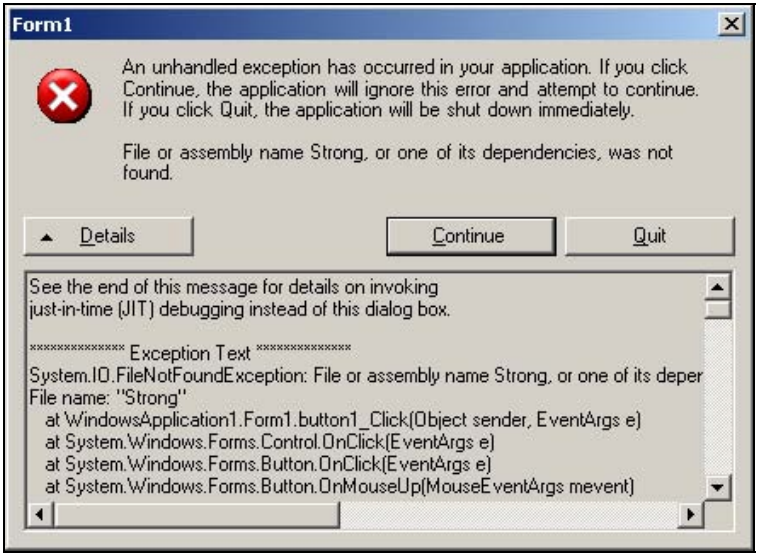


Рис. 4.21. Среда исполнения не смогла найти необходимую строго именованную сборку

Описание произошедшего исключения указывает на то, что не была найдена сборка `Strong`. Значит, сборка действительно подгружается из GAC.

Загрузка сборок, как и модулей, происходит по первому требованию. То есть

при первом обращении к сборке. Огромным плюсом такой политики подключения сборок является существенное увеличение скорости загрузки приложения, а также вероятная возможность работы приложения при отсутствии некоторых его компонентов.

Теперь напишем программу, использующую совместную сборку, не прибегая к помощи среды разработки Visual Studio .NET, что, впрочем, окажется не намного сложнее. Это будет консольное приложение, код которого приведен в листинге 4.8.

Листинг 4.8. Консольное приложение, использующее строго именованную сборку

```
/*
  Листинг 4.8
  File:   Mod.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Ради этой строки создавалось приложение. Здесь
    // вызывается функция из совместной сборки, расположенной в GAC.
    // В этом случае, вызов происходит по полному имени,
    // включающему пространство имен. В реальных приложениях
    // чаще всего пространство имен подключается глобально,
    // при помощи директивы using. (В нашем случае using nmXTest;)
    nmXTest.XTest.Test();
  }
};
```

Задача приложения заключается лишь в обращении к коду, расположенному в строго именованной сборке. Она реализуется единственной строкой кода функции `Main`. Откомпилировать этот исходный код в приложение можно при помощи следующей команды.

```
csc /r:Strong.dll Some.cs
```

Здесь ключ `/r` задает связь со строго именованной сборкой. При этом требуется, чтобы сборка находилась в распоряжении компилятора, то есть располагалась в текущем каталоге или в одном из каталогов, указанных в переменной `Path`. После компиляции надобность в этой сборке отпадет и ее можно будет смело удалить. На стадии компиляции сборка необходима для того, чтобы сделать правильные записи в манифесте программы о связи с этой сборкой, а также проверить правильность типов, используемых из нее.

Если заглянуть в манифест созданного приложения, там обнаружатся следующие строки, указывающие на связь со сборкой `Strong`.

```
// Это ссылка на нашу сборку со строгим именем.
.assembly extern Strong
{
   // Маркер открытого ключа сборки, гарантирующий уникальность
   // ее строгого имени
  .publickeytoken = (51 F6 C8 9D E8 95 38 87 )          // Q.....8.
   // Необходимая версия сборки Strong.
  .ver 1:0:0:0
}
```

Эта информация является строгим именем и может быть записана в более компактном виде.

```
Strong, Version=1.0.0.0, Culture=neutral, PublicKeyToken=51f6c89de8953887
```

Компилятор командной строки для удобства автоматически подключает некоторые стандартные сборки. Они перечислены в файле `csc.rsp`, который хранится вместе с компилятором (`csc.exe`) в основном каталоге среды исполнения (%windir%\Microsoft.NET\xxxx). Содержимое этого файла представлено ниже.

```
# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.

# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.Vsa.dll
/r:System.Configuration.Install.dll
/r:System.Data.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.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
```

Если в программе будет использован код одной из этих сборок, то она будет автоматически подключена в манифест. Если нет, — то не будет. Вы можете добавить в этот файл собственные ссылки либо подключить свою версию файла при помощи символа "собаки".

```
csc @MyRsp.rsp Some.cs
```

Помимо ссылок на сборки, в `rsp`-файле могут быть указаны любые параметры командной строки компилятора. К примеру, `rsp`-файл может выглядеть следующим образом (листинг 4.9).

Листинг 4.9. Пример `rsp` файла, с командами для компилятора C#

```
# Это файл дополнительных параметров для компилятора csc.
/t:library
/r:Strong.dll
Some.cs
```

Команды файла предписывают компилятору создать библиотечную сборку, использующую `Strong.dll`.

Внутренний формат имен

Загрузка дополнительных внешних сборок производится непосредственно перед обращением к ним. В общих чертах этот механизм описывался выше. Его работа возможна лишь благодаря продуманному внутреннему формату

имен. Каждое имя среды исполнения содержит в себе ссылку на сборку или модуль, в котором располагается тип, на который оно ссылается. К примеру, вызов метода `Test` из нашей сборки

```
nmXTest.XTest.Test();
```

транслируется компилятором в следующую инструкцию

```
call       void [Strong]nmXTest.XTest::Test().
```

Полное имя функции содержит информацию о сборке, в которой она расположена. Если же тип располагается в модуле, а не в сборке, то используется специальная директива `.module`.

```
call       void [.module Strong]nmXTest.XTest::Test()
```

Для местных типов имя сборки не указывается, хотя оно и присутствует в неявном виде в .NET байт-коде.

Все сказанное верно на уровне IL-кода, который является удобочитаемой интерпретацией .NET байт-кода. На физическом уровне полные имена, естественно, не задаются при помощи строки, в начале которой указывается имя сборки, обрамленное квадратными скобками. Имя формируется при помощи нескольких специальных таблиц и ссылок между ними. Но на данном этапе нас вполне устроит уровень IL-кода. Здесь имена задаются полностью (директив подключения пространств имен не существует) в следующем формате.

```
[ИмяСборки]ПространствоИмен::Имя
```

4.4. Политика версий для строго именованных сборок

Для начала уясним самое главное: политика версий существует только на уровне сборок и применяется только для строго именованных сборок. Сборка — это наименьший и единственный элемент, которому может быть назначена версия. Все элементы внутри сборки версий не имеют или имеют версию родительской сборки, что фактически равнозначно. Таким образом, политика версий действует только на сборки, а на другие программные элементы (классы, интерфейсы и т. д.) влияния не оказывает. Конечно, можно создать собственную искусственную политику версий на уровне типов, но в большинстве случаев это не будет оправдано.

Информация о версии

Информация о версии сборок состоит из четырех чисел. Первые два называются основной частью версии, а вторые два — дополнительной. К тому же, каждое из них имеет собственное имя, перечислим их в порядке следования в версии: `Major`, `Minor`, `Build` и `Revision` (рис. 4.22).

Где:

- `Major` — основная версия;
- `Minor` — второстепенная версия;
- `Build` — номер построения;
- `Revision` — номер ревизии для текущего построения.

На первый взгляд такое количество чисел в версии приложения может показаться избыточным. Действительно, для создания обычных программ оно ни к чему. Но при разработке больших корпоративных продуктов, включающих множество библиотек и приложений, подобное строение информации о версии просто необходимо. Поскольку в подобных проектах конфликты возникают не только на уровне различных версий приложений, но даже на уровне промежуточных ревизий и построений.

Рис. 4.22. Структура информации о версии

Информация о версии приложения разбита на две части не случайно. При поиске необходимой сборки, политика версий требует точного совпадения лишь основной версии сборки. А дополнительная версия используется для поиска наиболее свежей сборки. То есть будет загружена сборка с наибольшими номерами построения и ревизии, при условии совпадения основной части версии.

Фактически, основная часть информации о версии используется для разграничения версий, а дополнительная — для динамического обновления уже существующих версий. Дополнительная часть версии весьма удобна для создания пакетов обновления (Service Pack) и заплат безопасности.

Использование такой политики требует обратной совместимости только от библиотек с одинаковыми основными версиями. Это означает, что, поменяв основную часть информации о версии, будь то `Major` или `Minor` число, вы

более не обязаны поддерживать старые типы и другие программные элементы. Новые сборки попросту не будут рассматриваться средой исполнения в виде кандидатов на загрузку из более ранних версий приложений.

Работаем с информацией о версии

Для работы с информацией о версии на уровне исходного кода предназначен следующий атрибут.

`System.Reflection.Assembly.VersionAttribute.`

Его использование отличается от использования обычных атрибутов и требует специального синтаксиса, поскольку компилятору необходимо указать, что он применяется к сборке. В языке C# это выглядит так.

```
using System.Reflection;

[assembly: AssemblyVersion("1.0.0.0")]
```

Или так.

```
[assembly: System.Reflection.AssemblyVersion("1.0.0.0")]
```

В качестве единственного параметра атрибута необходимо в виде строки указать версию сборки, для которой применяется этот атрибут.

Версию для атрибута можно указывать не полностью. Существует возможность частичного задания версии, путем замены дополнительной части информации версии звездочками. В этом случае, номер построения будет равен количеству дней, прошедших с января 2000 года, а номер ревизии будет равен количеству секунд, прошедших с полуночи, деленных на два. Такая схема формирования информации о версии обеспечивает ее уникальность для каждого построения приложения.

Также существует возможность неполного задания информации о версии, при которой некоторые из чисел могут вообще не указываться и будут дополняться автоматически. Подытожим все способы задания версии в табл. 4.2.

Таблица 4.2. *Способы задания информации о версии при помощи атрибута* `AssemblyVersionAttribute`

Версия в исходном коде	Реальная версия
1	1.0.0.0
1.1	1.1.0.0
1.1.*	1.1. [количество дней, прошедших с января 2000]. [количество секунд, прошедших с полуночи, деленное на два]

1.1.1	1.1.1.0
1.1.1.*	1.1.1. [количество секунд, прошедших с полуночи, деленное на два]
1.1.1.1	1.1.1.1

За преобразование строки версии, заданной не в полном формате, отвечает компилятор. Вполне возможно, что при использовании компилятора стороннего производителя, эти правила выполняться не будут. Перед тем как использовать такую возможность, необходимо изучить документацию, прилагаемую к компилятору.

Для того чтобы убедиться, что информация о версии формируется компилятором, а не атрибутом `AssemblyVersionAttribute`, рассмотрим его код, взятый из .NET CLI Shared Source.

```
[AttributeUsage (AttributeTargets.Assembly, AllowMultiple=false)]
public sealed class AssemblyVersionAttribute : Attribute
{
  private String m_version;
  public AssemblyVersionAttribute(String version)
  {
    m_version = version;
  }
  public String Version
  {
    get { return m_version; }
  }
}
```

В программе отсутствуют операции вычисления даты и времени, которые обязательно должны были бы использоваться при формировании номера версии атрибутом.

Дополнительная информация о версии

Помимо сухой числовой информации о версии, существует возможность задать дополнительную информацию в виде строки. Для этого предназначен атрибут `System.Reflection.AssemblyInformationalVersion`. Его использование на языке C# выглядит так.

```
using System.Reflection;
[assembly: AssemblyInformationalVersion("This is super programm")]
```

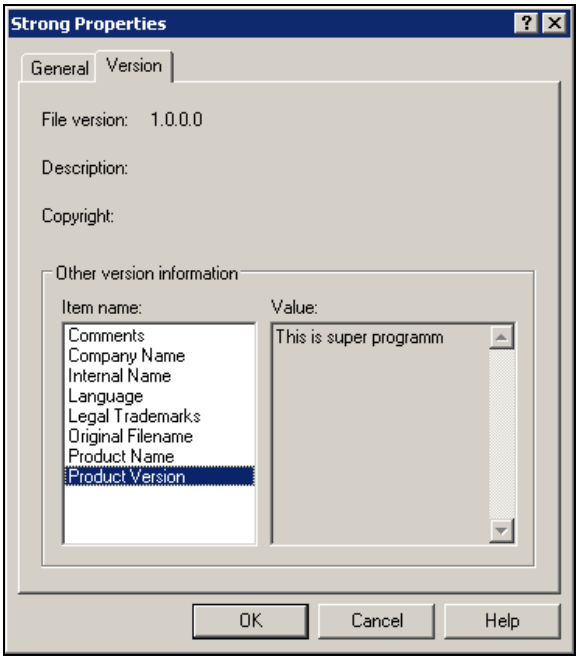


Рис. 4.23. Дополнительная информация о версии

Или так.

```
[assembly: System.Reflection.AssemblyInformationalVersion
("This is super programm");
```

Информацию, заданную при помощи этого атрибута, можно просмотреть на вкладке **Version**, в окне свойств файла, выбрав пункт **Product Version** (рис. 4.23).

Эта информация очень полезна, если требуется наглядно показать различие версий приложений. К примеру, таким способом можно указать на различие между версией для обучения (Education Edition) и полноценным профессиональным пакетом (Professional Edition).

Технология прямого запуска

В рамках политики управления версиями существует один малозаметный нюанс, который может вызвать очень неприятные проблемы. Речь пойдет о совместном использовании разных версий одинаковых сборок.

Различные части одного приложения могут запросить две "одинаковые"

сборки разных версий. Поскольку они могут быть запрошены косвенно, через зависимые сборки или модули, то само приложение может ничего не знать об этой ситуации. На удивление, проблем с использованием и подгрузкой самих сборок возникнуть не должно. Среда исполнения автоматически загрузит необходимые версии сборок, прозрачно для самой программы.

Напомним, что политика управления версиями применяется только к строго именованным сборкам. А строгое имя неотъемлемо включает в себя версию сборки. Следовательно, строгие имена сборок, у которых будут различаться только версии, для среды исполнения будут полностью различны. Сборки будут автоматически подгружены и проблем с их использованием не возникнет (рис. 4.24).

Поясним рисунок. Обратить внимание стоит, прежде всего, на сборку `yyy.dll`, ее две разные версии одновременно используются приложением. Одна запрашивается непосредственно самой программой, а вторая вызывается косвенно, через сборку `x.dll`. Обе эти сборки используются одновременно и не знают о существовании друг друга, несмотря на то, что обе загружены в одном адресном пространстве.

Ранее это было невозможно, поскольку загрузчик операционной системы не различает версии динамических библиотек. Соответственно, для него одинаковые файловые имена библиотек автоматически предполагают одинаковые библиотеки, даже если у них различный код. Эта проблема является одной из причин "ада динамических библиотек". Как видно, она с успехом решена в .NET.

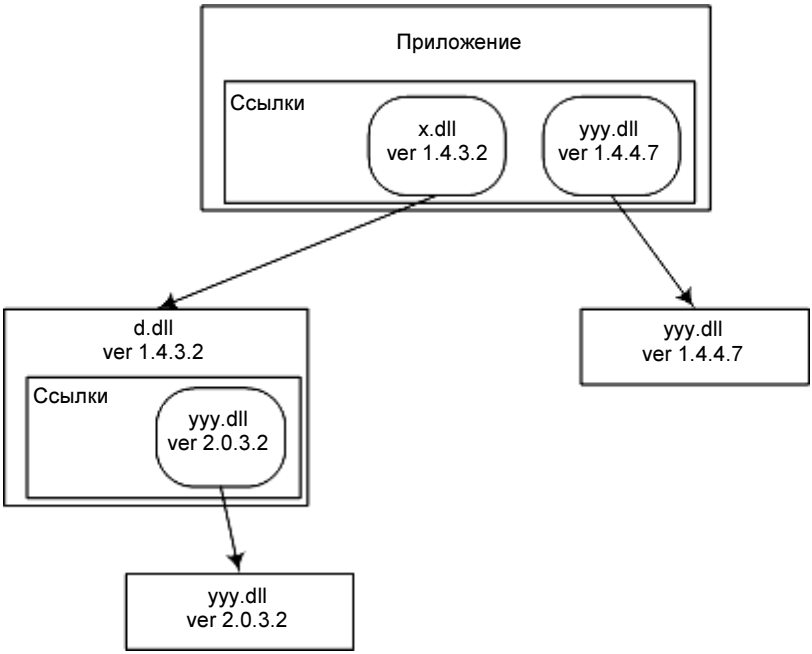


Рис. 4.24. Технология прямого запуска

Хотя технология прямого запуска и предполагает изоляцию различных версий сборок друг от друга, здесь кроется одна очень неприятная проблема. При использовании (даже неявно) двух сборок различных версий, весьма вероятны конфликты при совместном использовании объектов операционной системы. Простейшим примером таких объектов являются обычные файлы. Доступ к ним производится при помощи символьных имен, поэтому, если две версии одной и той же библиотеки попытаются одновременно использовать один файл, то коллизии, скорее всего, избежать не удастся. Либо не будут работать обе библиотеки, либо одна из них; все будет зависеть от выбора способа доступа к файлу.

Файлы являются не единственными проблемными объектами, на самом деле их множество. Перечислим некоторые из них: файлы, именованные файлы, проецируемые в память, сокеты, порты, события, семафоры, ветки реестра, а также огромное множество других объектов. Многие из них косвенно взаимодействуют с другими объектами, которые в свою очередь могут оказаться проблемными. Для того чтобы точно знать все подводные камни, необходимо очень хорошо разбираться во внутреннем устройстве операционной системы, а также во внутренней реализации FCL. Разработчики FCL старались отгородить пользователей от этих проблем, автоматизировав, по возможности, работу со всеми именованными и совместными объектами. Но полностью этого

сделать принципиально невозможно. Поэтому при создании совместных сборок, которые будут использоваться несколькими приложениями, необходимо быть предельно осторожным.

Внимание

Проблемы могут возникнуть не только при использовании различных версий сборок в границах одного приложения, но даже и для одной версии сборки, но уже в рамках всей системы в целом. Многие объекты операционной системы могут использоваться совместно на уровне всей системы, а не только одного приложения. Простейшим примером таких объектов являются те же файлы.

В качестве совета можно порекомендовать сделать имена всех совместных объектов уникальными. Осуществить это можно несколькими способами, либо генерируя имена по ходу исполнения программы, используя динамическую генерацию GUID (функция — `CoCreateGuid`), либо включать в имена версию сборки или даже ее полное строгое имя, обязательно включая маркер открытого ключа.

4.5. Процесс загрузки .NET-приложений

Любая программа .NET, будь то exe-файл или сборка, прежде всего, является файлом формата PE, запускаемым стандартными средствами. Следовательно, она должна взаимодействовать с загрузчиком исполняемых файлов Windows. Все программы .NET представляют собой байт-код, для исполнения которого необходима виртуальная машина .NET. Загрузчик исполняемых файлов операционной системы ничего не знает о байт-коде и передает код файла на исполнение процессору. Как же начинает работать виртуальная машина .NET? Оказывается, каждый файл приложения .NET имеет в таблице импорта ссылку на библиотеку `mscoree.dll`, которая является загрузчиком виртуальной машины. Mscoree расшифровывается как Microsoft Component Object Runtime Execution Engine.

Примечание

Таблица импорта — это специальная секция внутри PE-файла, в которой указаны библиотеки, необходимые для работы приложения или библиотеки.

После проецирования исполняемого файла .NET на адресное пространство процесса, загрузчик Windows сканирует таблицу импорта и загружает все указанные там библиотеки. В том числе и `mscoree.dll` (в большинстве при-

ложений .NET она будет единственной динамической библиотекой, присутствующей в таблице импорта). После загрузки всех необходимых библиотек и подготовки адресного пространства приложения, загрузчик Windows передает исполнение на точку входа приложения. В случае приложения .NET на месте точки входа расположена следующая заглушка.

```
start   proc near
  jmp ds:_CorExeMain
start   endp
```

Код заглушки представлен на языке Ассемблера. Он состоит из одного безусловного перехода на функцию `_CorExeMain`, расположенную в библиотеке `mscoree.dll`. Функция производит запуск виртуальной машины .NET, которая после старта начинает обрабатывать этот исполняемый файл. Она находит .NET байт-код, расположенный в нем, и при помощи метаданных отыскивает в нем точку входа. Затем начинает исполнение файла, передав его код JIT-компилятору.

Помимо обычных исполняемых файлов, приложения .NET могут размещаться в динамических библиотеках. В этом случае заглушка будет располагаться в функции `DllMain` и выглядит следующим образом.

```
start   proc near
    jmp ds:_CorDllMain
start   endp
```

Эта заглушка отличается от предыдущей лишь тем, что вызывает функцию `_CorDllMain`, также расположенную в библиотеке `mscoree.dll`.

Примечание

У многих может возникнуть вопрос, почему этот код называется заглушкой, а не функцией. Дело в том, что поскольку код не имеет основного признака функции, он не имеет пролога, а также не возвращает управление (На уровне Ассемблера это делается при помощи инструкции `ret`).

Далее подробно рассмотрим случай, когда приложение .NET располагается в динамической библиотеке.

Динамические библиотеки на основе сборок

Судя по названию подраздела, может показаться, что будут рассмотрены обычные библиотечные сборки. Однако речь пойдет о создании стандарт-

ных динамических библиотек, основанных на технологиях .NET, которые можно будет использовать при помощи базовых средств операционной системы. Продемонстрируем возможности технологии на примере. Для этого создадим динамическую библиотеку, экспортирующую функции, код которых будет непременно исполняться под управлением среды .NET. Затем создадим приложение, использующее эту библиотеку при помощи базового Windows API.

Код будущей библиотеки представлен на листинге 4.10.

Листинг 4.10. Классическая динамическая библиотека, основанная на технологиях .NET

```
/*
  Листинг 4.10
  File:   Mod.cpp
  Author: Дубовцев Алексей
*/
// Подключаем стандартную библиотеку.
#using <mscorlib.dll>
// Подключаем основное пространство имен стандартной библиотеки.
using namespace System;
//Эта функция будет экспортироваться из нашей библиотеки
// стандартным образом, через таблицу экспортируемых функций в
// заголовке PE файла библиотеки.
void Test()
{
  // Проинформируем пользователя о том, что функция была вызвана.
  Console::WriteLine(S"Hello, World, from .NET DLL!");
}
```

Код библиотеки содержит единственную функцию, доступ к которой будет возможен извне.

Примечание

Обратите внимание на модификатор **S**, расположенный перед строкой; он указывает на то, что она должна быть определена в стиле .NET и расположена совместно с IL-кодом в метаданных. Иначе она будет расположена в секции `.data` PE файла, как в обычном неуправляемом приложении. В этом случае, для ее использования потребуется обращение к специальным механизмам, что в свою очередь может несколько замедлить работу приложения.

Для компиляции исходного кода в конечную библиотеку понадобится дополнительный файл с определениями экспортируемых функций. Это файл `Some.def`, представленный в листинге 4.11.

Листинг 4.11. Файл, указывающий экспортируемые из библиотеки функции

```
;
;       Листинг 4.11
;       File:   Some.def
;       Author: Дубовцев Алексей
; Укажем имя файла библиотеки.
LIBRARY Some.dll
; В этом разделе указываются экспортируемые функции.
EXPORTS
  ; Экспортируем из библиотеки функцию Test, которая
  ; выведет приветствие на консоль.
  Test
```

Для компиляции исходного кода воспользуемся двумя следующими командами, поочередно вызывающими компилятор и линковщик.

```
cl Some.cpp /c /clr
link Some.obj /DLL /def:Some.def /noentry
```

Первая команда обеспечит компиляцию исходного кода в объектный файл, а вторая соберет из него динамическую библиотеку. В результате должна получиться динамическая библиотека, в таблице экспорта которой указана единственная функция — `Test`.

Теперь необходимо написать программу, использующую эту сборку. Ее задача будет заключаться в загрузке библиотеки и обращении к экспортируемой из нее функции. Исходный код программы приведен в листинге 4.12.

Листинг 4.12. Программа, использующая динамическую библиотеку с использованием стандартного API

```
/*
  Листинг 4.12
  File:   Use.cpp
  Author: Дубовцев Алексей
*/
// Подключаем базовое Windows API.
```

```
#include <windows.h>
// Точка входа в приложение.
void main()
{
  // Описатель библиотеки, которую нам предстоит загрузить.
  // Это будет именно та библиотека, которую мы создали
  // при помощи компилятора MC++.
  HANDLE hLib;
  // Указатель на функцию в загруженной библиотеке.
  void (*pT) (void);
  // Адрес функции в библиотеке.
  UINT dwAddr;
  // Загружаем библиотеку.
  hLib = LoadLibrary("Some.dll");
  // Если не получилось, то выходим.
  if (!hLib) return;
  // Отыскиваем в библиотеке необходимую функцию.
  pT = (void (*) ())GetProcAddress((HMODULE)hLib,"Test");
  // Если нужной нам функции библиотека не содержит, то
  // выходим.
  if (!pT) return;
  // Вызываем функцию.
  (*pT)();



// А еще ее можно вызвать вот так!
  dwAddr = (UINT)pT;
  __asm call dwAddr;
}
```

Для компиляции исходного кода необходимо воспользоваться следующей командой:

```
cl Use.cpp
```

В результате работы программы на консоли появятся две строки.

```
Hello, World, from .NET DLL!
Hello, World, from .NET DLL!
```

Обе они выводятся одной и той же функцией, расположенной в динамической библиотеке. Первый раз она вызывается при помощи стандартных

средств языка, а второй раз напрямую, используя ассемблерную инструкцию `call`.

При поверхностном осмотре ничего удивительного в работе приложения нет. Оно всего лишь выводит на консоль две строки. Для более углубленного анализа, дизассемблируем библиотеку `Some.dll`, функция `Test` которой и выводит на консоль строку. Код этой функции выглядит следующим образом.

```
.method public static void modopt([mscorlib]System.
Runtime.CompilerServices.CallConvCdecl)
        Test() cil managed
{
  .vtentry 1 : 1
  .export [1] as Test
  // Code size       11 (0xb)
  .maxstack  1
  // Загружаем указатель на строку.
  IL_0000:  ldstr      "Hello, World, from .NET DLL!"
  // Вызываем функцию, выводящую сообщение на консоль.
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  // Возвращаемся из функции.
  IL_000a:  ret
}
```

Полученный код полностью управляемый и может исполняться только под контролем виртуальной машины .NET. Но в нашей программе эта функция использовалась при помощи базового Windows API, не используя дополнительных средств среды .NET. Вызов функции осуществлялся при помощи прямой инструкции `call`, что в свете новой информации выглядит несколько алогично.

Конечно же, функция не вызывалась напрямую. На самом деле произошло обращение к некой заглушке, которая была указана в таблице экспорта библиотеки. Ее код после дизассемблирования выглядит так.

```
Test proc near
    jmp dword_10003000
Test endp
```

Я это обычно называю вызовом в "космос" или в никуда. На самом деле, при загрузке этой библиотеки, автоматически инициализируется среда исполнения .NET. (Этот механизм был описан ранее.) Она динамически перехватывает все связи в нашей библиотеке. И когда вызывается функция `Test`, управление неявно передается виртуальной машине .NET, которая вызывает

настоящую функцию. Таким образом, появляется возможность вызывать управляемый код .NET из обычного кода.

Хотя в этом разделе и показан пример взаимодействия неуправляемого и управляемого кода, но не это главное. В первую очередь необходимо обратить внимание на загрузку созданной динамической библиотеки, на элегантность и красоту технологии, которая объединяет базовый мир Windows и среду .NET в единое целое.

Локализация приложений при помощи сборок

Среда исполнения имеет в своем арсенале мощные, полностью автоматизированные средства локализации приложений. Основным средством локализации являются сателлитные сборки (satellite assembly). Такие сборки содержат регионально адаптированные ресурсы: строки, изображения и другую, регионально зависимую информацию. Microsoft настоятельно не рекомендует располагать в сателлитных сборках код, они предназначены лишь для хранения ресурсов.

В локализованных приложениях предполагается использовать двухуровневую модель, разделяя код и интерфейсную часть. Код размещается в отдельных монолитных сборках, которые будут едины для всех региональных версий приложения. А вся информация, связанная с интерфейсом пользователя и которая нуждается в локализации, помещается в сателлитные сборки. Они будут автоматически подключены средой исполнения в соответствии с региональными стандартами компьютера, на котором будет запускаться приложение. Рассмотрим процесс создания программы, основанной на сателлитных сборках и локализованной для русского и английского языков. Сначала разберем теоретические основы вопроса.

Итак, для каждой сборки задается региональная принадлежность. За это отвечает запись `.culture` в манифесте. Региональные идентификаторы в среде .NET состоят из двух частей. Первая определяет язык, вторая регион.

Примечание

Справедливо может возникнуть вопрос, для чего кроме языка задавать еще и регион? Региональная информация построена подобным образом из-за существования многочисленных диалектов языков. Порой люди, разговаривающие на разных диалектах одного и того же языка, не могут даже понять друг друга. Впрочем, идентификатор региона является опциональным и его можно не задавать.

Количество региональных идентификаторов составляет более двухсот. Наиболее интересные из них представлены в табл. 4.3.

Таблица 4.3. *Примеры региональных идентификаторов*

Идентификатор	Числовое значение	Язык и регион (английский)	Язык и регион (русский)
en-US	0x0409	English — United States	Английский — США
en-GB	0x0809	English — United Kingdom	Английский — Объединенное Королевство (Англия, Ирландия, Шотландия, Графство Уэллс)
ru-RU	0x0419	Russian — Russia	Русский — Россия
uk-UA	0x0422	Ukrainian — Ukraine	Украинский — Украина

Пугаться присутствия числовых идентификаторов во втором столбце не стоит, поскольку работать с ними рядовым программистам не придется, они используются лишь во внутренних целях среды исполнения.

Создадим локализованное приложение, оно будет крайне примитивно — его основной целью будет показать возможности локализации при помощи сателлитных сборок. Код самого приложения представлен в листинге 4.13.

Листинг 4.13. Приложение, локализованное под русский и английский языки

```
/*
  Листинг 4.13
  File:   Use.cpp
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Подключаем пространство имен, отвечающее за локализацию.
using System.Globalization;
// Подключаем пространство имен, содержащее сервисы для работы с потока-
ми.
using System.Threading;
// Подключаем пространство имен, содержащее сервисы для работы с ресурсами.
using System.Resources;
// Основной класс приложения.
class Application
{
  // Менеджер для работы с ресурсами в стиле .NET.
```

```
  private static ResourceManager rm;
  // Точка входа в приложение.
  public static void Main()
  {
    // Подключаем менеджер ресурсов.
    rm = new ResourceManager("MyStrings", typeof(Application).Assembly);
        // Выводим на консоль строку, запрошенную по идентификатору.
    // App_Hello_World
    Console.WriteLine(rm.GetString("App_Hello_World"));
    // ЛОКАЛИЗУЕМ ПРИЛОЖЕНИЕ — меняем его региональную
    // принадлежность.
    Thread.CurrentThread.CurrentUICulture = new CultureInfo("ru");
    // Выводим на консоль строку, запрошенную из строкового
    // ресурса по идентификатору App_Hello_World.
    Console.WriteLine(rm.GetString("App_Hello_World"));
  }
};
```

Теперь необходимо создать две версии строковых ресурсов. Первая версия будет регионально нейтральной, с которой приложение работает по умолчанию, а вторая будет русскоязычной. Первая будет размещаться в каталоге приложения, в файле `MyStrings.txt`. Этот файл в нашем случае будет состоять всего из одной строки.

```
App_Hello_World = Hello World
```

Первый параметр определяет идентификатор строки, а второй саму строку. Прежде чем использовать этот файл строк, его необходимо скомпилировать в файл ресурсов. Сделать это можно при помощи следующей команды.

```
resgen MyStrings.txt MyStrings.resources
```

Теперь можно скомпилировать основное приложение, встроив в исполняемый файл только что созданные ресурсы.

```
csc Sample.cs /res:MyStrings.resources
```

Теперь необходимо создать для приложения русские ресурсы. Они обязательно должны располагаться в подкаталоге `ru` приложения. Это обязательное требование среды исполнения, поскольку именно там она будет искать сателлитную сборку, соответствующую русскому языку. Если говорить более обобщенно, то ресурсы должны располагаться в подкаталоге, имя которого совпадает с региональным идентификатором искомого языка.

Итак, в каталоге `ru` необходимо создать файл `MyStrings.ru.txt`. В его имя также должен входить региональный идентификатор. К этому файлу предъявляется еще

одно требование: текст, располагающийся в нем, обязательно должен быть в формате Unicode.

```
App_Hello_World = Здравствуй, мир!
```

Идентификатор строки остался все тот же, но зато строка уже локализована. Из этого файла, также, как из предыдущего, необходимо создать файл ресурсов при помощи утилиты `resgen`.

```
resgen MyStrings.ru.txt MyStrings.ru.resources
```

Из которого уже предстоит создать сателлитную сборку. Сделать это можно при помощи утилиты `al`.

```
al /out:Sample.Resources.dll /c:ru /embed:MyStrings.ru.resources
```

Имя сателлитной сборки должно формироваться по следующему правилу `<Имя Исполняемого Файла Приложения>.Resources.dll`.

Теперь наше приложение готово к запуску и тестированию. Результатом его работы должны стать две следующие строки, выведенные на консоль.

```
Hello, World!
Здравствуй, мир!
```

Строение локализованного приложения на диске схематично изображено на рис. 4.25.

Рис. 4.25. Строение локализованного приложения

Кроме использования сателлитных сборок, локализацию можно проводить непосредственно через файлы ресурсов. Но эта тема выходит за рамки главы и рассмотрена не будет.

В заключение вопроса необходимо упомянуть о возможности задания региональной принадлежности программно, при помощи атрибута `AssemblyCulture`. Приведем пример.

```
Using System.Reflection;
[assembly: AssemblyCulture("en-US")]
```

или так

```
[assembly: System.Reflection.AssemblyCulture("en-US")]
```

Этот атрибут позволяет задать региональную принадлежность сборки с кодом. Но поскольку Microsoft не рекомендует располагать в регионально зависимых сборках код, таким атрибутом следует пользоваться только в крайних случаях.

4.6. Конфигурирование политики загрузки сборок

Структура конфигурационного файла

Среда исполнения .NET позволяет управлять процессом загрузки сборок, при помощи конфигурационных файлов. Это обычные текстовые файлы в формате XML, которые размещаются вместе с приложением в одном каталоге. Имя конфигурационного файла формируется по следующему правилу: `<Имя Исполняемого Файла приложения>.config`

То есть если приложение располагается в файле `Some.exe`, то его конфигурационный файл должен называться `Some.exe.config`. Для наглядности приведем пример простого конфигурационного файла (листинг 4.14).

Листинг 4.14. Пример конфигурационного файла

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="Strong"
                          publicKeyToken="51F6C89DE8953887"/>
        <codeBase version="1.0.0.0"
                      href="http://lexa:81/one/Strong.dll"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Любой конфигурационный файл обязательно содержит тег `<configuration>`. Теги, вложенные в него, разбивают конфигурационный файл на подразделы, каждый из которых отвечает за свою область настроек. Всего может быть восемь следующих подразделов:

- `<startup>` — отвечает за параметры запуска приложения;
- `<runtime>` — позволяет провести настройку политики загрузки приложения;
- `<remoting>` — содержит информацию об удаленных объектах и каналах;
- `<network>` — содержит настройки соединения приложения с Интернетом;
- `<cryptographySetting>` — содержит настройки криптографической подсистемы общей библиотеки классов;
- `<configuration>` — позволяет занести в конфигурационный файл пользовательские настройки;
- `<system.diagnostics>` — содержит дополнительные отладочные настройки;
- `<system.web>` — содержит настройки для приложений ASP.NET.

Из всего списка рассмотрим только тег `<runtime>`, поскольку он напрямую связан с политикой управления загрузкой сборки. Структура тега показана в листинге 4.15.

Листинг 4.15. Структура тега `<runtime>`

```
<configuration>
  <runtime>
    <developmentMode>
    <assemblyBinding>
      <probing>
      <publisherPolicy>
      <qualifyAssembly>
      <dependentAssembly>
        <assemblyIdentity>
        <bindingRedirect>
        <codeBase>
        <publisherPolicy>
```

Элементы, находящиеся на этой схеме правее других, должны быть вложены в своих левых соседей. К примеру, тег `<runtime>` должен быть вложен в тег `<configuration>` и т. д.

Рассмотрим более подробно перечисленные теги.

<developmentMode>

Прототип тега:

```
<developmentMode developerInstallation="true | false"/>.
```

Тег предназначен для разработчиков. В случае если его параметр `developerInstallation` имеет значение `true`, то среда исполнения помимо обычных каталогов попытается искать сборки в соответствии с путями, указанными переменной среды окружения `DEVPATH`.

<assemblyBinding>

Сам тег является пустым и служит лишь для объединения вложенных подтегов. Рассмотрим по порядку его подтеги.

<probing>

Прототип тега:

```
<probing privatePath="paths"/>.
```

Тег позволяет указать дополнительный список подкаталогов, в которых будет производиться поиск сборок используемых в персональном режиме.

Примечание

Сборками, используемыми в персональном режиме, называются те, которые располагаются в одном каталоге с приложением или в одном из подкаталогов.

Для примера, рассмотрим одно из созданных ранее приложений, которое использует сборку. Пусть это будет сборка `Strong.dll`, которая располагается в одном каталоге с приложением. Переместим эту сборку в предварительно созданный подкаталог `asm` (имя подкаталога выбрано абсолютно случайно) и попробуем запустить приложение. Естественно, что запуск не произойдет, и на экране появится диалоговое окно, сообщающее о произошедшем исключении (рис. 4.26).

Обратите внимание на тип произошедшего исключения, он указан в верхней части окна (`System.IO.FileNotFoundException`). Для того чтобы полностью разобраться, в чем дело, посмотрим детальную информацию об исключении.

Примечание

Для просмотра дополнительной информации об исключении, необходимо нажать кнопку **Yes**, в окне **Just-In-Time Debugging**. После чего можно ознакомиться с ней в центральной части появившегося окна. Нажатие кнопки **Break** приостановит исполнение приложения.

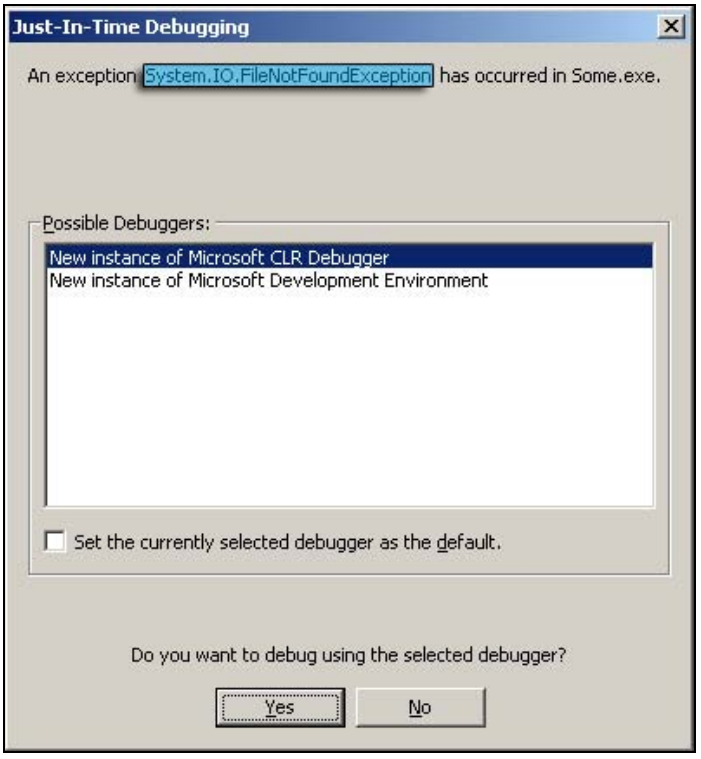


Рис. 4.26. Исключение, произошедшее при запуске приложения из-за отсутствия необходимой сборки

Поскольку мы использовали приложение консольного типа, то вся информация об исключении будет автоматически выведена на консоль. Пример лога сбоя представлен в листинге 4.16.

Листинг 4.16. Лог сбоя, произошедшего из-за отсутствия необходимой библиотеки

```
Unhandled Exception: System.IO.FileNotFoundException: File or assembly
name Strong, or one of its depe
ndencies, was not found.
File name: "Strong"
   at App.Main()
Fusion log follows:
=== Pre-bind state information ===
LOG: DisplayName = Strong, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=51f6c89de8953887
```

```
 (Fully-specified)
LOG: Appbase = E:\Projects\.NET\Assembly\Use\Probing\
LOG: Initial PrivatePath = NULL
Calling assembly : Some, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null.
===
LOG: Application configuration file does not exist.
LOG: Publisher policy file is not found.
LOG: Host configuration file not found.
LOG: Using machine configuration file from
C:\WINXP\Microsoft.NET\Framework\v1.0.3705\config\machine.config.
LOG: Post-policy reference: Strong, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=51f6c89de8953887
LOG: Attempting download of new URL
file:///E:/Projects/.NET/Assembly/Use/Probing/Strong.DLL.
LOG: Attempting download of new URL
file:///E:/Projects/.NET/Assembly/Use/Probing/Strong/Strong.DLL.
LOG: Attempting download of new URL
file:///E:/Projects/.NET/Assembly/Use/Probing/Strong.EXE.
LOG: Attempting download of new URL
file:///E:/Projects/.NET/Assembly/Use/Probing/Strong/Strong.EXE.
```

Изучение последних строк лога показывает, что среда исполнения пыталась загрузить сборку из дополнительных подкаталогов приложения, а также из файлов с другими именами. Этот процесс называется зондированием, от перевода английского слова probing.

Во время зондирования среда исполнения учитывает значение параметра `privatePath`, тега `probing`. Следовательно, в этом параметре необходимо лишь указать каталог `asm`. Конфигурационный файл будет выглядеть следующим образом:

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

После создания этого файла в каталоге приложения, загрузчик среды исполнения без труда найдет необходимую сборку и запустит приложение.

<publisherPolicy>

Прототип тега:

```
<publisherPolicy apply="yes|no"/>.
```

Разработчики среды исполнения .NET предусмотрели возможность простого распространения исправлений для приложений при помощи политики издателя. Политика позволяет автоматически заменять сборки на их новые версии. Тег обеспечивает принудительную отмену использования политики издателя к приложению. Для этого его необходимо лишь поместить в конфигурационный файл со значением параметра `apply`, равным `no`.

<qualifyAssembly>

Прототип тега:

```
<qualifyAssembly partialName="PartialAssemblyName" fullName=
"FullAssemblyName"/>.
```

Этот тег позволяет ассоциировать сокращенное имя сборки с полным строгим именем для удобства использования в программе. Параметры тега имеют следующий смысл:

- `partialName` — сокращенное имя сборки;
- `fullName` — полное строгое имя сборки.

К примеру, сделаем ассоциацию для сборки `Strong` (листинг 4.17).

Листинг 4.17. Создание псевдонима для строгого имени сборки

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <qualifyAssembly partialName="Strong"
                          fullName=
"Strong,version=1.0.0.0,culture=neutral,publicKeyToken=51f6c89de8953887"/>
      </assemblyBinding>
   </runtime>
</configuration>
```

Теперь везде в приложении можно использовать сокращенное имя `Strong`, вместо полного строгого имени. После введения связи, вызов

```
Assembly.Load("Strong");
```

будет рассмотрен средой исполнения как

```
Assembly.Load("Strong,version=1.0.0.0,culture=neutral,
publicKeyToken=51f6c89de8953887");
```

<dependentAssembly>

Этот тег позволяет управлять политикой загрузки персонально для каждой сборки. Сам он является пустым и служит лишь для объединения подтегов: `<assemblyIdentity>`, `<BindingRedirect>` и `<CodeBase>`. Первый из них является обязательным. Он указывает строгое имя сборки, к которой будут применяться теги `<BindingRedirect>` и `<CodeBase>`.

<assemblyIdentity>

Прототип тега:

```
<assemblyIdentity
   name="assembly name"
   publicKeyToken="public key token"
   culture="assembly culture"/>.
```

Параметры имеют следующий смысл:

- `name` — символьное имя сборки;
- `publicKeyToken` — маркер открытого ключа сборки;
- `culture` — региональный идентификатор сборки.

Для каждой сборки, которой необходимо задать дополнительные параметры загрузки, нужно создать тег `<dependentAssembly>`, в который надо вложить тег `<assemblyIdentity>`, идентифицирующий сборку. Пример использования тега приведен в листинге 4.18.

<bindingRedirect>

Прототип тега:

```
<bindingRedirect
   oldVersion="old assembly version"
   newVersion="new assembly version"/>.
```

Этот тег позволяет принудительно изменить политику управления версиями необходимой сборки. Вы указываете старую версию сборки и новую, которая будет подгружаться вместо нее (листинг 4.18).

Листинг 4.18. Изменение политики управления версиями для сборок

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

```
                                publicKeyToken="51f6c89de8953887"
                                culture="neutral" />
            <bindingRedirect oldVersion="1.0.0.0"
                             newVersion="2.0.0.0"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

Теперь, если приложение запросит сборку `Strong` версии 1.0.0.0, то, ничего не зная об этом, получит сборку версии 2.0.0.0.

<codeBase>

Этот тег является наиболее интересным из описываемых. Его прототип:

```
<codeBase
   version="Assembly version"
   href="URL of assembly"/>.
```

Он позволяет задавать произвольное местоположение зависимой сборки. При этом его параметры играют следующую роль:

- `version` — версия сборки. Она задается здесь, поскольку тег `<assemblyIdentity>` не позволяет указать версию сборки;
- `href` — адрес местоположения сборки в формате URL.

Причем в качестве значения параметра `href` может быть указан сетевой адрес. Продемонстрируем это на примере (листинг 4.19).

Листинг 4.19. Загрузка сборки с удаленного сервера

```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>
            <assemblyIdentity name="Strong"
                              publicKeyToken="51F6C89DE8953887"/>
            <codeBase version="1.0.0.0"
                      href="http://lexa:81/one/Strong.dll"/>
         </dependentAssembly>
      </assemblyBinding>
   </runtime>
</configuration>
```

Этот конфигурационный файл предписывает среде исполнения загружать сборку `Strong` с адреса **http://lexa:81/one/Strong.dll**, по протоколу `http`, через 81-й порт. Где `lexa` — это имя рабочей станции.

Проверка возможности сетевой загрузки

Для того чтобы протестировать работу конфигурационного файла, необходимо установить Web-сервер для локального тестирования или выложить сборку на один из серверов в сети Интернет. У меня на рабочей станции стоят два Web-сервера: IIS и Apache. Для того чтобы избежать конфликтов, они разнесены по разным портам — 81 и 82, соответственно. Именно поэтому указывался 81 порт в адресе ссылки на сборку. Ниже приведены краткие рекомендации по установке и настройке этих серверов.

Настройка IIS-сервера

IIS-сервер входит в стандартную поставку операционных систем класса Windows NT, начиная с 2000 версии. Правда, он может не входить в начальный пакет установки. Для того чтобы его инсталлировать, вам придется воспользоваться мастером добавления дополнительных компонент Windows из панели управления (**Add or Remove Programs->Add/Remove Windows Components**). После установки сервера в корневом каталоге диска C: появится папка `IntePub`. В ней располагается подкаталог `wwwroot` — это корневая папка Web-сервера, файлы которой доступны по протоколу HTTP. В ней необходимо создать папку `one`, в которую поместить сборку `Strong.dll`. После произведенных манипуляций сборка будет доступна по одному из следующих адресов.

```
http://localhost/one/Strong.dll
http://localhost:80/one/Strong.dll
http://имя_компьютера/one/Strong.dll
http://имя_компьютера:80/one/Strong.dll
```

Порт 80 можно не указывать, поскольку он по умолчанию используется при работе с протоколом `http`.

Также можно воспользоваться специальным апплетом управления Web-сервером ISS (**Control Panel->Administrative Tools->Internet Information Services**), который позволяет производить более тонкую настройку.

Настройка Apache-сервера

Apache-сервер настраивается несколько сложнее, чем IIS, правда, если вы будете следовать приведенным здесь рекомендациям, то проблем возникнуть не должно.

Примечание

Apache-сервер совершенно безвозмездно можно получить по следующему адресу: **http://apache.org/**. Не забудьте, что нужна версия для операционной системы Windows.

Этот Web-сервер не имеет интерактивной среды настройки, все параметры придется задавать при помощи специального конфигурационного сценария. Он располагается в файле: …\conf\httpd.conf, относительно корневого каталога сервера. Необходимо создать виртуальную связь папки на сервере, с каталогом на диске. Для этого добавим в текст этого файла следующую строку.

```
Alias /one "C:/Program Files/Apache Group/Apache2/one"
```

Первый параметр определяет имя виртуальной директории на сервере, а второй — путь до реальной папки на жестком диске.

Примечание

Каталог со сборкой может располагаться в любом месте жесткого диска, а не только в одном из подкаталогов Apache-сервера.

Естественно, в эту папку необходимо поместить сборку `Strong`.

Проверка возможности автоматической загрузки сборок из сети

После того как Web-сервер настроен и на нем располагается наша сборка, необходимо проверить возможность загрузки сборок по URL-адресу. Воспользуемся для этого одним из приложений, использующих сборку `Strong`, и конфигурационный файл, приведенный выше. Перед проверкой необходимо удалить сборку из каталога приложения, чтобы ее загрузка гарантированно происходила из сети.

Примечание

Для ссылок на локальный сервер, помимо его реального имени, вроде `lexa`, можно использовать специально зарезервированные имена. Это `localhost`, а также все IP-адреса из диапазона 127.0.0.1-127.0.0.255. То есть ссылки могут выглядеть следующим образом.

```
http://localhost/one/Strong.dll
http://127.0.0.1/one/Strong.dll
http://127.0.0.3/one/Strong.dll
http://127.0.0.255/one/Strong.dll
```

Чтобы задать URL-адреса для файла, располагающегося на жестком диске, воспользуйтесь псевдопротоколом `file://`. Например, таким образом.

```
file://C:\Strong.dll
```

При первом запуске приложения сборка будет скачана из сети по указному адресу и добавлена в кэш. Он располагается на вашем жестком диске по адресу:

```
"\Documents and Settings\Имя Текущей Учетной Записи\Local Settings\
Application Data\assembly\dl\".
```

При повторных запусках сборка будет загружаться уже с жесткого диска компьютера, что уменьшит сетевой трафик. Вы можете просмотреть список скачанных сборок, используя расширение оболочки и заглянув в папку `%WINDIR%\assembly\dowload`.

В заключение необходимо упомянуть об одном неприятном моменте. При указании сетевого URL-адреса избегайте скачивания сборки по протоколу `ftp`, поскольку в ранних версиях среды исполнения эта возможность попросту не работает. Причем запросы от среды исполнения к `ftp`-серверу исправно поступают, и даже происходит их скачивание (судя по логам ftp сервера), но подключение сборок по неведомым причинам не происходит.

Редактирование конфигурационных файлов при помощи встроенных средств среды исполнения

Конфигурационные файлы имеют формат XML, поэтому их редактирование при помощи обычных текстовых редакторов крайне неудобно. К тому же, необходимо либо помнить все теги и их прототипы либо постоянно иметь под рукой документацию.

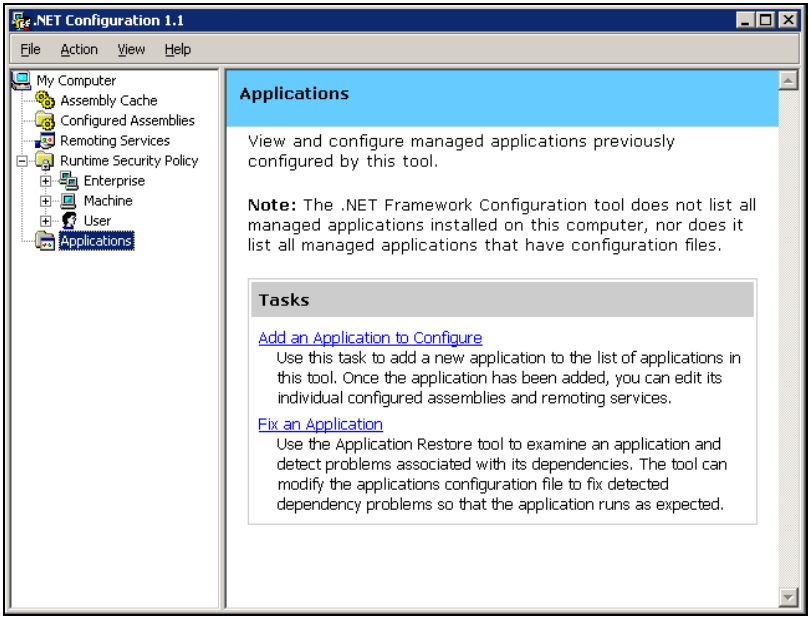


Рис. 4.27. Апплет .NET Framework Configuration, предназначенный для работы с конфигурационными файлами

Для тех, кого не устраивает редактирование конфигурационных файлов вручную, Microsoft создала специальный апплет (**Control Panel\Administration Tools\Microsoft .NET Framework Configuration**) (рис. 4.27).

Используя его, можно легко настроить все параметры, описанные ранее. Правда, существует одна особенность — этот апплет выполнен в виде оснастки консоли управления, которая присутствует только в системах класса NT. Поэтому пользователям операционных систем Windows 98 конфигурационные файлы придется редактировать вручную.

Процесс поиска сборок средой исполнения

Теперь, когда вы знакомы практически со всеми аспектами взаимодействия сборок в среде .NET, более детально рассмотрим процесс поиска сборок средой исполнения перед их непосредственной загрузкой. Этот процесс состоит из четырех этапов:

- анализ конфигурационного файла;
- проверка ранее загруженных сборок;
- поиск сборки в GAC;
- поиск файлов подчиненных сборок.

В первую очередь, после запроса приложения на загрузку сборки, среда исполнения производит поиск и анализ конфигурационного файла. Если конфигурационный файл будет найден, то среда исполнения скоординирует поведение своего загрузчика в соответствии с указанными в нем параметрами.

Перед тем как произвести поиск сборки на диске или в сети, среда исполнения проверит, а не была ли сборка уже загружена в среду. Этот процесс основан на механизме сравнения MVID, который позволяет точно идентифицировать сборку и ее модули.

Если сборка уже загружена, то среда исполнения просто устанавливает на нее соответствующие внутренние ссылки. Если нет, то среда исполнения приступает к следующему этапу поиска.

Если сборка строго именованная, то среда исполнения сначала пытается отыскать ее в GAC. Она обращается к менеджеру глобального хранилища сборок, расположенному в недрах динамической библиотеки `fusion.dll`. А он производит физический поиск сборки. Если это не привело к успеху, загрузчик среды исполнения попытается отыскать сборку в каталоге приложения.

Сначала поиск сборок будет производиться в основном каталоге приложения по реальному имени сборки. Если она не будет найдена, то загрузчик среды исполнения запустит процесс зондирования. Приведем отчет неудачной попытки поиска сборки `Strong.dll`

```
../Strong.DLL.
../Strong/Strong.DLL.
```

../Strong.EXE.
../Strong/Strong.EXE.

Достаточно интересным моментом здесь является то, что среда исполнения варьирует расширение файла от DLL до EXE.

4.7. Просмотр логов загрузки сборок и исключений при помощи утилиты *Fuslogvw.exe*

При загрузке приложений зачастую возникают проблемы. В случае консольного приложения полный отчет о произошедших исключениях выводится прямо на консоль. Но что делать, если приложение не имеет консоли — если оно оконного типа или это вообще Web-приложение. В этом случае, для просмотра лога проблем придется воспользоваться специальной утилитой Fuslogvw.exe, которая входит в поставку .NET Framework SDK от Microsoft. Вид окна этой утилиты приведен на рис. 4.28.

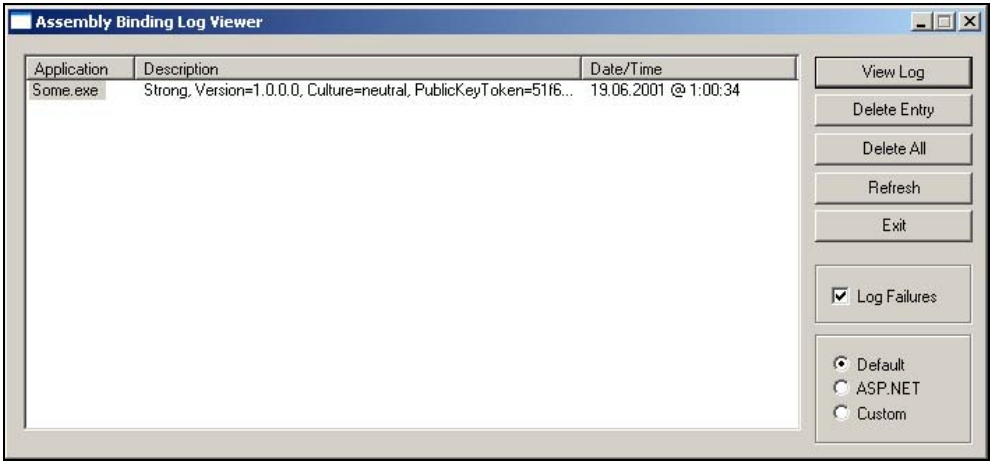


Рис. 4.28. Утилита Fuslogvw.exe, предназначенная для просмотров логов загрузки сборок

Примечание

По умолчанию запись отчетов отключена. Простым пользователям они не нужны и лишь будут занимать лишнее место на диске. Для того чтобы включить запись логов, необходимо установить значение параметра

HKLM\Software\Microsoft\Fusion\ForceLog

равным 1 (этот параметр имеет тип DWORD). Сделать это можно либо при помощи утилиты regedit или автоматически, прибегнув к помощи следующего reg-файла (листинг 4.20).

Листинг 4.20. Скрипт, включающий запись логов сбоев

```
REGEDIT4
;
;       Листинг 4.20
;       File:   Log.reg
;       Author: Дубовцев Алексей
;
 [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Fusion]
"ForceLog"="1"
```

После внесения необходимых изменений в реестр, станет возможным просмотр логов при помощи этой утилиты. Для того чтобы в списке главного окна утилиты отображались логи сбоев, необходимо установить галочку напротив переключателя **Log Failures**.

В листинге 4.21 приведен пример лога сбоя. Для удобства все комментарии вставлены прямо в его тело, в стиле C-подобных языков (// или /**/).

Листинг 4.21. Пример лога сбоя

```
// В конце этой строки указана дата и время сбоя, произошедшего при
// загрузке приложения.
*** Assembly Binder Log Entry  (19.06.2001 @ 1:00:34) ***
The operation failed.
// Указывается код причины сбоя и ее описание.
// Вы можете узнать описание кода самостоятельно,
// используя утилиту ErrorLookup из поставки Visual Studio,
// или воспользоваться функцией API FormatMessage.
Bind result: hr = 0x80070002. The system cannot find the file specified.
Assembly manager loaded from:
// Это путь к библиотеке, которая производила запись лога,
// в ней располагается менеджер, управляющий сборками.
C:\WINXP\Microsoft.NET\Framework\v1.0.3705\fusion.dll
// А это сбойное приложение.
Running under executable  E:\Projects\.NET\Assembly\Use\Some.exe
--- A detailed error log follows.
// Общая информация, известная до загрузки.
=== Pre-bind state information ===
// Идентификационные параметры сборки.
LOG: DisplayName = Strong, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=51f6c89de8953887
```

```
 (Fully-specified)
// Папка приложения.
LOG: Appbase = E:\Projects\.NET\Assembly\Use\
LOG: Initial PrivatePath = NULL
LOG: Dynamic Base = NULL
LOG: Cache Base = NULL
LOG: AppName = NULL
Calling assembly : Some, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null.
// А вот отсюда начинается поиск сборки.
// Просмотр путей разработки.
LOG: Processing DEVPATH.
LOG: DEVPATH is not set. Falling through to regular bind.
// Попытка загрузить конфигурационный файл приложения.
LOG: Attempting application configuration file download.
// Файл загружен.
LOG: Download of application configuration file was attempted from
file:///E:/Projects/.NET/Assembly/Use/Some.exe.config.
LOG: Found application configuration file
(E:\Projects\.NET\Assembly\Use\Some.exe.config).
// Никаких исправлений для этой сборки не найдено.
LOG: Publisher policy file is not found.
LOG: Host configuration file not found.
LOG: Using machine configuration file from
C:\WINXP\Microsoft.NET\Framework\v1.0.3705\config\machine.config.
// Обнаружена политика для загрузки следующей сборки.
LOG: Post-policy reference: Strong, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=51f6c89de8953887
// Попытка найти сборку в глобальном хранилище не удалась.
LOG: Cache Lookup was unsuccessful.
// Пытаемся скачать сборку из сети.
LOG: Attempting download of new URL ftp://localhost/osfdne/Strong.dll.
// Ничего не вышло.
LOG: All probing URLs attempted and failed.
```

Более подробно этот лог рассматриваться не будет, поскольку вся необходимая информация указана в комментариях.

В главе была изложена основная информация, которая может понадобиться при работе со сборками. Не расстраивайтесь, если вы что-то не усвоили с первого раза, поскольку сборки являются основой платформы .NET, следовательно, вы будете сталкиваться с ними на каждом шагу. И, так или иначе, постигните основные методы работы с ними.

Глава 5

Атрибуты

В главе рассказано о новом слове в мире информации о типах — атрибутах. Эта технология революционно изменила методы и подходы в создании новейших программ. С ее помощью стала возможным разработка многих, доселе казавшихся невозможных технологий. А главное, введение атрибутов позволило существенно упростить конечный программный код верхнего уровня, что делает многие современные технологии более доступными для широких масс. Любой программист обязан уметь работать с атрибутами, поскольку они стали неотъемлемой частью современного программного мира.

5.1. Немного истории

Информация о типах, частью которой являются атрибуты, всегда была камнем преткновения в программном мире. Сначала она была доступна только компиляторам и динамическое обращение к ней не допускалось. Считалось, что это сугубо служебная информация, которая не нужна простым программистам. Но с усложнением программного обеспечения и развитием технологий создания программного кода, становилось ясно, что это не так. Переломный момент настал в период бурного развития ООП (объектно-ориентированного программирования). Появились задачи, решение которых было невозможно без использования динамической информации о типах. Особенно необходимость в такой информации ощущалась при проектировании сложных библиотек классов. Для примера можно привести библиотеку MFC (Microsoft Foundation Classes). При ее разработке в компилятор еще не была введена поддержка RTTI (Run-Time Type Information, информация о типах во время исполнения), но она была крайне необходима при проектировании библиотеки. Поэтому разработчикам пришлось создать искусственную версию RTTI на основе макросов и служебных сервисов, которая, к сожалению, получилась очень громоздкой и неудобной. В итоге у создателей компиляторов не оставалось выбора — они были просто вынуждены добавить в свои разработки поддержку RTTI. Правда, необходимо отметить, что из-за своих недостатков повсеместного распространения эта технология не получила.

COM и атрибуты

Новые идеи в технологию информации о типах привнесла технология COM. При создании объектной модели COM стало ясно, что она нуждается в более мощной технологии информации о типах, чем существующие на тот момент. При разработке новой технологии информации о типах, был применен новаторский подход — помимо обычной информации о типах, программисты могли добавлять дополнительную информацию при помощи атрибутов.

В COM для идентификации программных элементов используются специальные уникальные идентификаторы GUID. Эти идентификаторы связываются с COM-классами и интерфейсами. Соответственно, чтобы запросить интерфейс или класс, необходимо воспользоваться связанным с ним идентификатором, а не с символьным именем. При разработке библиотек типов было предложено использовать атрибуты для связывания идентификаторов с программными элементами. Поскольку для описания библиотек типов создавался совершенно новый язык MIDL (Microsoft Interface Definition Language, язык описания интерфейсов от Microsoft), то никаких проблем с обратной совместимостью не возникало.

Приведем пример использования атрибутов в языке MIDL.

```
// Это и есть атрибут.
[uuid(DB01A1E3-A42B-11cf-8F20-00805F2CD064)]
interface ISomeInterface
{
        void Function1();
        void Function2();
};
```

Здесь `uuid` это и есть атрибут — они всегда задаются в квадратных скобках перед программным элементом, с которым должны быть связаны. При помощи атрибутов можно задать не только информацию об идентификаторе, язык MIDL вводит множество дополнительных атрибутов, при помощи которых можно указать различную информацию. Более того, существует возможность задавать пользовательские атрибуты. Делается это при помощи ключевого слова `custom`, которое позволяет связать с типом произвольную пользовательскую информацию. При этом доступ к пользовательскому атрибуту производится при помощи того же уникального идентификатора. Приведем пример.

```
custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, FunnyWorld, Version=0.0.0.0,
Culture=neutral, PublicKeyToken=9efab99be5760e0a)
```

Таким образом, указав идентификатор, можно получить дополнительную информацию.

Но технология атрибутов имела серьезный недостаток. Она не была обязательной при использовании COM, а использовалась как опция для автоматизации поддержки сервисов автоматизации. Как следствие, большинство программистов просто не знало о существовании этой возможности. К тому же, информация о типах, предоставляемых через библиотеки типов, была доступна только через низкоуровневые интерфейсы, в которых мало кто разбирался. Поскольку работа с низкоуровневыми интерфейсами была далеко не тривиальной задачей, использовать дополнительную информацию о типах никто не хотел, к тому же все прекрасно работало и без них.

Единственная задача, решение которой было невозможно без использования информации о типах, является маршализация. Здесь она попросту необходима, поскольку только через информацию о типах можно узнать размер и порядок передаваемых по сети данных. Но все сервисы маршализации были реализованы в недрах подсистемы COM и работали прозрачно для рядовых программистов. Им необходимо было лишь правильно настроить ряд необходимых сервисов, не касаясь при этом низкоуровневых тонкостей.

Тем не менее, технология поддержки информации о типах, заложенная в COM, послужила хорошим фундаментом для дальнейшего развития подобных технологий. Были выявлены основные проблемы и недостатки подхода, учтенные затем при разработке среды .NET.

Атрибуты в среде .NET

В среде .NET атрибуты получили беспрецедентное развитие. Разработчики довели эту технологию до совершенства, сделав ее гибкой и мощной. Возникает естественный вопрос, для чего нужны атрибуты и можно ли обойтись без их использования.

Необходимость атрибутов

Рассмотрим простой пример, демонстрирующий, для чего могут понадобиться атрибуты. Представим себе команду программистов, использующую самые современные методологии разработки программного обеспечения.

В соответствии с методологией, все классы должны разбиваться на группы надежности их реализации: ненадежные и надежные, которым сопоставлены красный и белый цвета. Цвета классам задаются в зависимости от того, каким программистом они созданы. К примеру, программист Вася опытный и классы, создаваемые им, будут помечаться как белые. А программист Коля пока еще не опытный, его классы будут помечены красным цветом.

"Зачем цвета? — спросите вы. — Не проще ли задавать имена разработчиков?" Ответ лежит в принципах новой методологии разработки программного обеспечения. В соответствии с ней необходимо избегать предвзятого мнения о членах команды и для этого нужно отказаться от использования

имен. Если Коля все-таки станет опытным программистом, то классы, помеченные его именем, все равно будут восприниматься в силу привычки как ненадежные. А новая методология позволяет избежать этой жизненной проблемы.

Для решения поставленной задачи воспользуемся пользовательскими атрибутами.

```
// Класс, разработанный опытным программистом, о чем говорит
// его белый окрас.
[ColorAttribute(white)]
class SomeClass
{
        void FirstFucntion()
        {
                ...
        }
                ...
};
// Класс, разработанный неопытным программистом, о чем говорит
// красный окрас.
[ColorAttribute(red)]
class SomeClass
{
        void FirstFucntion()
        {
                ...
        }
                ...
};
```

Таким образом, специалисты, тестирующие эти классы, будут знать, насколько им необходимо быть внимательными при тестировании того или иного класса. Здесь было наглядно продемонстрировано, как можно использовать атрибуты для решения повседневных проблем.

Сама методология является плодом фантазии автора — все совпадения случайны.

Примечание научного редактора

Заметим, что этот пример, конечно, не иллюстрирует основную идею атрибутов, а представляет собой довольно факультативный вариант. Основная идея заключается в следующем. Атрибуты не просто позволяют определить дополнительную информацию, связанную с элементом метаданных, а предоставляют

механизм для обращения к этой информации в ходе выполнения программы и, следовательно, для динамического изменения поведения программы на основе анализа этой дополнительной информации. Иными словами, атрибуты позволяют реализовать идею обращения к информации времени разработки в ходе выполнения управляемого кода. При этом процесс обращения к этой информации в ходе работы программы называется рефлексией, или отражением (reflection).

Атрибуты в стиле .NET

Рассмотрим более подробно механизм определения атрибутов. Для его исследования используем простейший атрибут `ObsoleteAttribute`. Он позволяет пометить программные элементы как устаревшие и служит лишь в информативных целях, не неся никакой особой нагрузки.

Примечание

В переводе с английского языка Obsolete означает устаревший, вышедший из употребления.

Применяется атрибут следующим образом.

```
[ObsoleteAttribute("This functon will not supplied in the future.")]
```

Возникает несколько вопросов:

- что представляет собой `ObsoleteAttribute`;
- каким образом он связывается с программным элементом;
- какая сущность будет связана с программным элементом при использовании атрибута.

Сам атрибут `ObsoleteAttribute` представляет собой обычный класс. Точнее, не совсем обычный — к классам атрибутов предъявляется строгое требование, они обязательно должны являться потомками класса `Attribute`. Сам класс `Attribute` особого интереса не представляет и будет рассмотрен несколько позже. Определение атрибута `ObsoleteAttribute` выглядит следующим образом.

```
public sealed class System.ObsoleteAttribute : Attribute
```

Этот класс, по сути дела, ничем не отличается от любых других, главное его достоинство в том, что он является потомком класса `Attribute`.

Фактически, ограничение, накладываемое общеязыковой спецификацией на происхождение класса атрибута от `Attribute`, является искусственным, поскольку явных технических ограничений со стороны среды исполнения не имеется. Теоретически в качестве атрибута может выступать абсолютно любой класс. Проблема появляется лишь на уровне внешних интерфейсов технологии отражения. Значения, возвращаемые всеми без исключения сервисами, работающими с атрибутами, имеют тип `Attribute`. Соответственно,

при возвращении атрибута, имеющего другой тип, произойдет нестыковка при преобразовании типов и будет выдано исключение `InvalidCastException`. Таким образом, получается, что объявить атрибут, не являющийся предком класса `Attribute`, можно, только вот получить доступ к нему при помощи стандартных сервисов технологии отражения не удастся. К тому же, создать подобный атрибут возможно только при помощи языка IL, компиляторы остальных высокоуровневых языков строго следят за тем, чтобы это правило не было нарушено.

Следующий интересный вопрос относится к механизму связывания атрибутов с программными элементами. Процесс прикрепления атрибута похож на создание нового объекта. На первый взгляд может показаться, что в метаданные помещается сам экземпляр атрибута, созданный во время компиляции. Однако это не так. В метаданные помещаются лишь упакованные параметры, которые были переданы конструктору атрибута.

```
[ObsoleteAttribute("This functon will not supplied in the future.")]
```

В этом случае, конструктору атрибута `ObsoleteAttribute` передается лишь одна строка, которая будет аккуратно сохранена в метаданных. В общем случае компилятор аккуратно упакует все значения и имена переданных параметров, а затем положит их в метаданные. Помимо этого, он прикрепит специальную ссылку к соответствующему программному элементу, которая будет указывать на сохраненные данные и предоставлять информацию о типе атрибута.

Естественно, что во время исполнения программы, атрибут, прикрепленный к любому программному элементу, может быть запрошен. Иначе их использование было бы бессмысленным. По первому требованию программиста информация об атрибуте будет извлечена из метаданных. После чего будет создан новый экземпляр типа атрибута при помощи параметров конструктора, упакованных компилятором в метаданные. Таким образом, пользователь получит экземпляр атрибута (объект), в котором будет располагаться вся информация, указанная программистом при определении атрибута. При этом прошу обратить внимание на то, что информация об атрибуте извлекается, только при непосредственном обращении к нему.

Теперь, когда мы разобрались с теоретической частью механизма работы атрибутов, можно приступить к более углубленному изучению вопроса.

5.2. Виды атрибутов

Все атрибуты в среде .NET можно условно разделить на четыре группы (рис. 5.1).

- Атрибуты, используемые компилятором.
- Атрибуты, используемые средой исполнения.

- Атрибуты, используемые библиотекой классов.
- Пользовательские атрибуты.

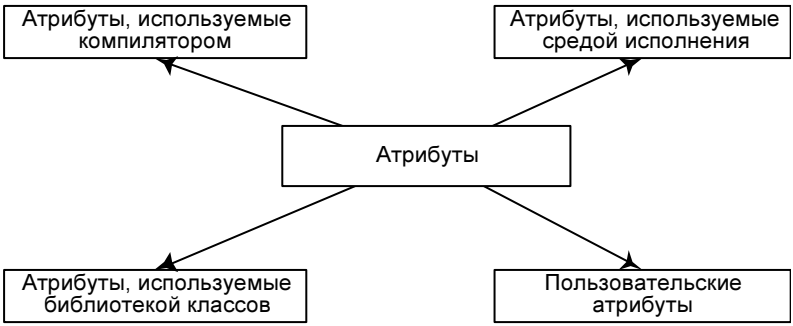


Рис. 5.1. Четыре вида атрибутов

Всего от типа `System.Attribute` унаследовано более 190 классов. Таким обширным количеством прямых потомков не может похвастаться ни один класс стандартной библиотеки, кроме базового класса `System.Object`.

Рассмотрим более подробно каждый вид атрибутов.

Атрибуты, используемые компилятором

Иногда их еще называют компиляторными атрибутами. Информация, предоставляемая этими атрибутами, используется компилятором при генерации конечного кода. В ходе анализа исходных текстов компилятора C# удалось получить список компиляторных атрибутов, который представлен ниже:

```
System.Security.Permissions.CodeAccessSecurityAttribute
System.Security.Permissions.SecurityPermissionAttribute
System.Security.UnverifiableCodeAttribute
System.Diagnostics.DebuggableAttribute
System.MarshalByRefObject
System.ContextBoundObject
System.Runtime.InteropServices.InAttribute
System.Runtime.InteropServices.OutAttribute
System.Attribute
System.AttributeUsageAttribute
System.AttributeTargets
System.ObsoleteAttribute
System.Diagnostics.ConditionalAttribute
System.CLSCompliantAttribute
```

```
System.Runtime.InteropServices.GuidAttribute
System.Reflection.DefaultMemberAttribute
System.ParamArrayAttribute
System.Runtime.InteropServices.ComImportAttribute
System.Runtime.InteropServices.FieldOffsetAttribute
System.Runtime.InteropServices.StructLayoutAttribute
System.Runtime.InteropServices.LayoutKind
System.Runtime.InteropServices.MarshalAsAttribute
System.Runtime.InteropServices.DllImportAttribute
System.Runtime.CompilerServices.IndexerNameAttribute
System.Runtime.CompilerServices.DecimalConstantAttribute
System.Runtime.CompilerServices.RequiredAttributeAttribute
```

Некоторые из этих атрибутов будут вставлены в конечный код, а некоторые из них уйдут в небытие, отслужив на стадии компиляции. Для примера можно привести атрибут `ComImportAttribute`. Встретив такой атрибут, компилятор пометит класс специальной директивой `import`, указывающей среде исполнения на то, что класс импортирован из среды COM. Но в код программы этот атрибут помещен не будет.

Среди перечисленных атрибутов присутствует уже упоминавшийся атрибут `ObsoleteAttribute`. Компилятор, обнаружив его, выдаст на консоль предупреждение, указанное в качестве единственного параметра конструктора при определении атрибута. Приведем пример присоединения такого атрибута к одному из методов основного класса приложения (листинг 5.1).

Листинг 5.1. Пример использования атрибута `ObsoleteAttribute`

```
/*
  Листинг 5.1
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Присоединим к одному из методов данного класса
  // атрибут ObsoleteAttribute.
  [ObsoleteAttribute("This functon will not supplied in the future")]
  public static void Function()
```

```
  {
    // Выведем на консоль приветствие.
    Console.WriteLine("Hello, World!");
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Используем нашу функцию для того,
    // чтобы увидеть предупреждение.
    // Если функция не будет использована, то предупреждение
    // на консоль выведено не будет.
    Function();
  }
};
```

Во время компиляции приложения на консоль выводится следующая строка.

```
warning CS0618: 'App.Function()' is obsolete: ' This functon will not
supplied in the future'
```

Атрибут никоим образом не влияет на работу приложения, он используется лишь компилятором. И, тем не менее, он будет встроен в код приложения. Это делается для перестраховки, поскольку код приложения может быть использован при создании других программ. Для того чтобы убедиться в этом, взглянем в IL-код созданной нами программы (рис. 5.2).

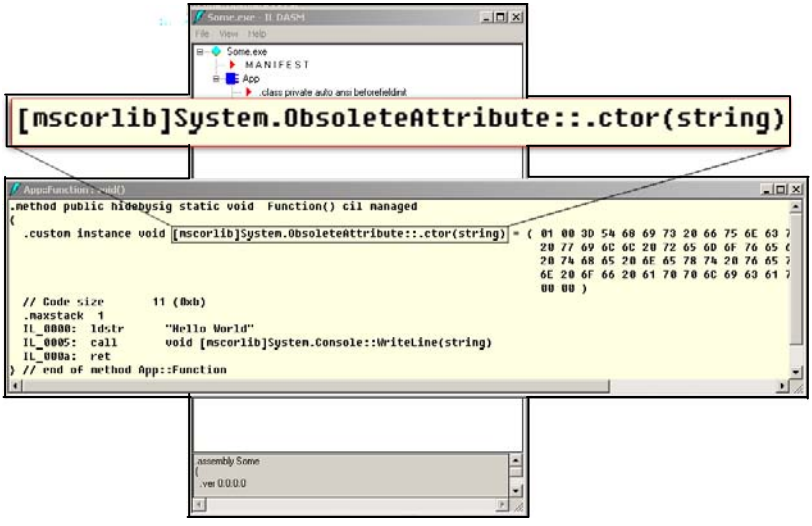


Рис. 5.2. Атрибут `ObsoleteAttribute` в конечном коде приложения

Обратите внимание на группу шестнадцатеричных чисел, изображенных справа на рисунке. Они представляют собой упакованные параметры конструктора атрибута. В этой строке закодировано сообщение "This functon will not supplied in the future". Можно упомянуть, что этот атрибут имеет дополнительный конструктор, заставляющий компилятор вместо предупреждения генерировать ошибку. Для этого необходимо передать в качестве второго параметра конструктора значение `true`.

```
[ObsoleteAttribute("Some", true)]
```

Атрибуты, используемые средой исполнения

Такие атрибуты используются виртуальной машиной среды исполнения в служебных целях. Их достаточно много, они влияют на ход исполнения уже откомпилированной программы.

К примеру, можно упомянуть атрибут `FieldOffset`, который указывает относительное смещение поля структуры. Во время создания структуры, при условии выполнения определенных требований, среда исполнения сканирует все ее поля на наличие атрибута, и если он будет найден, располагает их в памяти в соответствии с указанными значениями. Эти атрибуты позволяют декларативным способом настроить поведение среды исполнения, что более удобно и наглядно, чем обращение к дополнительным программным сервисам.

Атрибуты, используемые библиотекой классов

Атрибуты этого вида используются в служебных целях классами, входящими в состав стандартной библиотеки. Многие технологии, входящие в состав общей библиотеки, вводят не только собственные классы, но и атрибуты, позволяющие более гибко оперировать с типами.

Пользовательские атрибуты

К этой группе относятся атрибуты, созданные самими пользователями (программистами). Теперь любой пользователь может создавать собственные атрибуты и безгранично расширять информацию о типах в своих приложениях. Далее более подробно рассмотрим способы создания пользовательских атрибутов. Но прежде необходимо научиться использовать обычные атрибуты.

5.3. Создание пользовательских атрибутов

Возможности механизма поддержки атрибутов в среде .NET беспрецедентны. Программисты могут создавать собственные атрибуты, ничем не уступающие по возможностям системным. Более того, в среде .NET нет строгого различия между пользовательскими и системными атрибутами. С точки зрения программной модели, они являются абсолютно идентичными элементами.

Общеязыковая спецификация требует при определении пользовательских атрибутов заканчивать их имена словом `Attribute`. Правилу лучше всего неукоснительно подчиняться, поскольку это в дальнейшем существенно облегчит чтение кода и работу с пользовательскими атрибутами.

Рассмотрим простой пример. В нем будет определен пользовательский атрибут `DeveloperAttribute`, который позволит указать имя разработчика, создавшего программный элемент, и дату его создания. Код примера приведен в листинге 5.2.

Листинг 5.2. Пример создания пользовательского атрибута

```
/*
  Листинг 5.2
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Определим собственный класс атрибута.
// Обратите внимание, он является потомком класса Attribute.
// Еще раз напомню, что это обязательное требование
// к определению атрибутов.
sealed class DeveloperAttribute : Attribute
{
  // Введем служебные поля для хранения данных атрибута.
  // Здесь мы будем хранить имя разработчика.
  private string m_Name;
  // А здесь будет располагаться дата создания объекта.
  private string m_CreateTime;
  // Конструктор атрибута по умолчанию, то есть тот,
  // который обязательно должен всегда использоваться.
  public DeveloperAttribute(string name)
```

```
  {
    // Установим значение внутреннего поля.
    m_Name = name;
  }
    // Опишем общедоступные свойства, позволяющие получить доступ
  // к данным атрибута.
  public string Name
  {
    get
    {
      // Возвращаем реальное значение имени.
      return m_Name;
    }
  }
  public string CreateTime
  {
    get
    {
      // Возвращаем значение времени.
      return m_CreateTime;
    }
    set
    {
      // Устанавливаем значение времени.
      m_CreateTime = value;
    }
  }
};
// Применим атрибут к основному классу приложения.
[Developer("Алексей Дубовцев", CreateTime = "3,3,2012")]
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
  }
};
```

Пример будет успешно скомпилирован. Правда, толку пока от него мало, поскольку никакой функциональной нагрузки он на себе не несет. Но скоро мы научимся использовать прикрепленные к программным элементам атрибуты.

Особенности использования конструкторов атрибутов

Хотя приведенный ранее пример успешно компилируется, в нем есть одна неоднозначность, которая незаметна с первого взгляда. При определении конструктора было указано, что он имеет лишь единственный параметр, принимающий в строку имя разработчика. Приведем прототип единственного конструктора атрибута.

```
public DeveloperAttribute(string name)
```

Но при создании атрибута, конструктору передавалось два параметра.

```
[Developer("Алексей Дубовцев", CreateTime = "3,3,2012")]
```

Почему же заведомо неверный код был успешно обработан компилятором? Оказывается, что конструкторы атрибутов имеют одну неочевидную особенность: они могут принимать, так называемые, именованные параметры. Под именованными параметрами подразумеваются поля и свойства атрибута. Обратите внимание, при задании второго параметра явно указывалось имя свойства, которое должно принять этот параметр.

```
[Developer("Алексей Дубовцев", CreateTime = "3,3,2012")]
```

Таким образом, конструктор атрибута может принимать два вида параметров (рис. 5.3).

- Позиционные параметры — параметры, явным образом указанные в конструкторе атрибута.
- Именованные параметры — поля или свойства атрибута.

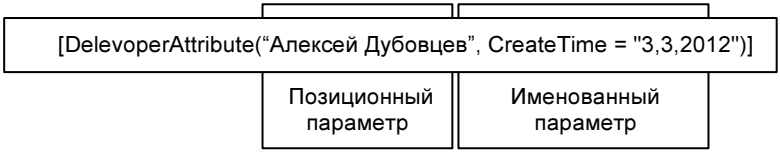


Рис. 5.3. Два вида параметров конструкторов атрибутов

При этом указание именованных параметров является опциональным. То есть их можно указывать по желанию, что весьма удобно. Возвращаясь к предыдущему примеру, можно было бы применить атрибут следующим образом.

```
[Developer("Алексей Дубовцев")]
```

При этом значение поля `CreateTime` остается на совести разработчика атрибута. При использовании и проектировании атрибута необходимо учитывать, что некоторые его поля могут быть не проинициализированы. А при определении (прикреплении) атрибутов необходимо помнить, что позиционные параметры должны присутствовать всегда и неразрывно стоять первыми в списке аргументов конструктора. Под неразрывностью подразумевается невозможность перемешивания именованных и позиционных параметров.

Именованные параметры — поля или свойства

В предыдущем примере в качестве именованных параметров использовались свойства. Но можно использовать и поля — принципиального различия в этих подходах нет. Продемонстрируем это, переписав пример с использованием полей (листинг 5.3).

Листинг 5.3. Поля как именованные параметры

```
/*
  Листинг 5.3
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Определим собственный класс атрибута.
// Обратите внимание, он является потомком класса Attribute.
// Еще раз напомню, это обязательное требование
// к определению атрибутов.
sealed class DeveloperAttribute : Attribute
{
  // Введем общедоступные поля, которые можно будет использовать
  // как именованные параметры.
  // Здесь будем хранить имя разработчика.
  public string Name;
  // А здесь будет располагаться дата создания.
  public string CreateTime;
  // Конструктор атрибута по умолчанию, то есть тот,
  // который обязательно должен всегда использоватся.
  public DeveloperAttribute(string name)
  {
    // Установим значение внутреннего поля.
```

```
    Name = name;
  }
};
// Применим атрибут к основному классу приложения.
[Developer("Алексей Дубовцев", CreateTime = "3,3,2012")]
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
  }
};
```

Определение атрибута стало более компактным, но зато потерялась возможность контролировать значения, передаваемые при создании атрибута. Напомним, что во время компиляции создание атрибутов не происходит, оно происходит только во время непосредственного запроса атрибута. Можно указать на одну интересную вещь, произошедшую после замены полей на события. Теперь, при использовании нашего атрибута, стало возможным определять имя разработчика двумя способами — через позиционный и через именованный параметры.

```
[Developer("Алексей Дубовцев", Name = "Василий", CreateTime = "3,3,2012")]
```

При использовании свойств этого не удалось бы сделать, поскольку свойство `Name` не предоставляет возможности устанавливать свое значение. Следует отметить, что именованный параметр будет "перекрывать" позиционный. В нашем случае имя разработчика, в конечном итоге, окажется "Василий". Среда исполнения сначала вызывает основной конструктор атрибута, а затем распаковывает остальные параметры и устанавливает значения соответствующих полей.

В заключение подраздела отметим, что для определения именованных параметров Microsoft рекомендует использовать свойства, а не поля.

Ограничение набора типов

Атрибуты являются частью системы информации о типах, которая задействована на всем протяжении работы управляемых приложений, поэтому к ним предъявляются строгие требования по производительности. Уже упоминалось, что информация, передаваемая при помощи атрибута, упаковывается и помещается в метаданные. А при необходимости распаковывается и помещается в экземпляр атрибута. Соответственно, к процессу распаковки предъявляются серьезные требования по производительности. Именно поэтому

было введено ограничение набора типов, которые могут являться полями атрибута и значения которых задаются при инициализации.

При определении атрибутов возможно использование следующих типов.

- Элементарные типы (`bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `string`).
- Тип `System.Type`.
- Перечисления (`enum`), имеющие общедоступный атрибут защиты.

Необходимо иметь в виду, что ограничение накладывается лишь на типы, передаваемые через конструктор атрибута. В служебных целях сам класс атрибута может использовать любые типы. Что естественно, поскольку в метаданные упаковываются только параметры конструктора атрибута.

Примечание

Не забывайте, что конструктор атрибута включает в себя позиционные параметры, передаваемые через функцию конструктор класса атрибута, а также именованные параметры, являющиеся свойствами класса атрибута.

Приведем пример, в котором атрибут `DeveloperAttribute` будет хранить время создания не в строковом виде, а при помощи класса `DateTime` (листинг 5.4). Его нельзя использовать для прямой передачи через конструктор.

Листинг 5.4. Использование типа `DateTime` для хранения информации об атрибуте

```
/*
  Листинг 5.4
  File:   Some.cs
  Author: Дубовцев Алексей */
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Определим собственный класс атрибута.
// Обратите внимание, он является потомком класса Attribute.
// Еще раз напомню, что это обязательное требование
// к определению атрибутов.
sealed class DelevoperAttribute : Attribute
{
  // Введем служебные поля для хранения данных атрибута.
  // Здесь будем хранить имя разработчика.
  private string m_Name;
   // А здесь будет располагаться дата создания.
```

```
  private DateTime m_CreateTime;
  // Конструктор атрибута по умолчанию, тот,
  // который обязательно должен быть всегда использован.
  // К сожалению, на параметры
  // функции конструктора накладываются существенные ограничения
  // в отношении типов и вместо того, чтобы передавать настоящий
  // экземпляр атрибута DateTime, придется передавать время
  // покомпонентно.
  public DelevoperAttribute(string name, int iYear, int iMonth, int iDay)
  {
    m_Name = name;
    // Внутри будем хранить значение времени, как полноценный
    // экземпляр типа DateTime.
    m_CreateTime = new DateTime(iYear,iMonth,iDay);
  }
  // Опишем общедоступные свойства, позволяющие получить доступ
  // к данным атрибута.
  public string Name
  {
    get
    {
      // Возвращаем реальное значение имени.
      return m_Name;
    }
  }
  public DateTime CreateTime
  {
    get
    {
      // Возвращаем значение времени.
      return m_CreateTime;
    }
    set
    {
      // Устанавливаем значение времени.
      m_CreateTime = value;
    }
  }
};
```

```
// Применим атрибут к основному классу приложения.
[Delevoper("Алексей Дубовцев", 2012,3,3)]
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
  }
};
```

С одной стороны, можно для наглядности ввести именованные параметры, позволяющие задавать дату. Но тогда теряется гарантия того, что дата создания будет задана полностью, поскольку именованные параметры могут быть опущены программистом. В этом случае становится не понятно, по какому алгоритму формировать дату создания.

5.4. Особенности использования атрибутов в различных языках

В классических Си-подобных языках, типа C# и MC++, атрибуты задаются при помощи квадратных скобок. Причем если атрибутов несколько, то их можно определить внутри одной скобки через запятую или каждый из них заключить в квадратные скобки.

```
[ComImportAttribute][ObsoleteAttribute("Hello, World!")]
class Hello
...
или так
[ComImportAttribute, ObsoleteAttribute("Hello, World!")]
class Hello
...
```

Оба способа правомерны и равноправны.

В других языках программирования, способы задания атрибутов существенно отличаются. Рассмотрим варианты задания атрибутов в некоторых популярных языках.

VB .NET

В этом языке атрибуты задаются при помощи угловых, а не квадратных скобок. К тому же они обязательно должны быть определены в одной строке с программным элементом, к которому они присоединяются. Для объедине-

ния строк в VB .NET используется символ подчеркивания (_), установленный в конце строки. Небольшой пример использования атрибута приведен в листинге 5.5.

Листинг 5.5. Пример использования атрибута на языке VB .NET

```
'
' Листинг 5.5
' File:   Some.cs
' Author: Дубовцев Алексей
'
' Подключим основное пространство имен общей библиотеки классов.
Imports System
' Основной класс приложения.
class App
  ' А это использование атрибута.
  ' ПРОШУ ОБРАТИТЬ ВНИМАНИЕ! На то, что атрибут
  ' должен быть расположен на одной строке с
  ' программным элементом, к которому он применен.
  ' В связи с этим, в конце строки стоит знак
  ' подчеркивания через пробел, который указывает
  ' компилятору, что строка еще не закончилась.
  <ObsoleteAttribute("This functon will not supplied in the future")> _
  Shared Sub Function1
    ' Выведем приветствие на консоль.
    Console.WriteLine("Hello, World!")
  End Sub
   ' Точка входа в приложение.
  Shared Sub Main
    ' Вызовем функцию для того, чтобы убедиться
    ' в том, что предупреждение выводится компилятором.
    ' Это позволит удостовериться, что атрибут
    ' был применен.
    Function1()
  End Sub
End Class
```

Понять применение треугольных скобок еще худо-бедно можно, но использование символа подчеркивания доставляет некоторые неудобства.

J#

Поскольку при проектировании языка перед разработчиками стояло требование сохранить совместимость с языком и средой Java, введение поддержки атрибутов вызвало немалые трудности. Но разработчикам удалось обойти их, хотя и не самым изящным способом. В этом языке для задания атрибутов используется специальный вид комментариев следующего формата.

```
/** @atribute ИмяТипаАтрибута() */
```

Такое странное сочетание символов в начале строки объясняется опасениями разработчиков случайного совпадения пользовательского комментария с форматом задания атрибута. И правда, предсказать, какие программист решит объявлять комментарии, довольно сложно.

Приведем пример, по структуре и функциональности аналогичный предыдущему (листинг 5.6).

Листинг 5.6. Пример использования атрибутов на языке J#

```
//Листинг 5.6
//File:   Some.cs
//Author: Дубовцев Алексей
// Подключим основное пространство имен общей библиотеки классов.
import System.*;
// Основной класс приложения.
public class App
{
  // Применим атрибут в стиле J#.
  /** @attribute Obsolete("This functon will not supplied in
the future") */
   public static void Function()
   {
     Console.WriteLine("Hello, World!");
   }
   // Точка входа в приложение.
   public static void main(System.String [ ] args )
   {
     Function();
   }
}
```

За более подробной информацией по использованию атрибутов в языке J# следует обратиться к документации, которая распространяется в составе MSDN.

IL

Определение атрибутов на этом низкоуровневом языке полностью отражает их внутреннюю сущность. Атрибуты задаются именно в том виде, в котором они хранятся в метаданных. Приведем краткий пример.

```
.method public hidebysig static void  Function() cil managed
{
    .custom instance void  [mscorlib]System.ObsoleteAttribute::.ctor(string) =
( 01 00 2D 54 68 69  73 20 66 75 6E D1 81 74 6F 6E   // ..-This fun..ton
    20 77 69 6C 6C 20 6E 6F 74 20 73 75 70 70 6C 69   //  will not suppli
    65 64 20 69 6E 20 74 68 65 20 66 75 74 75 72 65   // ed in the future
    00 00 )
```

Атрибут задается директивой `custom`. При ее использовании обязательно указывается полное имя типа атрибута, а также параметры в упакованном виде, которые необходимо распаковать и передать в экземпляр атрибута. Такой способ задания атрибутов практически полностью исключает возможность ручного задания атрибутов при написании IL-кода. Если, конечно, вы не разберетесь в формате упаковки параметров, который является общедоступным и описан в стандарте ECMA TC39/TG3, в разделе 22.3, документа Partition II Metadata.

Возможность неполного задания имени атрибута

Для удобства, при использовании атрибутов в языках высокого уровня, общеязыковая спецификация допускает задание имени не в полном формате. Разрешается опускать слово `Attribute`, которым обязаны заканчиваться имена всех атрибутов. К примеру, атрибут `ObsoleteAttribute` можно применять следующим образом.

```
[Obsolete("bla bla bla")]
```

Точно также можно поступать и с остальными атрибутами. Полное и сокращенное задания имени являются абсолютно идентичными, различия видны только на уровне исходного кода. В откомпилированном коде все равно будут заданы полные имена типов атрибутов. Вся забота по поддержке такой возможности ложится на компилятор.

Область применимости атрибутов

Атрибуты могут применяться к любым программным элементам среды .NET. Никаких технических ограничений на это не существует. Ниже приведен список всех программных элементов, к которым могут применяться атрибуты.

- Сборка
- Модуль

- Класс
- Конструктор
- Делегат
- Перечисление (`enum`)
- Событие
- Поле
- Интерфейс
- Метод
- Параметр (метода, функции)
- Возвращаемое значение (метода или функции)
- Свойство
- Структура

Зачастую такая функциональность является излишней. Применение многих атрибутов ограничено определенным кругом программных элементов с чисто смысловой точки зрения. Например, некорректно применять атрибут `GuidAttribute`, задающий уникальный идентификатор, к значению, возвращаемому методом. Это просто бессмысленно. Именно поэтому был создан механизм ограничения области применимости атрибута. Интересно, что сам механизм полностью построен на технологии атрибутов. Точнее сказать, он представляет собой атрибут.

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class AttributeUsageAttribute : Attribute
```

Применяя его при определении класса атрибута, можно задать область применимости.

Примечание

В языке C#, для того чтобы применить некоторый атрибут к сборке или модулю, необходимо воспользоваться ключевыми словами `assembly` или `module`.

```
[assembly: Атрибут()]
[module: Атрибут()]
```

5.5. Конструктор *AttributeUsage*

Конструктор атрибута принимает один параметр.

```
public AttributeUsageAttribute(
   AttributeTargets validOn
);
```

Параметр может принимать значение одного из флагов, являющегося членом перечисления `AttributeTargets`. Его описание можно найти в табл. 5.1.

Таблица 5.1. *Флаги перечисления* `System.AttributeTargets`

Член перечисления	Описание (Атрибут может применяться к ...)
Флаги	
`All`	любому программному элементу
`Assembly`	сборке
`Class`	классу
`Constructor`	конструктору
`Delegate`	делегату
`Enum`	перечислению
`Field`	полю
`Interface`	интерфейсу
`Method`	методу
`Module`	модулю
`Parameter`	параметру
`Property`	свойству
`ReturnValue`	возвращаемому значению
`Struct`	структуре

Комбинируя флаги перечисления при помощи логической операции "или" (|), можно регулировать область применимости атрибута к тем или иным программным элементам. К примеру, для того чтобы ограничить область применения атрибута классами, структурами и интерфейсами, необходимо использовать следующую комбинацию атрибутов.

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct |
AttributeTargets.Interface)]
```

В таком виде можно применять атрибут `AttributeUsage`, к любому классу атрибута.

AttributeUsage

Для полноты описания рассмотрим сам класс `AttributeUsage`, поскольку возможности, предоставляемые им, могут быть весьма полезны. В табл. 5.2 приведено краткое описание его членов.

Таблица 5.2. *Описание членов класса* `AttributeUsageAttribute`

Член класса	Описание
`AllowMultiple`	Определяет, может ли быть атрибут применен более одного раза к одному программному элементу. (значение по умолчанию `false`)
`Inherited`	Определяет, будет ли атрибут проецироваться на потомки данного программного элемента. Имеет смысл только по отношению к структурам, классам, интерфейсам или их методам и членам. (значение по умолчанию `true`)
`ValidOn`	Позволяет узнать, для каких программных элементов применен атрибут

Первые два свойства являются именованными параметрами конструктора атрибута. Последнее свойство предоставляет доступ к значению, переданному через позиционный параметр. Рассмотрим каждое из свойств более подробно.

AttributeUsage.AllowMultiple

Дает возможность применять один и тот же атрибут несколько раз к одному программному элементу.

```
public bool AllowMultiple {get; set;}
```

Рассматривая необходимость предоставления такой возможности с точки зрения введенного выше атрибута `DeveloperAttribute`, можно предположить ситуацию, когда класс разрабатывается не одним человеком, а несколькими. Тогда применение нескольких атрибутов кажется вполне целесообразным.

```
[Delevoper("Алексей Дубовцев", 2012,3,3)]
[Delevoper("Дмитрий Фозлеев", 2012,3,3)]
class App
{
        ...
```

или так

```
[Delevoper("Алексей Дубовцев", 2012,3,3),
 Delevoper("Андрей Шиповников", 2002,3,3)]
class App
{
        ...
```

Для того чтобы множественное применение атрибута `DeveloperAttribute` стало возможным, необходимо внести следующие изменения в определение атрибута.

```
[AttributeUsageAttribute(AttributeTargets.All, AllowMultiple = true)
sealed class DelevoperAttribute : Attribute
```

В противном случае осуществить задуманное не удастся.

AttributeUsage.Inherited

Соответственно, с его помощью можно контролировать действия на типы-потомки того программного элемента, к которому он был применен.

```
public bool Inherited {get; set;}
```

Рассмотрим действие этого свойства на примере атрибута `DeveloperAttribute`. Возьмем ситуацию, когда один класс, разработанный автором, был использован в качестве базового для разработки другого класса.

```
[Delevoper("Алексей Дубовцев", 2012,3,3)]
class App
{
};
class Hehe : App
{
};
```

Поскольку действие атрибута по умолчанию распространяется на всех потомков программного элемента, к которому он применен, получается, что разработчиком класса `Hehe` является Дубовцев Алексей, а это неверно. Таким образом, становится понятным, для чего может понадобиться ограничивать распространение атрибута на классы-потомки.

Для того чтобы осуществить это, необходимо изменить определение класса атрибута следующим образом.

```
[AttributeUsageAttribute(AttributeTargets.All, Inherited = false)
sealed class DelevoperAttribute : Attribute
```

Теперь действие атрибута будет ограничено лишь тем программным элементом, к которому он непосредственно применен.

AttributeUsage.ValidOn

Свойство позволяет только считывать свое значение.

```
public AttributeTargets ValidOn {get;}
```

Оно предназначено для получения информации об области применимости атрибута, заданной через позиционный параметр во время определения атрибута.

Получение информации об атрибутах

После того как мы научились использовать любые стандартные атрибуты и создавать собственные пользовательские атрибуты любой сложности, необходимо научиться получать информацию из них. Для решения этой задачи предназначен следующий метод.

```
// Возвращает атрибут заданного типа.
public static Attribute Attribute.GetCustomAttribute(
   // Программный элемент, от которого необходимо
   // получить заданный атрибут.
   MemberInfo element,
   // Тип атрибута, который мы хотим
   // получить.
   Type attributeType
);
```

Метод позволяет получить атрибут заданного типа, связанный с определенным программным элементом. Естественно, что получить необходимый атрибут удастся далеко не от любого программного элемента — операция выполнится только в том случае, если атрибут действительно к нему применялся.

Продемонстрируем работу данного метода на примере (листинг 5.7).

Листинг 5.7. Запрос атрибута, примененного к программному элементу

```
/*
  Листинг 5.7
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Опишем собственный пользовательский атрибут.
sealed class DelevoperAttribute : Attribute
{
  // Введем служебные поля для хранения данных атрибута.
  // Здесь будем хранить имя разработчика.
  private string m_Name;
  // А здесь будет располагаться дата создания.
  private DateTime m_CreateTime;
```

```
  // Конструктор атрибута по умолчанию, то есть тот,
  // который обязательно должен всегда быть использован.
  public DelevoperAttribute(string name, int iYear, int iMonth, int iDay)
  {
    m_Name = name;
    m_CreateTime = new DateTime(iYear,iMonth,iDay);
  }
    // Опишем общедоступные свойства, позволяющие получить доступ
  // к данным атрибута.
  public string Name
  {
    get
    {
      // Возвращаем реальное значение имени.
      return m_Name;
    }
  }
  public DateTime CreateTime
  {
    get
    {
      // Возвращаем значение времени.
      return m_CreateTime;
    }
    set
    {
      // Устанавливаем значение времени.
      m_CreateTime = value;
    }
  }
};
// Применим атрибут к основному классу приложения.
[Delevoper("Алексей Дубовцев", 2003,3,3)]
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
```

```
    // Запросим пользовательский атрибут типа DelevoperAttribute
    // от программного элемента типа App.
    DelevoperAttribute da = (DelevoperAttribute)
Attribute.GetCustomAttribute(typeof(App), typeof (DelevoperAttribute));
    // Проверим, был ли связан, необходимый нам атрибут, с
    // указанным программным элементом.
    if (da != null)
    {
      // Запрос атрибута удался, можно приступать
      // к получению информации из него.
      // Выведем имя человека, создававшего тип.
      Console.WriteLine("Deleveper Name = {0}", da.Name);
      // Выведем на консоль информацию о дате создания
      // программного элемента, заданную при помощи атрибута.
      Console.WriteLine("Create time = {0}", da.CreateTime.ToString());
    }
  }
};
```

В результате работы программы на консоль будут выведены следующие строки.

```
Deleveper Name = Алексей Дубовцев
Create time = 14.09.2002 0:00:00
```

Естественно, что получить информацию можно не только о собственных пользовательских атрибутах, но и о любых других. При этом необходимо учитывать, что некоторые из компиляторных атрибутов не встраиваются в конечный код приложения и получить доступ к ним будет принципиально невозможно.

Набор сервисов для работы с атрибутами не ограничивается одной функцией `Attribute.GetCustomAttribute`. На самом деле их огромное количество, большинство из которых будут описаны в *главе 12*, посвященной динамическому управлению типами.

5.6. Атрибуты. Что там внутри?

А теперь пришло время изучить внутренние механизмы работы атрибутов. Рассмотрим два важных аспекта: механизм определения атрибутов внутри метаданных и распределение объектов атрибутов во время исполнения приложений.

Механизм связывания атрибутов

Когда атрибут связывается с программным элементом, компилятор создает соответствующую запись в таблице метаданных. Запись указывает на то, что с программным элементом связан некий атрибут. Она представляет собой имя типа атрибута, прототип используемого конструктора, а также набор параметров в упакованном виде, которые необходимо передать в конструктор атрибута при создании его экземпляра.

Приведем пример описания атрибута `DeveloperAttribute` из предыдущего примера на языке IL.

```
// Имя типа атрибута, а также информация, необходимая для выбора
// подходящего прототипа, который будет использоваться при создании
// экземпляра атрибута.
.custom instance void DelevoperAttribute::.ctor(string,
                                                int32,
                                                int32,
                                                int32) =
// А это значение параметров конструктора в упакованном виде.
(       01 00 1F D0 90 D0 BB D0 B5 D0 BA D1 81 D0 B5 D0
        B9 20 D0 94 D1 83 D0 B1 D0 BE D0 B2 D1 86 D0 B5
        D0 B2 D2 07 00 00 07 00 00 00 0E 00 00 00 00 00 )
```

Примечание

Точное описание формата упакованных данных, содержащих значения параметров конструктора, приведено в документации, описывающей стандарт ECMA TC39/TG3 (документ Partition II Metadata, раздел 22.3). Здесь эта информация, в силу своей узкой специализации, рассматриваться не будет.

Перейдем к рассмотрению механизма загрузки атрибутов при подключении соответствующих программных элементов, с которыми они связаны. При загрузке любого программного элемента среда исполнения считывает связанные с ним атрибуты не полностью, а лишь ту часть, в которой хранится информация о типе атрибута и версии его конструктора. Таким образом, после загрузки программного элемента класса он будет выглядеть следующим образом (рис. 5.4).

Обратите внимание, что упакованные параметры не подгружаются и, тем более, не создается никаких экземпляров атрибута. Таким образом, ни за размер упакованных параметров атрибута, ни за время, необходимое на его создание, беспокоиться не стоит.

Но как же возможна работа с программным элементом, если атрибуты, которые связянны с ним, подгружаются не полностью, а соответственно не создаются их экземпляры? Перед тем как получить ответ на этот вопрос,

давайте вспомним, что такое атрибут? Это дополнительная информация о типах, — ключевым здесь является слово **дополнительный**. То есть это некоторая дополнительная информация, которая в общем случае не обязательна для функционирования данного типа.

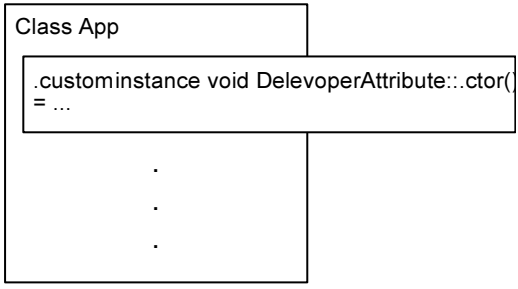


Рис. 5.4. Атрибут после загрузки класса, но до его явного запроса

Атрибуты запрашиваются для получения дополнительный информации о программном элементе, с которым он связан. Таким образом, если атрибут не будет запрошен, то запись о нем так и останется лежать мертвым грузом в таблице метаданных и использована не будет.

Если же все-таки запрос атрибута произошел, то среда исполнения распаковывает параметры его конструктора, создает экземпляр атрибута и передает его пользователю. То есть создание экземпляра атрибута происходит непосредственно при его запросе. Ни раньше ни позже. Это очень легко проверить, добавив в конструктор `DeveloperAttribute` контрольный вывод на консоль, который позволит засечь обращение к конструктору. Естественно очевидно, что создание экземпляра не может произойти в обход его конструктора, а посему мы всегда будем в курсе дел, происходящих внутри среды исполнения.

```
// Единственный конструктор атрибута, который обязательно будет вызван
// при создании экземпляра атрибута.
public DelevoperAttribute(string name, int iYear, int iMonth, int iDay)
{
  m_Name = name;
  m_CreateTime = new DateTime(iYear,iMonth,iDay);
  // Сообщим пользователю о том, что вызван метод — конструктор
  // атрибута, а соответственно и о том, что
  // создан новый экземпляр атрибута.
  Console.WriteLine("Привет, Дианка, я тебя обожаю");
}
```

Для полной уверенности модифицируем функцию точку входа (Main) в предыдущую программу, добавив в самое ее начало вывод приветственного сообщения на консоль. Это позволит убедиться в том, что экземпляр атрибута не создается до вызова функции Main.

```
// Точка входа в программу.
public static void Main()
{
  // Если я был не прав и атрибуты все-таки создаются
  // при загрузке приложения, то данное сообщение первым
  // на консоль выведено не будет.
  Console.WriteLine("This line must show on the console first");
  DelevoperAttribute da = (DelevoperAttribute)
Attribute.GetCustomAttribute(typeof(App), typeof(DelevoperAttribute));
  // Проверим, был ли связан необходимый нам атрибут
  // с указанным программным элементом.
  if (da != null)
  {
    // Запрос атрибута удался, можно приступать
    // к получению информации из него.
    // Выведем на консоль имя пользователя.
    Console.WriteLine("Deleveper Name = {0}", da.Name);
    // Выведем на консоль информацию о дате создания
    // программного элемента, заданную при помощи атрибута.
    Console.WriteLine("Create time = {0}", da.CreateTime.ToString());
  }
}
```

В результате работы приложения, на консоль будут выведены следующие строки:

```
This line must show on the console first
Hello World from DelevoperAttribute constructor
Deleveper Name = Алексей Дубовцев
Create time = 14.07.2002 0:00:00
```

Данный лог ясно подтверждает тот факт, что экземпляры атрибутов создаются лишь во время их запроса.

Примечание научного редактора

Читатель может недоумевать: но если я изменил состояние экземпляра атрибута, то, может, это именно то, что я хотел сделать? Поэтому, наверное, стоит еще раз подчеркнуть, что атрибуты представляют механизм использования информации времени разработки в ходе выполнения программы. Логично, что

сама первичная информация, описываемая атрибутом, не должна изменяться программой (хотя с экземпляром, находящимся в памяти, мы вольны делать что угодно): если атрибуты и являются машиной времени, то такой, которая курсирует только в одном направлении.

Каждому по атрибуту

Было бы логично предположить, что из соображений оптимизации, среда исполнения создает лишь один экземпляр при запросе атрибута, а затем уже просто передает ссылки на него, не прибегая к созданию новых объектов — экземпляров атрибута.

Оказывается, это предположение в корне не верно — среда исполнения создает новый экземпляр атрибута при каждом его запросе от программного элемента, с которым он связан! Таким образом, текущее количество экземпляров атрибутов будет точно соответствовать количеству его запросов (рис. 5.5).

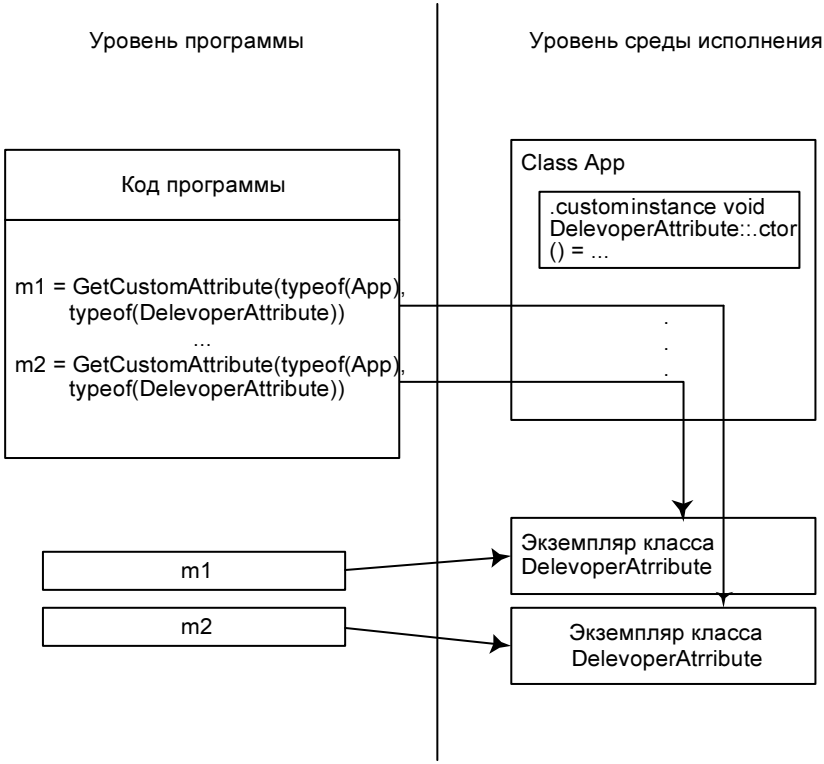


Рис. 5.5. Создание персонального экземпляра атрибута при его запросе

На схеме слева представлено два запроса `DeveloperAttribute`, которые помещают в переменные `m1` и `m2` ссылки на экземпляры данного атрибута. И хотя данные объектов, на которые указывают ссылки `m1` и `m2`, полностью идентичны, тем не менее, это различные объекты. С первого взгляда подобное транжирство памяти может показаться неоправданным. Однако такой подход позволяет гарантировать невозможность искажения данных атрибута внешними сервисами. Возвращаясь к нашей схеме, представим себе, что если бы экземпляр атрибута все-таки создавался один, и переменные `m1` и `m2` указывали на одну и ту же сущность. Тогда, изменив экземпляр атрибута при помощи одной из переменных, мы бы исказили общие данные дополнительной информации о типах, предоставляемой через атрибут, и при всех последующих запросах мы бы получали неверные данные.

Пускай вас не удивляет, что можно изменить значения полученного атрибута, ведь он является обычным классом, а нам попросту предоставляют его экземпляр. К тому же, с точки зрения безопасности, это, конечно же, является не допустимым. Именно поэтому, при каждом запросе атрибута создается его новый экземпляр.

Для того чтобы продемонстрировать это, приведем пример (листинг 5.8), который будет изменять значение, сохраненное в экземпляре атрибута после его получения, а затем получать его по новой и выводить информацию на консоль. Таким образом, если экземпляр создается лишь один раз, то повторный запрос атрибута представит ложные данные.

Листинг 5.8. Попытка изменения данных, предоставляемых атрибутом

```
/*
  Листинг 5.8
  File:   Some.cs
  Author: Copyright (C) 2002 Dubovcev Aleksey
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Опишем собственный пользовательский атрибут.
sealed class DelevoperAttribute : Attribute
{
  // Введем служебные поля для хранения данных атрибута.
  // Здесь мы будем хранить имя разработчика.
  private string m_Name;
  // А здесь будет располагаться дата создания.
  private DateTime m_CreateTime;
  // Конструктор атрибута по умолчанию,
```

```
  // который обязательно должен быть всегда использован.
  public DelevoperAttribute(string name, int iYear, int iMonth, int iDay)
  {
    m_Name = name;
    m_CreateTime = new DateTime(iYear,iMonth,iDay);
  }
    // Опишем общедоступные свойства, позволяющие получить доступ
  // к данным атрибута.
  public string Name
  {
    get
    {
      // Возвращаем реальное значение имени.
      return m_Name;
    }
    // ВОТ ЗДЕСЬ я добавил метод, позволяющий
    // изменять свойство Name.
    set
    {
      m_Name = value;
    }
  }
  public DateTime CreateTime
  {
    get
    {
      // Возвращаем значение времени.
      return m_CreateTime;
    }
    set
    {
      // Устанавливаем значение времени.
      m_CreateTime = value;
    }
  }
};
// Применим атрибут к основному классу приложения.
[Delevoper("Алексей Дубовцев", 2003,3,3)]
class App
```

```
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Осуществим первый запрос атрибута, после
    // которого сразу попытаемся изменить его значение.
        DelevoperAttribute da = (DelevoperAttribute)
Attribute.GetCustomAttribute(typeof(App), typeof(DelevoperAttribute));
    // Проверим, удался ли запрос атрибута.
    if (da != null)
    {
      // Изменим одно из полей атрибута.
      da.Name = "Some programmer";
      // Затем выведем на консоль информацию,
      // предоставляемую атрибутом, для того чтобы
      // убедиться в том, что информация была модифицирована.
      Console.WriteLine("Deleveper Name = {0}", da.Name);
      Console.WriteLine("Create time = {0}", da.CreateTime.ToString());
    }
    // Повторно запросим атрибут.
      da = (DelevoperAttribute) Attribute.GetCustomAttribute(↳
typeof(App), typeof(DelevoperAttribute));
    // Проверим, удался ли его запрос.
    if (da != null)
    {
      // Выведем информацию, предоставляемую атрибутом,
      // для того чтобы проверить, была ли она изменена
      // на предыдущем шаге работы приложения.
      Console.WriteLine("Deleveper Name = {0}", da.Name);
      Console.WriteLine("Create time = {0}", da.CreateTime.ToString());
    }
  }
};
```

В результате работы данного приложения на консоль были выведены следующие строки:

```
Deleveper Name = Some programmer
Create time = 14.07.2002 0:00:00
Deleveper Name = Алексей Дубовцев
Create time = 14.07.2002 0:00:00
```

Лог явно говорит о том, что изменение самого атрибута не удалось, получилось лишь изменить конкретный экземпляр атрибута.

Как видите, изменение первого экземпляра вполне удалось, но это никак не отразилось на втором запросе. Таким образом, подтверждается безопасность и устойчивость механизмов работы с атрибутами.

Класс *Attribute*

Как ни странно это бы звучало, но этот класс прикладному программисту необходим меньше всего. Поскольку для применения атрибутов он вообще не потребуется. Использовать его придется лишь тем, кто собирается разрабатывать свои атрибуты. Первый раз он понадобится при объявлении собственного пользовательского атрибута, для того чтобы наследоваться от него. Второй раз, когда будет необходимо запросить свой атрибут от некоторого программного элемента, к которому он может быть потенциально применен. Да и то, это можно сделать без использования данного класса, при помощи сервисов технологии отражения.

Тем не менее, класс `Attribute` придется рассмотреть, хотя и достаточно кратко.

Таблица 5.3. *Класс* `System.Attribute`

Член класса	Описание
Свойства	
`TypeId`	Возвращает уникальный идентификатор атрибута
Методы	
`GetCustomAttribute`	Возвращает атрибут заданного типа от указанного программного элемента
`GetCustomAttributes`	Возвращает несколько атрибутов заданного типа от указанного программного элемента
`IsDefaultAttribute`	Позволяет узнать, был ли инициализирован атрибут значениями по умолчанию
`IsDefined`	Позволяет узнать, был ли применен атрибут к заданному программному элементу

При работе с классом чаще всего вы будете пользоваться тремя статическими методами, которые позволяют получить информацию об атрибутах.

Attribute.GetCustomAttribute

Метод позволяет получить атрибут заданного типа, связанный с определенным программным элементом. Существует несколько перегруженных версий данного метода, позволяющих работать с различными программными элементами.

```
// Возвращает атрибут заданного типа, примененный к сборке.
public static Attribute Attribute.GetCustomAttribute(
  // Сборка, с которой, как мы предполагаем, связан атрибут.
  Assembly,
  // Тип атрибута, который мы хотим получить.
  Type
);
// Возвращает атрибут заданного типа, примененный к модулю.
public static Attribute Attribute.GetCustomAttribute(
  // Модуль, с которым, как мы предполагаем, связан атрибут.
  Module,
  // Тип атрибута, который мы хотим получить.
  Type
);
// Возвращает атрибут для программных элементов, представленных
// классом MemberInfo. Напомню вам, что это могут быть
// типы (классы, структуры, интерфейсы, перечисления), а также
// члены типов (поля, методы, свойства, события, вложенные типы)
public static Attribute Attribute.GetCustomAttribute(
  // От данного программного элемента мы хотим получить атрибут.
  MemberInfo,
  // Тип атрибута, который мы хотим получить.
  Type
);
// Возвращает атрибут, примененный к параметру метода.
public static Attribute Attribute.GetCustomAttribute(
  // Параметр, с которым, как мы предполагаем, связан атрибут.
  ParameterInfo,
  // Тип атрибута, который мы хотим получить.
  Type
);
```

Помимо перечисленных выше существует также группа методов, которая позволяет учитывать особенности перехода атрибутов по наследству. Данные

методы позволяют отсеять атрибуты, доставшиеся программным элементам от своих предков. Интуитивно понятно, что такая возможность имеет смысл только для типов, поскольку только они обладают свойством наследования друг от друга. Тем не менее, существуют методы, как бы позволяющие контролировать наследование атрибутов, даже для тех программных элементов, к которым это понятие принципиально не применимо. К примеру, можно привести параметры функций. На самом деле, такие методы введены лишь для формы и указания по поводу типа наследования начисто игнорируются их внутренним кодом.

```
// Приведенные методы аналогичны предыдущим за исключением того, что вы
// явно можете задать, хотите ли вы получать атрибуты, пришедшие из
// классов предков.
// Таким образом, если в качестве последнего параметра передать
// false, то будут рассматриваться только те атрибуты, которые
// непосредственно применены к данному программному элементу. Иначе
// будут возвращены все атрибуты, пришедшие по дереву иерархии наследования.
public static Attribute Attribute .GetCustomAttribute(
  Assembly,
  Type,
  bool
);
public static Attribute Attribute.GetCustomAttribute(
  MemberInfo,
  Type,
  bool inherit
);
public static Attribute Attribute.GetCustomAttribute(
  Module,
  Type,
  bool
);
public static Attribute Attribute.GetCustomAttribute(
  ParameterInfo,
  Type,
  Bool
);
```

При использовании всех версий метода, необходимо обязательно учитывать возможность множественного применения атрибута. То есть если атрибут одного и того же типа применен к программному элементу более одного раза. В этом случае, работа функции будет невозможна, поскольку под кри-

терий поиска будут подходить несколько атрибутов, а функция в силу своих конструкционных особенностей может вернуть лишь один. Если такое произойдет, метод выдаст исключение `AmbiguousMatchException`. Для работы с подобными атрибутами придется использовать метод `GetCustomAttributes`.

Attribute.GetCustomAttributes

Метод позволяет за один запрос получить сразу несколько атрибутов. При этом основной бонус метода заключается в том, что вам необязательно знать тип атрибута для того, чтобы получить его. Вы можете запросить все атрибуты, примененные к некоторому программному элементу. Использование метода очень удобно при динамическом исследовании кода.

```
// Следующие четыре метода возвращают список всех атрибутов, примененных
// к заданному программному элементу.
public static Attribute[] Attribute.GetCustomAttributes(
  Assembly
);
public static Attribute[] Attribute.GetCustomAttributes(
  MemberInfo
);
public static Attribute[] Attribute.GetCustomAttributes(
  Module
);
public static Attribute[] Attribute.GetCustomAttributes(
  ParameterInfo
);
// Следующие четыре метода аналогичны предыдущим, за исключением того,
// что позволяют указать, будут ли учитываться предки типа
// при поиске атрибута.
public static Attribute[] Attribute.GetCustomAttributes(
  Assembly,
  bool inherit
);
public static Attribute[] Attribute.GetCustomAttributes(
  MemberInfo,
  bool inherit
);
public static Attribute[] Attribute.GetCustomAttributes(
  Module,
  bool inherit
```

```
);
public static Attribute[]  Attribute.GetCustomAttributes(
  ParameterInfo,
  bool inherit
);
```

Продемонстрируем работу данного метода примером, который будет выводить список всех атрибутов, связанных с основным классом своего приложения (листинг 5.9).

Листинг 5.9. Сканирование всех атрибутов, связанных с заданным типом

```
/*
  Листинг 5.9
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// В данном пространстве имен определен необходимый нам атрибут Guid.
using System.Runtime.InteropServices;
// Применим к основному классу приложения
// два атрибута.
// Во избежание неясности скажу, что они были
// выбраны абсолютно случайно.
[
  Obsolete("Hoho"),
  Guid("11bb2f86-96cc-4ee4-9e70-5de1da5cd3ee")
]
// Основной класс приложения.
class App
{
  // Точка входа в приложения.
  public static void Main()
  {
    // Запросим все атрибуты, связанные с классом App,
    // и выведем на консоль информацию по каждому из них.
    foreach (Attribute a in Attribute.GetCustomAttributes(typeof(App)))
      Console.WriteLine(a.ToString());
  }
}
```

В результате работы приложения на консоль будут выведены следующие строки:

```
System.ObsoleteAttribute
System.Runtime.InteropServices.GuidAttribute
```

Как показывает лог, мы смогли получить все атрибуты, связанные с классом `App`, не указывая при этом их типы.

Также существуют перегруженные версии метода, позволяющие получить несколько атрибутов заданного типа.

```
// Эти методы позволяют получить атрибуты только определенного
// типа, использование методов имеет смысл, только если
// вы подозреваете, что к типу может быть несколько раз
// применен один и тот же атрибут.
public static Attribute[] Attribute.GetCustomAttributes(
  Assembly,
  Type
);
public static Attribute[] Attribute.GetCustomAttributes(
  MemberInfo,
  Type
);
public static Attribute[] Attribute.GetCustomAttributes(
  Module,
  Type
);
public static Attribute[] Attribute.GetCustomAttributes(
  ParameterInfo,
  Type
);
// А это наиболее расширенная версия функций запроса множественных
// атрибутов, она позволяет задать как тип атрибута, так и
// отфильтровать их с учетом наследования программных элементов.
public static Attribute[] Attribute.GetCustomAttributes(
  Assembly,
  Type,
  bool inherit
);
public static Attribute[] Attribute.GetCustomAttributes(
  MemberInfo,
```

```
  Type,
  bool inherit
);
public static Attribute[] Attribute.GetCustomAttributes(
  Module,
  Type,
  bool inherit
);
public static Attribute[] Attribute.GetCustomAttributes(
  ParameterInfo,
  Type,
  bool inherit
);
```

Эти методы просто незаменимы, когда приходится иметь дело с множественным связыванием атрибута. При помощи других модификаций метода работать с подобными атрибутами попросту невозможно.

Attribute.IsDefaultAttribute

Позволяет определить, инициализировался ли атрибут значениями по умолчанию.

```
public virtual bool Attribute.IsDefaultAttribute();
```

Здесь имеется в виду случай, когда при применении атрибута использовался конструктор без позиционных параметров, а именованные параметры заданы не были. Предполагается, что при определении собственных пользовательских атрибутов этот методов будет вами перекрыт. На деле же, он никем не используется и не перекрывается. В общей библиотеке классов он реализован следующим образом.

```
public virtual bool IsDefaultAttribute() {
            return false;
        }
```

Больше упоминаний о нем вы нигде, кроме документации, найти не сможете.

Attribute.IsDefined

Метод позволяет определить факт применения атрибута заданного типа к некоторому программному элементу.

```
// Следующие методы позволяют определить, связан ли с типом
// ( программным элементом) необходимый нам атрибут.
public static bool Attribute.IsDefined(Assembly, Type);
```

```
public static bool Attribute.IsDefined(MemberInfo, Type);
public static bool Attribute.IsDefined(Module, Type);
public static bool Attribute.IsDefined(ParameterInfo, Type);
// А эти четыре метода аналогичны предыдущим, за исключением
// того, что позволяют указывать признак наследования атрибута
public static bool Attribute.IsDefined(Assembly, Type, bool);
public static bool Attribute.IsDefined(MemberInfo, Type, bool);
public static bool Attribute.IsDefined(Module, Type, bool);
public static bool Attribute.IsDefined(ParameterInfo, Type, bool);
```

В основном метод используется для работы с атрибутами, не принимающими параметров при инициализации. При использовании таких атрибутов важен именно сам факт их применения, а не информация, которая передается с их помощью. Фактически, никакой информации, кроме своего существования, они не несут.

5.7. Концепция атрибутов

Теперь, когда мы полностью ознакомились с механизмами и принципами работы атрибутов, становится понятной вся концепция их идеологии. Атрибуты делают возможным декларативное программирование, при котором мы можем связывать с типами дополнительные данные. Где тип атрибута является ключом, по которому мы можем получить дополнительную информацию, связанную с типом. И в то же время тип атрибута служит механизмом для осмысления данной информации. В отличие от пользовательских атрибутов COM, мы получаем не просто набор данных, а объект, четко описывающий расположенную в нем информацию.

Здесь можно усмотреть развитие идей ассоциативного программирования с декларативным уклоном, причем в последние годы они развиваются уже не на уровне данных, а на уровне типов. Самое интересное заключается в том, что развитие наблюдается не только в управляемых средах, подобных .NET, но также и в классических языках разработки приложений, подобных C++.

Глава 6

Делегаты и события

В главе будут рассмотрены делегаты и события, которые играют немаловажную роль в жизни любой серьезной программы. Делегаты позволяют ссылаться на функции, описанные в программе, события служат для упрощения обработки тех или иных событий, происходящих во время работы программы. Оба механизма являются очень важными, но в то же время достаточно простыми и понятными.

Примечание научного редактора

Говоря более строго, основное назначение делегатов – обеспечение реализации механизма функций обратного вызова, необходимого в том случае, когда требуется косвенный вызов функции. В качестве стандартных примеров подобных ситуаций можно упомянуть:

1. Определение семантики сравнения элементов при использовании обобщенных алгоритмов сортировки и поиска или подобных им;
2. Установление взаимосвязи между функциями WinAPI и клиентским кодом при проектировании Windows-приложений (например, в оконных процедурах);
3. Проектирование классов, взаимодействующих друг с другом на основе модели программирования, ориентированной на события.

6.1. Роль событий в программах

В любой программе используются функции и методы, реализующие ее логику. Они вызываются при наступлении в ходе исполнения программы каких-либо событий. Все программы, независимо от языков разработки и среды управления, содержат функции, которые обрабатывают произошедшие события. Рассмотрим простейший пример программы, выводящей сообщение на консоль и содержащей функцию обработки события (листинг 6.1).

Листинг 6.1. Пример простейшей программы, содержащей событие

```
/*
  Листинг 6.1
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Выведем на консоль приветственное сообщение.
    Console.WriteLine("Hello, World!");
  }
}
```

На первый взгляд никакого события здесь нет. Однако это не так, в программе происходит событие запуска приложения и функция `Main` является его точкой входа. Фактически, работа приложения начинается вовсе не с функции `Main`, сначала среда исполнения должна загрузить сборку в память, подгрузить все необходимые для ее работы библиотеки, настроить рабочую среду, а уж затем произойдет запуск приложения. Для обработки этого события и будет вызвана функция `Main`.

В обработке событий, возникающих при работе программ, непосредственное участие принимают функции обратного вызова.

Примечание

Название "Функция обратного вызова" происходит от английского словосочетания Callback function. Такое название функции приобрели за то, что предназначены для обратной связи с внешними системами.

Эти функции вызываются извне программы, внешними сервисами, организуя с ними обратную связь. Ярким примером такой функции является `Main`. Не удивляйтесь — это действительно так. Она удовлетворяет всем критериям функции обратного вызова.

- Функция реализована в самой программе.
- Но вызывается извне, то есть не самой программой, а кем-нибудь другим.

И действительно, функция вызывается средой исполнения для обработки события запуска приложения.

Таким же образом обстоят дела и в неуправляемой среде, такое положение дел наблюдается везде, вне зависимости от среды программирования и типа операционной системы. В рассмотренном примере с функцией `Main` есть одно узкое место. А именно, данная функция известна заранее и система заранее знает, куда за ней обращаться. Указатель на функцию всегда расположен в манифесте сборки, откуда он будет считан загрузчиком и передан системе.

> ***Примечание научного редактора***
>
> Коротко, механизм управления исполнением кода, ориентированный на события, – это механизм, обеспечивающий взаимные уведомления программных объектов и привязку этих уведомлений к функциям-обработчикам. Если этого взаимного уведомления нет, то использование термина "событие" является безусловной натяжкой. Даже при использовании функций обратного вызова вовсе не всегда используются события, а вот события всегда связаны с функциями обратного вызова. Так, в CLR модель событий, безусловно, основывается на делегатах, при этом использование делегатов вовсе не обязательно подразумевает “events oriented programming”.

Рассмотрим другой, более абстрагированный пример, — обработку сообщений Windows. Во многих источниках наивно указано, что от Windows поступают сообщения вида `WM_что-то_там`, но редко где говорится, что в основе механизма сообщений Windows лежат все те же функции обратного вызова. Перед тем как получить события от Windows, необходимо зарегистрировать специальную функцию обратного вызова, которая и будет получать эти самые сообщения.

```
LONG FAR PASCAL MainWndProc(  HWND hWnd,
    // Вот оно сообщение Windows.
    unsigned message,
    WORD wParam,
    LONG lParam)
```

Они, по сути дела, являются вовсе не сообщениями, а кодами, передаваемыми функциям обратного вызова. Для нас интересным здесь являются не сами сообщения, а функции, принимающие эти сообщения. Для того чтобы сообщения стали передаваться в программу, необходимо зарегистрировать функцию обратного вызова в системе. Для этого в неуправляемых приложениях использовался адрес функции, то есть число, указывающее местоположение кода функции в памяти. Зная его, внешний код может беспрепятственно обратиться к функции. У такого подхода есть два существенных недостатка.

- Не соблюдается безопасность типов.
- Отсутствие контроля вызовов со сторон системы безопасности.

Разберем каждый из пунктов более подробно. Указатель на функцию не несет никакой информации, кроме адреса функции, а информация о списке параметров функции и типе возвращаемого значения функции доступна только на уровне исходного кода. Как результат, не обеспечена возможность контроля корректности обращения к функции. Проверить соответствие параметров и аргументов попросту не представляется возможным. При программировании с использованием управляемого кода такой подход неприемлем, поскольку не обеспечивает безопасности типов.

Концепция делегатов позволяет преодолеть эту проблему, а также получить и некоторые дополнительные преимущества, обсуждаемые далее.

6.2. Введение в делегаты

Общие сведения

Делегаты являются ссылками на методы, инкапсулирующие настоящие указатели и предоставляющие удобные сервисы для работы с ними. Ссылки представляют собой объекты соответствующего типа. Все делегаты являются объектами типа `System.Delegate` или `System.MulticastDelegate`, который является производным от первого.

Уже к окончанию разработки среды исполнения, ее архитекторы пришли к выводу, что целесообразно использовать только класс `MulticastDelegate`, а класс `Delegate` является избыточным. Но, опасаясь серьезных архитектурных нестыковок, разработчики были вынуждены оставить класс `Delegate` в составе общей библиотеки классов. Фактически, класс `Delegate` является рудиментом, доставшимся по наследству из-за начальных архитектурных просчетов разработчиков среды .NET.

Различие между этими классами заключается в том, что экземпляры первого (делегаты) могут хранить лишь одну ссылку на метод, а экземпляры второго могут содержать сразу несколько ссылок на методы. Благодаря этому, можно подсоединять к одному делегату несколько методов, каждый из которых при единственном обращении к делегату будет вызваться по цепочке. Таким образом, из программы будет виден лишь один делегат, а за ним, на самом деле, будет скрываться несколько методов (рис. 6.1).

Эта возможность очень удобна для поддержки событий, поскольку позволяет без использования дополнительных механизмов присоединить к событию несколько функций обработчиков. Фактически, делегат представляет собой объект — черный ящик, скрывающий в своих недрах указатели на функции. Важно понять, что делегаты, по сути дела, ничем не отличаются от обычных пользовательских объектов. Главная их особенность состоит лишь в том, что они имеет поддержку со стороны среды исполнения. Об их свойствах, в от-

личие от обычных объектов, знают даже компиляторы, предоставляющие удобные специальные сервисы для работы с ними.

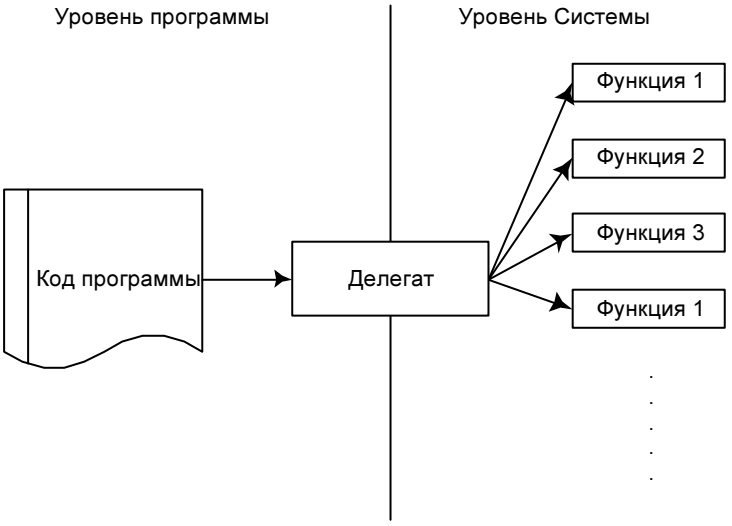


Рис. 6.1. Делегат может ссылаться на несколько методов или функций

Виды методов

Все методы в среде .NET можно разделить на две группы: статические (`static`) и экземплярные (`instance`).

Если делегат ссылается на статический метод, то все действительно просто. Так как в этом случае есть вся необходимая для вызова метода информация: адрес метода и параметры. Если же делегат ссылается на экземплярный метод, то задача усложняется. Для того чтобы вызвать метод, делегату необходимо знать ссылку на объект, к которому привязан данный конкретный метод. Оказывается, что эта ссылка хранится в самом объекте делегата и указывается при его создании. На протяжении всей жизни объекта делегата данная ссылка не изменяет своего значения, она всегда постоянна и может быть задана только при его создании. Таким образом, вне зависимости от того, ссылается ли делегат на статическую функцию или на экземплярный метод, обращение к нему извне ничем отличаться не будет. Всю необходимую функциональность обеспечивает сам делегат, вкупе со средой исполнения. Это очень удобно, поскольку множество разных делегатов можно привязывать к одному событию.

6.3. Делегаты – начинаем непосредственную работу

Создаем собственный делегат

Итак, делегат представляет собой экземпляр пользовательского класса, являющегося потомком класса `MulticastDelegate`. Соответственно, необходимо уметь объявлять подобные классы. Напрямую проделать операцию наследования от класса `MultiCastDelegate` или `Delegate` не получится, при попытке сделать это, компилятор выдаст следующую ошибку.

```
error CS0644: 'Hello' cannot inherit from special class 'System.Delegate'
```

В переводе на русский язык сообщение означает: "Не могу наследовать от специального класса `System.Delegate`". Наследование запрещает сам компилятор и только для этих классов, потому что в большинстве компиляторов предусмотрены специальные средства для работы с делегатами. В C# это ключевое слово **delegate**.

При описании делегата мы должны указать прототип метода, на который будут ссылаться экземпляры данного делегата. В общем виде описание делегата будет выглядеть следующим образом.

```
delegate void MyDelegate(string s);
```

Описывая делегат, необходимо понимать, что вводиться новый тип. А именно класс, являющийся потомком от `MulticastDelegate`. То есть конструкция, приведенная выше, на самом деле является следующей.

```
class MyDelegate : MulticastDelegate
{
   ...
  // Здесь специальным образом закодирован прототип метода, на который
  // мы собираемся ссылаться при помощи экземпляров делегата.
```

Компилятор проводит такую трансляцию неявно, дабы не пугать рядовых программистов ужасами внутренней архитектуры делегатов. Таким образом, для того чтобы воспользоваться делегатом, необходимо создать экземпляр данного класса, который бы указывал на нужный нам метод. Сам по себе делегат является типом и никаких ссылок на метод содержать не может, а соответственно, не может и использоваться для обращения к ним. Для этого необходимо использовать экземпляр делегата.

Создание экземпляра делегата просто, поскольку компилятор уже подготовил все необходимое. Он создал специальный конструктор, которому нужно передать указатель на метод, а со всем остальным разберется среда исполнения.

Создание экземпляра делегата происходит следующим образом:

```
MyDelegate del = new MyDelegate(MyHandler);
```

Где `MyDelegate` — это тип делегата, `del` — экземпляр делегата, который будет создан в результате выполнения конструкции, `MyHandler` — метод, на который будет ссылаться этот делегат. Соответственно, после создания экземпляра делегата можно обращаться к методам, на которые он ссылается. В языках высокого уровня существует возможность обращаться к экземпляру делегата, как к самому методу. Выглядеть это будет так.

```
del("Hello, world!");
```

После выполнения строки будет вызван метод `MyHandler`, на который ссылается наш делегат. В завершение приведем пример, который будет аккумулировать сказанное ранее и наглядно демонстрировать работу с простейшим делегатом.

Листинг 6.2. Простейший пример работы с делегатами

```
/*
  Листинг 6.2
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Опишем собственный делегат.
  // В этом случае — это не что иное, как описание
  // вложенного класса, просто оно замаскировано
  // при помощи специального ключевого слова языка.
  // Соответственно, для того чтобы использовать данный
  // класс, придется создать его экземпляр.
  delegate void MyDelegate(string s);
  // Это функция, которую мы будем использовать для проверки
  // работоспособности делегата.
  // Она является статической и для своей работы
  // не требует экземпляра объекта.
  static void MyHandler(string s)
  {
```

```
    // Просто выведем на консоль единственный аргумент,
    // переданный функции.
    Console.WriteLine(s);
  }
  // Точка входа в приложение.
  static void Main()
  {
    // Создадим экземпляр нашего делегата (типа! (класса!)),
    // описанного нами ранее, передав ему в качестве
    // параметра конструктора ссылку на функцию,
    // которую мы хотим связать с делегатом.
    MyDelegate del = new MyDelegate(MyHandler);
    // Вызовем функцию через делегат,
    // как видите, все достаточно просто, мы можем пользоваться
    // экземпляром делегата как функцией.
    del("Hello World");
  }
};
```

В результате работы приложения на консоль будет выведена следующая строка:

```
Hello, World!
```

Возникает естественный вопрос: "А к чему, собственно, такие сложности, не проще ли напрямую обратиться к методу `MyHandler`?" Оказывается, не проще. Хотя для данного примера, действительно, проще, поскольку мы точно знаем, какой метод нам нужно вызвать. Но для проектирования обратной связи с внешними системами делегаты незаменимы. Мы просто передаем внешней системе делегат, ссылающийся на наш внутренний метод. И покорно ждем, когда внешняя система обратится к нашей внутренней функции через переданный делегат.

Делегат и экземплярные методы

Теперь рассмотрим такой важный вопрос, как вызов экземплярных методов. Единственным отличием от обращения к статическим функциям будет способ создания делегата. Необходимо будет указать ссылку на объект, который будет использоваться при вызове метода через данный делегат.

```
MyDelegate del = new MyDelegate(sc.MyHandler);
```

Где `sc` является ссылкой на экземпляр объекта, который должен использоваться при обращении к методу `MyHandler`.

Приведем пример, демонстрирующий работу с подобными делегатами. Дабы излишне не запутывать читателя, а также упростить чтение листинга примера, в него введен дополнительный класс. В классе описан метод, на который будет ссылаться делегат. Данный метод будет использовать поле класса, что позволит нам следить за работой метода, изменяя его (листинг 6.3).

Листинг 6.3. Использование делегата для обращения к экземплярному методу

```
/*
  Листинг 6.3
  File:   Some.cs
  Author: Copyright (C) 2002 Dubovcev Aleksey
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем дополнительный класс, над методом которого будем
// экспериментировать.
class SomeClass
{
  // Поле класса.
  // Ни у кого не должно возникать сомнений, что оно
  // напрямую связано с экземпляром данного типа.
  public String SomeField;
  // Обычный метод, связанный с экземпляром типа.
  // Его будем вызывать при помощи делегата.
  public void MyHandler(string s)
  {
    // Выведем на консоль значение поля SomeField, а также
    // аргумент, переданный методу.
    Console.WriteLine(SomeField + s);
  }
};
// Основной класс приложения.
class App
{
  // Опишем собственный делегат.
  // В данном случае — это не что иное, как описание
```

```
  // вложенного класса, просто оно замаскировано
  // при помощи специального ключевого слова языка
  // Соответственно, для того чтобы использовать этот
  // класс, придется создать его экземпляр.
  delegate void MyDelegate(string s);
  // Точка входа в приложение.
  static void Main()
  {
    // Создадим экземпляр тестового класса.
    SomeClass sc = new SomeClass();
    // Создадим экземпляр делегата, содержащего,
    // помимо ссылки на сам метод, также ссылку
    // на объект, для которого будет вызван метод.
    MyDelegate del = new MyDelegate(sc.MyHandler);
    // Изменяем значение поля тестового объекта.
    sc.SomeField = "Hello, World!";
    // Вызовем метод через делегат.
    del(" - from Instance Delegate");
    // Эквивалентен следующему вызову
    // sc.MyHandler(" - from Instance Delegate");
    // Снова изменим значение поля тестового объекта.
    sc.SomeField = "Good bye, World!";
    // Снова обратимся к методу.
    del(" - from Instance Delegate");
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
Hello, World! - from Instance Delegate
Good bye, World! - from Instance Delegate
```

Видно, что метод действительно был вызван от нужного объекта. Изменения, внесенные в объект после первого обращения к методу, естественно, отразились на втором обращении к нему.

Теперь, после того как мы умеем пользоваться делегатами и знаем, для чего они нужны, приступим к их более детальному изучению.

6.4. *MulticastDelegate*

Этот класс является неотъемлемым в жизни любого делегата. Он предоставляет основные сервисы по управлению делегатами. Хотя многие из рассматриваемых здесь его членов все же описаны в базовом классе `Delegate`. Но в данном случае это можно отнести к конструкционным особенностям и не обращать на них внимания. Ниже, в табл. 6.1, приведено краткое описание наиболее важных членов класса `MulticastDelegate`.

Таблица 6.1. *Описание членов класса* `MulticastDelegate`

Член класса	Описание
Свойства	
`Method`	Возвращает метод, на который ссылается делегат
`Target`	Возвращает объект, к которому привязан метод, на который ссылается делегат
Методы	
`DynamicInvoke`	Позволяет динамически обратиться к функциям, связанным с делегатом
`GetInvocationList`	Возвращает список функций, привязанных к делегату
`Equality Operator`	Оператор (==), позволяет определить равенство делегатов
`Inequality Operator`	Оператор (!=), позволяет определить, различны ли делегаты
`Combine`	Добавляет в список вызова делегата ссылку на дополнительную функцию
`Remove`	Убирает из списка вызова делегата ссылку на указанную функцию
`CreateDelegate`	Позволяет динамически создать делегат

Обратите внимание, что многие из методов описаны как статические. Это очень важный момент, который мы рассмотрим несколько позднее.

MulticastDelegate.Method

Возвращает метод, на который ссылается делегат.

```
public MethodInfo Method {get;}
```

Особенность свойства состоит в том, что если делегат содержит не одну ссылку на функцию, а несколько, то свойство всегда будет возвращать последнюю добавленную ссылку.

MulticastDelegate.Target

Возвращает объект, с которым связан метод, на который ссылается делегат.

```
public object Target {get;}
```

Если делегат ссылается на статический метод, свойство вернет пустую ссылку (`null`). Особенность свойства состоит в том, что если делегат содержит не одну, а несколько ссылок на методы, то свойство вернет объект, связанный с последним из добавленных в очередь методов.

Пример использования свойств *Method* и *Target*

Для наглядной демонстрации возможностей рассмотренных свойств, приведем пример, использующий их (листинг 6.4). В нем будет объявлен простой тестовый класс (`SomeClass`), содержащий единственный экземплярный метод (`InstanceMethod`). В программе будет создан объект данного класса. Далее будет создан делегат и к нему присоединен метод `InstanceMethod`. Затем на консоль будет выведена информация о делегате, полученная при помощи свойств `Method` и `Target`.

Листинг 6.4. Пример использования свойств `Method` и `Target`

```
/*
  Листинг 6.4
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем дополнительный экспериментальный класс.
class SomeClass
{
  // Объявим в нем экземплярный метод.
  public void InstanceMethod()
  {
    // Сообщим пользователю о том, что метод вызван.
    Console.WriteLine("InstanceMethod was called - Hello, World!");
  }
```

```
};
// Основной класс приложения.
class App
{
  // Опишем делегат.
  // Обратите внимание, его прототип должен совпадать с прототипом
  // метода, на который он ссылается.
  delegate void MyDelegate();
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим экземпляр тестового класса.
    SomeClass sc = new SomeClass();
    // Создадим экземпляр делегата
    MyDelegate del = new MyDelegate(sc.InstanceMethod);
    // Выведем на консоль тип объект, к которому привязан метод,
    // на который ссылается делегат.
    Console.WriteLine("Target type = {0}",
    del.Target.GetType().ToString());
    // Выведем на консоль информацию по методу, на который ссылается
    // делегат.
    Console.WriteLine("Method = {0}",del.Method.ToString());
    Console.WriteLine();
    // Ну и для завершения картины вызовем сам делегат.
    del();
  }
}
```

В результате работы приложения на консоль будут выведены следующие строки.

```
Target type = SomeClass
Method = Void InstanceMethod()
InstanceMethod was called - Hello, World!
```

В случае, когда делегат содержит ссылку только на один метод, использование этих свойств крайне примитивно. Если же делегат содержит более одной ссылки, такой подход к получению информации годиться не будет. Необходимо будет воспользоваться другими способами. Оба свойства будут возвращать информацию только по последнему методу, добавленному в список вызова делегата.

При использовании свойства `Target` необходимо помнить, что статические методы не имеют привязки к объектам и значение этого свойства всегда будет равно нулевой ссылке.

MulticastDelegate.DynamicInvoke

Метод позволяет динамически обратиться к делегату.

```
public object Delegate.DynamicInvoke(
   // Аргументы, которые следует передать
   // при вызове функций, связанных с делегатом.
   object[] args
);
```

Практическая польза метода весьма сомнительна, тем не менее, приведем пример работы с ним. Чтобы разнообразить пример, попытаемся вызвать делегат с неверным количеством параметров и посмотрим, что получится (листинг 6.5).

Листинг 6.5. Динамическое обращение к делегату

```
/*
  Листинг 6.5
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Опишем делегат.
  delegate void MyDelegate();
  // А это подопытная функция-обработчик, которая будет связана
  // с делегатом.
  static void Handler()
  {
    Console.WriteLine("Handler method was called - Hello, World!");
  }
  // Точка входа в программу.
```

```
  public static void Main()
  {
    // Создадим экземпляр делегата.
    SomeDelegate sd = new SomeDelegate(Handler);
    // Сначала корректно обратимся к делегату.
    sd.DynamicInvoke(new object[0]);
    // Теперь вызовем делегат, с неверным количеством
    // параметров.
    sd.DynamicInvoke(new object[2]);
  }
};
```

В результате работы приложения сначала на консоль будет выведена строка.

```
Handler method was called - Hello, World!
```

Затем будет выведено сообщение о возникшем исключении `TargetParameterCountException`, со следующим объяснением

```
System.Reflection.TargetParameterCountException: Parameter count mismatch.
```

То есть нам сообщают, что при вызове делегата было передано неверное количество аргументов. Таким образом, при работе с методом `DynamicInvoke` необходимо иметь в виду, что могут произойти неприятности подобного рода.

Операторы сравнения (*Equality* и *Inequality*)

Операторы позволяют выяснить, ссылаются ли два делегата на одну и ту же функцию.

```
public static bool operator ==(
   Delegate d1,
   Delegate d2
);
public static bool operator !=(
   Delegate d1,
   Delegate d2
);
```

При сравнении делегатов учитывается не только метод, на который ссылается делегат, но и ссылка на объект, с которым связан метод. Если делегат будет содержать несколько ссылок на методы, тогда сравнение будет производиться для каждого из них.

MulticastDelegate.Combine и MulticastDelegate.Remove

Методы предназначены для поддержки делегатов, ссылающихся одновременно на несколько функций. Первый метод позволяет объединить несколько делегатов в один, в списке вызова которого будут находиться методы всех скомбинированных делегатов.

```
// Объединяет два делегата в один.
public static Delegate Delegate.Combine(
   // Первый делегат.
   Delegate,
   // Второй делегат.
   Delegate
);
// Объединяет произвольное количество делегатов в один.
public static Delegate Delegate.Combine(
   // Массив делегатов, которые будут
   // объединены в один.
   Delegate[]
);
```

Обращаем внимание на то, что в результате работы метода будет возвращен новый делегат, который в своем списке вызова будет содержать функции всех указанных делегатов. При этом старые делегаты при выполнении данной операции абсолютно не пострадают, их списки вызова останутся нетронутыми.

Наряду с возможностью объединения нескольких делегатов в один, существует обратная операция — изъятия указанных методов из списка вызова некоторого делегата.

```
// Позволяет изъять из делегата связь с определенными функциями,
// представленные также делегатами.
public static Delegate Delegate.Remove(
   // Делегат-источник.
   Delegate source,
   // Функции, связанные с этим делегатом,
   // будут изъяты из делегата-источника.
   Delegate value
);
```

Рассмотренный метод, также как и метод `Combine`, не изменяет значение параметров, переданных ему. Он создает абсолютно новый делегат, в который помещает результаты своей работы.

Разработчики среды .NET решили, что простым программистам будет затруднительно использовать эти методы, в силу их нетривиальности. Поэтому в компилятор C# были введены специальные средства поддержки делегатов. Вы можете оперировать с делегатами при помощи обычных операций сложения и вычитания (+,-,+=,-=). При этом компилятор автоматически будет генерировать код, обращающийся к методам `Combine` и `Remove`.

Для наглядности приведем пример, демонстрирующий работу с делегатами, ссылающихся на несколько методов (листинг 6.6).

Листинг 6.6. Работа с делегатами, ссылающимися на несколько методов

```
/*
  Листинг 6.6
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Описание делегата, необходимо понимать, что это
  // всего лишь описание класса, то есть типа.
  // Просто делается это при помощи специального ключевого слова
  // языка.
  // Для того чтобы использовать класс,
  // придется создать его экземпляр.
  delegate void MyDelegate(string s);
  // Это функция, на которую мы будем ссылаться при помощи
  // экземпляра делегата.
  static void MyHandler(string s)
  {
    // Просто выведем на консоль аргумент, полученный нашей
    // функцией.
    Console.WriteLine(s);
  }
  // Точка входа в приложение.
  static void Main()
  {
```

```
    // Создадим экземпляр делегата, описанного ранее.
    // В качестве единственного параметра конструктора
    // передадим ссылку на метод MyHandler.
    MyDelegate del = new MyDelegate(MyHandler);
    // Создадим новый делегат и скомбинируем его
    // с ранее созданным делегатом.
    del += new MyDelegate(MyHandler);
    // То же самое можно сделать следующим образом.
    del = del + new MyDelegate(MyHandler);
    // А теперь уберем из списка вызова делегата
    // одну из ссылок на метод MyHanlder.
    del -= new MyDelegate(MyHanlder);
    // Затем обратимся к конечному делегату, естественно,
    // что при этом по цепочке будут вызваны все методы,
    // связанные с ним.
    del("Hello, world!");
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
Hello, World!
Hello, World!
```

Обратите внимание, что функция была вызвана два раза, хотя мы добавляли ее в список вызова делегата три раза. Но если учесть, что мы один раз изъяли ее из списка вызова, то все становится понятным. Интересно, что в списке вызова делегата все методы одинаковые, но этот факт для сервисов, оперирующих с делегатами, не имеет никакого значения.

Компилятор преобразует арифметические операции над делегатами в обращения к методам `Combine` и `Remove`. Для того чтобы наглядно продемонстрировать это, приведем листинг функции `Main` на языке IL, который получился после транслирования исходного кода компилятором языка C# (листинг 6.7).

Листинг 6.7. Код функции `Main` на языке IL

```
/*
  Листинг 6.7
  File:   Some.il
  Author: Дубовцев Алексей
*/
```

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  .maxstack  4
  .locals init (class App/MyDelegate V_0)
  // Загружаем нулевой указатель — функция статическая,
  // не привязанная к экземпляру объекта.
  ldnull
  // Загружаем указатель на функцию.
  ldftn      void App::MyHandler(string)
  // Создаем экземпляр делегата.
  newobj     instance void App/MyDelegate::.ctor(object,
                                                 native int)
  stloc.0
  ldloc.0
  // Создаем еще один точно такой же делегат с одним
  // методом в списке вызова.
  ldnull
  ldftn      void App::MyHandler(string)
  newobj     instance void App/MyDelegate::.ctor(object,
                                                 native int)
  // Здесь как раз и происходит комбинирование делегатов,
  // на уровне языка C# это выглядело как оператор сложения (+=).
  call class [mscorlib]System.Delegate
             [mscorlib]System.Delegate::Combine(
                      class [mscorlib]System.Delegate,
                      class [mscorlib]System.Delegate)
  castclass  App/MyDelegate
  stloc.0
  ldloc.0
  // Создаем еще один точно такой же делегат с одним
  // методом в списке вызова.
  ldnull
  ldftn      void App::MyHandler(string)
  newobj     instance void App/MyDelegate::.ctor(object,
                                                 native int)
  // Здесь как раз и происходит комбинирование делегатов,
  // на уровне языка C# это выглядело как оператор сложения (+).
  call       class [mscorlib]System.Delegate
```

```
                    [mscorlib]System.Delegate::Combine(
                            class [mscorlib]System.Delegate,
                            class [mscorlib]System.Delegate)
  castclass  App/MyDelegate
  stloc.0
  ldloc.0
  // Создаем еще один точно такой же делегат с одним
  // методом в списке вызова.
  ldnull
  ldftn      void App::MyHandler(string)
  newobj     instance void App/MyDelegate::.ctor(object,
                                                 native int)
  // А здесь происходит изъятие метода из списка вызова делегата,
  // на уровне языка C# это выглядело как оператор вычитания (-=).
  call       class [mscorlib]System.Delegate
                    [mscorlib]System.Delegate::Remove(
                            class [mscorlib]System.Delegate,
                            class [mscorlib]System.Delegate)
  castclass  App/MyDelegate
  stloc.0
  ldloc.0
  ldstr      "Hello, world!"
  // А здесь происходит обращение к делегату.
  // Более подробно этот момент рассмотрим несколько позднее,
  // хотя, взглянув на следующую строку, вы можете обо всем
  // догадаться самостоятельно.
  callvirt   instance void App/MyDelegate::Invoke(string)
  ret
}
```

Понимая, что чтение IL-листингов может быть для некоторых читателей несколько утомительно, приведем код примера на языке C#, напрямую использующего методы `Combine` и `Remove` (листинг 6.8).

Листинг 6.8. Пример, явно использующий методы `Combine` и `Remove`

```
/*
  Листинг 6.8
  File:   Some.cs
  Author: Дубовцев Алексей
```

```
*/
// Подключим основное пространство имен.
using System;
// Основной класс приложения.
class App
{
  // Описание делегата, необходимо понимать, что это
  // всего лишь описание класса, то есть типа,
  // просто делается это при помощи специального ключевого слова
  // языка.
  // Для того чтобы использовать класс,
  // придется создать его экземпляр.
  delegate void MyDelegate(string s);
  // Это функция, на которую мы будем ссылаться при помощи
  // экземпляра делегата.
  static void MyHandler(string s)
  {
    // Просто выведем на консоль аргумент, полученный нашей
    // функцией.
    Console.WriteLine(s);
  }
  // Точка входа в приложение.
  static void Main()
  {
    // Создадим экземпляр делегата, описанного выше.
    // В качестве единственного параметра конструктора
    // передадим ссылку на метод MyHandler.
    MyDelegate del = new MyDelegate(MyHandler);
    // Создадим новый делегат и скомбинируем его
    // с ранее созданным делегатом.
    //del += new MyDelegate(MyHandler);
    del = (MyDelegate)Delegate.Combine(del,new MyDelegate(MyHandler));
    // То же самое можно сделать следующим образом.
    //  del = del + new MyDelegate(MyHandler);
    del = (MyDelegate)Delegate.Combine(del,new MyDelegate(MyHandler));
    // А теперь уберем из списка вызова делегата
    // одну из ссылок на метод MyHanlder.
    //del -= new MyDelegate(MyHanlder);
    del = (MyDelegate)Delegate.Remove(del,new MyDelegate(MyHandler));
```

```
    // А так мы вызовем функцию через делегат.
    // Как видите, все просто, мы можем пользоваться
    // экземпляром делегата как функцией.
    del("Hello, world!");
  }
};
```

Результат работы данного примера нисколько не будет отличаться от предыдущего.

```
Hello, World!
Hello, World!
```

Таким образом, использование методов `Combine`, `Remove` и арифметических операций, предоставляемых компилятором, по сути дела, ничем не отличается. Разве что обращение ко вторым более удобно, и программы, написанные с их использованием, читаются гораздо легче.

Delegate.Invoke или что там внутри?

Напомним, как выглядит механизм вызова делегата через обращение к методу `Invoke`.

```
ldstr      "Hello, world!"
callvirt   instance void App/MyDelegate::Invoke(string)
```

Программистам, использующим языки высокого уровня, это может показаться странным, поскольку на их уровне обращение к делегату происходит как к обычной функции.

```
del("Hello, World!")
```

Дело в том, что метод `Invoke` недокументирован. Он не является членом классов `Delegate` и `MulticastDelegate`. Соответственно, можно предположить, что это специальный метод, генерируемый компилятором. Для проверки предположения обратимся к IL-коду. Действительно, там обнаружится метод `Invoke`, его код приведен ниже.

```
.method public hidebysig virtual instance void
            Invoke(string s) runtime managed
{
}
```

Его изучение показывает, что метод лишен какого-либо кода. Он попросту пустой и ничего не может делать. Так может показаться, если не принимать во внимание спецификатор `runtime`, используемый при определении метода. Данный спецификатор указывает на то, что метод будет реализован по

ходу исполнения программы самой средой исполнения. Это означает, что код, обрабатывающий вызов метода, располагается в недрах самой среды исполнения. А она прекрасно осведомлена о внутреннем устройстве делегата и справится с вызовом всех его методов без дополнительной помощи со стороны программиста.

Помимо этого можно отметить, что конструктор каждого делегата описывается подобным образом. Для примера приведем код конструктора одного из делегатов.

```
.method public hidebysig specialname rtspecialname
        instance void  .ctor(object 'object',
                             native int 'method') runtime managed
{
}
```

Код, реализующий делегат, также находится внутри среды исполнения. Оказывается, метод `Invoke` отвечает не только за вызов делегата, но также за хранение информации о прототипе методов, на которые может ссылаться делегат. Не правда ли, изящное решение? Прототип закодирован в самом методе `Invoke`, то есть его прототип полностью совпадает с прототипом делегата. Таким образом, не надо прибегать к использованию дополнительных средств, вроде `MethodInfo`, для хранения информации о прототипе. К тому же, это страхует от возможности случайной передачи неверного количества или типа параметров при классическом обращении к делегату.

MulticastDelegate.GetInvocationList

Возвращает список функций, находящихся в очереди вызова делегата.

```
public sealed override Delegate[] GetInvocationList();
```

Достаточно оригинален способ возврата значений. Функция возвращает не список функций, а список делегатов. И неспроста, поскольку помимо ссылок на функции делегат может содержать ссылку на объект, с которым связан делегат. В случае получения только ссылки на функции или методы, часть информации могла быть потеряна.

Несмотря на кажущуюся абстрактность метода, он имеет очень важное практическое применение.

Существует один фундаментальный недостаток механизма вызова функций, связанных с делегатом. Точнее их даже два. И оба они связаны со случаем, когда в списке вызова делегата присутствует более одной функции.

- Функции, на которые ссылаются делегаты, могут возвращать значения. Но мы их получить не сможем, поскольку штатная функция вызова `Invoke` возвращает значение, которая вернула последняя из опрошенных функций.

- При обращении к делегату, содержащему в списке вызова несколько функций, вовсе не гарантируется, что все функции из списка будут вызваны. Если одна из них выбросит исключение, то работа метода `Invoke` будет прервана и остальные функции вызваны не будут.

Особое волнение вызывает вторая проблема, поскольку важные сообщения попросту могут не дойти до адресата. Что неприемлемо при разработке надежных систем. Рассмотрим пример (листинг 6.9).

Листинг 6.9. Демонстрация ненадежности делегатов, ссылающихся на несколько функций

```
/*
  Листинг 6.9
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Опишем делегат, с которым будем работать.
  // Напомню, что это всего лишь тип.
  delegate void MyDelegate(string s);
  // Первый метод, на который мы будем ссылаться при помощи
  // делегата. Именно он будет вызывать исключение,
  // не позволяющее обратиться ко второму методу.
  static void MyHandler(string s)
  {
    Console.WriteLine("Handler invoked");
    // А здесь произойдет исключение.
    throw new Exception();
  }
  // Второй метод, на который мы будем ссылаться из делегата.
  static void MyHandler2(string s)
  {
    // Если вдруг эта строка будет выведена, значит
    // предположения о ненадежности делегатов неверны.
    Console.WriteLine("Handler2 invoked");
  }
```

```
  // Точка входа в программу.
  static void Main()
  {
    // Создадим экземпляр описанного выше делегата.
    MyDelegate del = new MyDelegate(MyHandler);
    // Добавим к делегату еще одну ссылку на нашу функцию.
    del += new MyDelegate(MyHandler2);
    // Введем дополнительный защищенный блок, для того
    // чтобы отлавливать происходящие исключения.
    try
    {
      // Вызовем функции, связанные с делегатом.
      del("Hello, world!");
    }
    // Введем обработчик исключения, который предотвратит "падение"
    // программы.
    catch (Exception ex)
    {
      // Сообщим пользователю о том, что произошло исключение.
      Console.WriteLine("Problem! Exception caught");
    }
  }
};
```

В результате работы программы на консоль будут выведены следующие строки.

```
Handler invoked
Problem! Exception caught
```

Таким образом, исходя из информации, представленной в логе, нетрудно понять, что в список вызова делегата было добавлено два метода, но вызван был только первый из них. Надеюсь, причины такого поведения программы понятны.

Обойти проблему можно следующим образом. Для этого придется воспользоваться методами `GetInvocationList` и `DynamicInvoke`, чтобы организовать обращение к делегатам полностью в ручном режиме. Первый метод понадобиться для ручного перечисления методов, входящих в список вызова делегата, второй — для динамического обращения к ним. Преимущество подхода заключается в том, что можно полноценно контролировать обращение к каждому методу, на который ссылается делегат. Код примера вы обнаружите в листинге 6.10.

Листинг 6.10. Надежный подход работы с делегатами, ссылающимися на несколько методов

```
/*
  Листинг 6.10
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Опишем делегат, с которым будем работать.
  // Напомню, что это всего лишь тип.
  delegate void MyDelegate(string param);
  // Первый метод, на который мы будем ссылаться при помощи
  // делегата. Именно он будет вызывать исключение,
  // не позволяющее обратиться ко второму методу.
  static void Handler1(string param)
  {
    // Выведем на консоль значение переданного параметра,
    // а также уведомим пользователя о том, что данный
    // метод был вызван.
    Console.WriteLine("Handler1 invoked, param = {0}",param);
    // Преднамеренно выбросим исключение.
    throw new Exception();
  }
  // Второй метод, на который мы будем ссылаться из делегата.
  static void Handler2(string param)
  {
    // Сообщим пользователю о том, что метод
    // был вызван, а также выведем на консоль
    // значение переданного параметра.
    Console.WriteLine("Handler2 invoked, param = {0}",param);
  }
  // Точка входа в программу.
  static void Main()
  {
    // Создадим экземпляр описанного делегата.
```

```
    MyDelegate del = new MyDelegate(Handler1);
    // Присоединим к нему еще одну функцию.
    del += new MyDelegate(Handler2);
    // Последовательно пройдем по каждому делегату, входящему
    // в список вызова ранее созданного делегата.
    foreach (Delegate d in del.GetInvocationList())
    {
      // Обернем вызов функции в защищенный
      // блок, что позволит предотвратить
      // преждевременное завершение алгоритма.
      try
      {
        // Зададим параметры для вызываемой
        // Функции.
        Object[] param = { "Hello" };
        // А вот это и есть вызов одной из функций,
        // входящих в список вызова делегата.
        d.DynamicInvoke(param);
      // Это блок обработки исключений, произошедших
      // в защищенном блоке.
      catch(Exception ex)
      {
        // Сообщим пользователю о том, что при попытке
        // вызова одной из функций произошло исключение.
        Console.WriteLine("Oh mama some exception was occurred");
      }
    }
  }
};
```

В результате работы программы на консоль будут выведены следующие строки.

```
Handler1 invoked, param = Hello
Oh mama some exception was occurred
Handler2 invoked, param = Hello
```

Видно, что задачи, поставленные перед приложением, были успешно выполнены. Несмотря на то, что в одной из функций, на которую ссылался делегат, было выброшено исключение, это не отразилось на работе остальных функций.

Помимо кода, обрабатывающего исключения, можно добавить специальные сервисы, анализирующие значения, возвращаемые функциями, вызываемыми через делегат, что дополнительно позволит решить первую проблему. Достаточно странным представляется тот факт, что подобные возможности изначально не предоставлены в числе сервисов управления делегатами. Как-то мало верится, что специалисты Microsoft были не осведомлены о данной проблеме.

Отметим, что обращаться к методам можно не только через функцию `DynamicInvoke`. Для этих целей также можно использовать свойство `Method`. А точнее метод `Invoke`, предоставляемый классом `MethodInfo`, экземпляр которого можно получить через вышеупомянутое свойство класса `Delegate`. В итоге, код вызова можно заменить на следующий.

```
// Было.
d.DynamicInvoke(param)
// Стало.
d.Method.Invoke(d.Target,param)
```

Особую практическую пользу в этом подходе найти, конечно, сложно. Но зато вы будете всегда иметь запасной вариант при работе с делегатами.

MulticastDelegate.CreateDelegate

Метод позволяет динамически создать делегат заданного типа.

```
public static Delegate Delegate.CreateDelegate(
       Type type,
       MethodInfo method);
public static Delegate Delegate.CreateDelegate(
       Type type,
       Object target,
       string method);
public static Delegate Delegate.CreateDelegate(
       Type type,
       Type target,
       String method);
public static Delegate Delegate.CreateDelegate(
       Type type,
       Object target,
       String method,
       bool ignoreCase);
```

Этот метод введен в рамках поддержки технологии отражения, он позволяет динамически создавать делегаты произвольного типа. Он используется в том

случае, если на момент компиляции не известно, сколько аргументов принимает метод обратного вызова и какого они типа.

"Нулевые делегаты"

При программировании делегатов может создастся впечатление, что существуют, так называемые, нулевые делегаты. Под нулевыми делегатами подразумеваются экземпляры классов делегатов, не содержащие ссылок на методы. Такой делегат можно попытаться получить, изъяв из списка все методы, на которые он ссылается при помощи функции `Remove`. В коде это будет выглядеть так.

```
del = new MyDelegate(SomeMethod);
del -= new MyDelegate(SomeMethod);
```

И если до исполнения инструкции делегат ссылался на один метод `SomeMethod`, то в результате ее исполнения мы должны получить пустой делегат. На самом деле, мы получаем нулевую ссылку, то есть просто `null`. Что, кстати говоря, неочевидно, поскольку мы и далее имеем переменную `del`, которая подразумевает под собой ссылку на экземпляр делегата.

Большинство программистов попросту не ожидают такого подвоха со стороны системы и попадаются в коварную ловушку — они пытаются использовать методы данного экземпляра делегата. А поскольку он равен `null`, то происходит исключение `NullReferenceException`. Для того чтобы избежать подобных ошибок, в неоднозначных местах программы необходимо обязательно проверять значение делегата на `null`. Делается это следующим образом.

```
if (h != null)
  del("Hello, World!\n");
```

Во всех случаях, когда значение делегата не известно со стопроцентной уверенностью, такая проверка обязательно должна присутствовать. Это позволит вам избежать очень неприятных ошибок, которые обязательно возникнут.

6.5. События

В разделе будут рассмотрены члены типов — события.

Как устроены события и зачем они нужны

Начнем изучение с несколько идеализированного примера. Допустим, имеется некоторый компонент, и пускай для упрощения примера это будет банальная кнопка. Всем известно, что на кнопки можно нажимать. При нажатии кнопки происходит событие "щелчок", о котором необходимо обязательно уведомлять пользователя нашего компонента. Для этого введем в класс,

представляющий компонент кнопку, общедоступное поле `Click`, являющееся экземпляром делегата. И каждый раз, когда будет происходить соответствующее событие, будем обращаться к этому делегату. А он, в свою очередь, будет вызывать прикрепленные к нему функции и методы. Приведем код (листинг 6.11).

Листинг 6.11. Пример события на основе делегата

```
// Введем специальный делегат.
delegate void ClickHandler();
class Button
{
  // Это общедоступное поле-делегат, к которому каждый
  // может присоединить собственный метод.
  public ClickHandler Click;
  // Несколько идеализированная функция обработки
  // сообщений, приходящих на кнопку.
  void OnMsg(...)
  {
    // Предположим
    switch(msg)
    {
      // Вот мы как бы засекли нажатие на кнопку.
      case WM_LBUTTONDOWN:
        // Вызовем функции, связанные с нашим делегатом,
        // предварительно проверив, а зарегистрирована
        // ли хотя бы одна функция в поле-делегате.
        if (Click != null)
          Click();
    }
};
```

Теперь пользователи нашего класса смогут присоединить свои функции к переменной-полю экземпляру делегату и получить уведомление о произошедшем событии.

Но поскольку среда .NET является объектно-ориентированной, то мы обязаны соблюдать правила инкапсуляции полей. Соответственно, необходимо ввести дополнительные методы, обслуживающие поле-экземпляр делегата и контролирующие все, производимые над ним операции, а само поле полагается сделать закрытым. Сделаем это следующим образом — добавим две

функции `add_Click` и `remove_Click`, которые, соответственно, будут добавлять и убирать события в очередь делегата. Новый код представлен на листинге 6.12.

Листинг 6.12. Пример события на основе делегата, с поддержкой инкапсуляции полей

```
// Введем специальный делегат.
delegate void ClickHandler();
class Button
{
  // Поле-делегат, к которому будут присоединяться обработчики
  // нажатия кнопки.
  // Обратите внимание, это поле теперь является закрытым,
  // оно недоступно извне класса.
  private ClickHandler Click;
  // Введем два общедоступных метода,
  // которые будут предоставлять сервисы работы с событием.
  public add_Click(ClickHandler delegate)
  {
    // Воспользуемся удобным арифметическим оператором
    // для комбинирования делегатов.
    Click += delegate;
  }
  // Удаление функции из списка вызова.
  public remove_Click(ClickHandler delegate)
  {
    // Удалим функции, представленные делегатом.
    // delegate из нашего списка вызовов
    Click -= delegate;
  }
  // Несколько идеализированная функция обработки
  // сообщений, приходящих на кнопку.
  void OnMsg(...)
  {
    // Предположим.
    switch(msg)
    {
    // Вот мы как бы засекли нажатие кнопки.
```

```
    case WM_LBUTTONDOWN:
      // Вызовем функции, связанные с нашим делегатом,
      // предварительно проверив, а зарегистрирована
      // ли хотя бы одна функция в поле-делегате.
        if (Click != null)
          Click();
    }
};
```

Теперь наш компонент удовлетворяет основным парадигмам объектно-ориентированного программирования. Только вот, его код стал уж больно громоздким и его использование не будет столь наглядным, как прежде.

События .NET

Понимали это и разработчики, создававшие среду .NET. Для решения этой задачи они ввели специальное поле-событие, которое автоматически вводит поле-делегат и два дополнительных метода, обслуживающих его. Таким образом, автоматически реализуется представленная выше модель.

В языке высокого уровня C# определение таких полей осуществляется при помощи ключевого слова `event`. Также компилятор берет на себя заботу о работе с этими полями-событиями, благодаря чему к ним можно прибавлять и вычитать делегаты, хотя они, по сути дела, таковыми не являются. Приведем наглядный пример использования полей-событий (листинг 6.13).

Листинг 6.13. Использование полей-событий

```
/*
  Листинг 6.13
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем собственный делегат, не принимающий
// никаких значений.
delegate void MyDelegate();
// Это тестовый класс, он представляет собой гипотетический
// компонент кнопки.
class Button
```

```
{
  // Введем общедоступное событие, к которому
  // смогут подключаться все желающие.
  public event MyDelegate Click;
  // Данная функция необходима для того, чтобы
  // симулировать событие нажатия на кнопку.
  public void SimulateClick()
  {
    // Вызываем функции, связанные с событием Click,
    // предварительно проверив, зарегистрировался
    // ли кто-нибудь в данном событии.
    if (Click != null)
      Click();
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  static void Main()
  {
    // Создаем экземпляр класса/компонента.
    Button sc = new Button();
    // Добавляем обработчик к его событию.
    sc.Click += new MyDelegate(Handler);
    // Сами вызовем функцию, которая инициирует
    // возникновение события нажатия на кнопку.
    sc.SimulateClick();
  }
  // А это функция-обработчик события нажатия на кнопку.
  static void Handler()
  {
    Console.WriteLine("Hello, World!");
  }
};
```

В результате работы приложения, на консоль будет выведена следующая строка:

```
Hello, World!
```

Таким образом, работа с событиями представляется предельно простой. Она, по сути дела, ничем не отличается от взаимодействия с обыкновенными делегатами.

Внутренний механизм поддержки событий

Рассмотрим, какой код поддержки события был создан компилятором. Для этого изучим IL- код.

Во-первых, компилятор создал поле-делегат, в котором хранятся все зарегистрированные обработчики события.

```
.field private class MyDelegate Click
```

Обратите внимание, что поле является строго закрытым и может быть использовано только из самого класса, исключая даже его потомков (`private`). Также компилятор создал одноименное специализированное поле-событие, в котором указаны методы, реализующие внешнюю работу с данным событием. Причем необходимо отметить, что это поле по умолчанию является общедоступным.

```
.event MyDelegate Click
{
  .addon instance void Button::add_Click(class MyDelegate)
  .removeon instance void Button::remove_Click(class MyDelegate)
}
```

Дабы не утомлять читателя излишним чтением IL-листингов, приведем код методов на языке C#.

```
public void add_Click(MyDelegate del)
{
  Click += del;
}
public void remove_Click(MyDelegate del)
{
  Click -= del;
}
```

Методы будут использоваться только при работе с событием извне класса. Внутри же используется прямое обращение к полю `Click`, что несколько быстрее, чем вызов методов `add_Click` и `remove_Click`. Здесь можно усмотреть борьбу команды разработчиков компилятора C# за производительность создаваемых ими программ.

Для того чтобы подтвердить сказанное, приведем IL-код функции `SimulateClick`, которая непосредственно обращается к событию из самого

класса, если быть до конца точным, то не к самому событию, а к полю-экземпляру делегата.

```
.method public hidebysig instance void
        SimulateClick() cil managed
{
  .maxstack  1
  ldarg.0
  // Загружаем в стек указатель на интересующее нас поле.
  ldfld      class MyDelegate Button::Click
  brfalse.s  IL_0013
  ldarg.0
  // Загружаем в стек указатель на интересующее нас поле.
  ldfld      class MyDelegate Button::Click
  callvirt   instance void MyDelegate::Invoke()
  ret
}
```

Обратите внимание на выделенную жирным шрифтом инструкцию `ldfld` — она предназначена для работы с полями объекта. Также в листинге отсутствуют обращения к функциям `add_Click` и `remove_Click`. Что и требовалось доказать!

Контроль над событиями

При разборе первого примера подраздела говорилось, что введение дополнительных функций позволяет контролировать работу с событием. Но как оказалось, события .NET полностью закрыты для программиста. Они скрывают всю грязную работу за кулисами и контролировать обращение к событиям не представляется возможным. Но это утверждение верно только при стандартном способе использования событий. Также существует дополнительный расширенный режим их использования, при котором программист может самостоятельно объявить функции, управляющие поддержкой данного события. Для этого в языке C# предусмотрена специальная конструкция, с использованием двух дополнительных ключевых слов `add` и `remove`.

```
event Delegate SomeEvent
{
  add
  {
    // Код, реализующий добавление делегата к списку
    // вызова события.
  }
```

```
  remove
  {
    // Код, реализующий изъятие делегата из списка вызова
    // события.
  }
}
```

Единственное, что может смутить при рассмотрении кода, так это то, что `add` и `remove`, по сути дела, являются функциями, принимающими один параметр. Но из кода совершенно непонятно, как получить доступ к этому параметру. Тут есть одна маленькая хитрость — для этого необходимо воспользоваться ключевым словом **value**.

Недостатком использования такого подхода является то, что теперь компилятор не будет автоматически описывать поле-делегат и его придется вводить самостоятельно. Рассмотрим пример, демонстрирующий работу данного подхода (листинг 6.14). Этот пример по функциональности и внутренней архитектуре будет аналогичен предыдущему, только код поддержки работы с событием будет реализован в нем вручную.

Листинг 6.14. Поддержка событий в ручном режиме

```
/*
  Листинг 6.14
  File:   Some.cs
  Author: Copyright (C) 2002 Dubovcev Aleksey
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Опишем делегат, имеющий пустой прототип.
delegate void MyDelegate();
// Опишем подопытный класс/компонент.
class Button
{
  // Закрытое поле-ссылка на экземпляр делегата,
  // который будет обслуживать событие.
  private MyDelegate m_Click;
  // Пользовательское событие, с возможностью контроля доступа
  // к нему.
  public event MyDelegate Click
  {
    // Эта функция будет вызвана при попытке добавления
```

```
    // делегата в список вызова события.
    add
    {
      // Сообщим пользователю о том, что произведена
      // попытка добавить делегат.
      Console.WriteLine("add handler was invoked");
      // Добавим делегат в список вызова.
      m_Click += value;
    }
    // Эта функция будет вызвана при попытке изъятия
    // делегата из списка вызова события.
    remove
    {
      // Сообщим о том, что произведена попытка
      // изъять делегат из списка вызова.
      Console.WriteLine("remove handler was invoked");
      // Удалим эту функцию из списка обработки.
      m_Click -= value;
    }
  }
  // Эта функция необходима для того, чтобы
  // симулировать событие нажатия на кнопку.
  public void SimulateClick()
  {
    // Вызываем функции, связанные с событием Click,
    // предварительно проверив, зарегистрировался
    // ли кто-нибудь в этом событии.
    if (Click != null)
        Click();
  }
};

// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим тестовый экземпляр компонента/класса.
```

```
    Button sc = new Button();
    // Добавим обработчик события.
    sc.Click += new MyDelegate(Handler);
    // Косвенно вызовем функцию нашего обработчика.
    sc.SimulateClick();
    // Уберем функцию-обработчик из списка вызова
    sc.Click -= new MyDelegate(Handler);
    // Попытаемся снова осуществить вызов.
    sc.SimulateClick();
  }
  // Функция-обработчик для компонента/класса.
  public static void Handler()
  {
    Console.WriteLine("Hello World - Handler was invoked");
  }
};
```

В результате работы примера получим на консоли следующие строки.

```
add handler was invoked
Hello World - Handler was invoked
remove handler was invoked
```

Как видим, приложение действовало по заданной схеме, контролируя при этом добавление и изъятие делегатов.

Такая возможность может быть весьма полезна для анализа и слежения за функциями, прикрепленными к событию. К примеру, можно запретить добавление событий кодом, не имеющим определенных привилегий защиты.

6.6. Дополнительные возможности при работе с делегатами

Здесь будут рассмотрены дополнительные возможности, предоставляемые общей библиотекой типов для упрощения работы с делегатами. Некоторые из них могут быть весьма полезны в повседневном программировании.

Список делегатов — *EventHandlerList*

В рамках компонентной модели общей библиотеки классов введен дополнительный класс `EventHandlerList`. Он предназначен для упрощения разработки компонентов, содержащих большое количество событий. Класс по-

зволяет хранить в своем экземпляре неограниченное количество событий, организуя доступ к ним по помощи произвольных ключей типа `Object`. Наиболее интересные члены класса `EventHandlerList` описаны в табл. 6.2.

Таблица 6.2. *Члены класса* `System.ComponentModel.EventHandlerList`

Член класса	Описание
Методы	
`AddHandler`	Добавляет делегат в список по ключу
`RemoveHanlder`	Изымает делегат из списка по ключу

По сути дела, список является и не списком вовсе, а ассоциативным массивом. Но основное его достоинство заключается не в этом. Главное его отличие от обычных коллекций состоит в том, что он производит добавление и изъятие элементов из списка, учитывая особенности делегатов. Операции добавления и изъятия производятся при помощи методов `Combine` и `Remove`. Таким образом, по одному ключу может храниться несколько скомбинированных однотипных делегатов. Соответственно, при изъятии делегата по заданному ключу, элемент списка будет удален не полностью, а произойдет рекомбинация делегата, в ходе которой некоторые из его ссылок будут потеряны. Но сам делегат может не истощиться, а содержать еще несколько ссылок на методы. Такое поведение списка весьма удобно при разработке компонентов, предоставляющих пользователям множество событий. Можно попросту добавлять и удалять делегаты по заданным ключам, не задумываясь о дополнительной поддержке событий.

Продемонстрируем работу со списком на примере того же компонента кнопки (листинг 6.15). На этот раз он будет содержать два события, делегаты которых будут храниться в специализированном закрытом списке.

Листинг 6.15. Использование специализированной коллекции `EventHandlerList` при работе с делегатами

```
/*
  Листинг 6.15
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку компонентной
// модели.
```

```
using System.ComponentModel;
// Опишем делегат с пустым прототипом.
delegate void MyDelegate();
// Введем тестовый класс/компонент.
class Button
{
  // В этом закрытом списке будут храниться делегаты,
  // представляющие наши события.
  private EventHandlerList m_EHList;
  // Далее описаны два ключа, которые будут использоваться
  // для индексации элементов внутри списка.
  // Эти ключи определены с использованием идентификатора
  // readonly, поскольку необходима стопроцентная гарантия,
  // что ключи не изменят своего значения во время работы
  // программы, иначе работа программы будет непредсказуема.

  private static readonly object m_MouseUpKey = new Object();
  private static readonly object m_MouseDownKey = new Object();
  // Конструктор по умолчанию для класса.
  public Button()
  {
    // Инициализируем список.
    m_EHList = new EventHandlerList();
  }
  // Опишем первое событие с поддержкой контроля.
  public event MyDelegate MouseUp
  {
    add
    {
      // Добавим делегат в список по ключу.
      m_EHList.AddHandler(m_MouseUpKey,value);
    }
    remove
    {
      // Удалим делегат из списка по ключу.
      m_EHList.RemoveHandler(m_MouseUpKey,value);
    }
  }
  // Опишем второе событие с поддержкой контроля.
  public event MyDelegate MouseDown
  {
```

```
    add
    {
      // Добавим делегат в список по ключу.
      m_EHList.AddHandler(m_MouseDownKey,value);
    }
    remove
    {
      // Удалим делегат из списка по ключу.
      m_EHList.RemoveHandler(m_MouseDownKey,value);
    }
  }

  // Данная функция будет вызывать первое событие.
  public void SimulateMouseUp()
  {
    // Получаем нужный делегат из нашего списка
    // при помощи ключа.
    MyDelegate eh = (MyDelegate)m_EHList[m_MouseUpKey];
    // Проверяем, присоединены ли функции к делегату, полученному из
    // коллекции. В случае, если это так, вызываем делегат.
    if (eh != null)
      eh();
  }
  // А эта функция будет вызывать второе событие.
  public void SimulateMouseDown()
  {
    // Получаем нужный делегат из нашего списка
    // при помощи ключа.
    MyDelegate eh = (MyDelegate)m_EHList[m_MouseDownKey];
    // Проверяем, присоединены ли функции к полученному
    // делегату. В случае, если это так, вызываем делегат.
    if (eh != null)
      eh();
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Создаем экземпляр тестового класса.
```

```
    Button sc = new Button();
    // Присоединяем обработчики к событиям.
    sc.MouseUp += new MyDelegate(HandlerMouseUp);
    sc.MouseDown += new MyDelegate(HandlerMouseDown);
    // Вызываем события
    sc.SimulateMouseUp();
    sc.SimulateMouseDown();
  }
  // Обработчик для первого события.
  public static void HandlerMouseUp()
  {
    // Уведомляем пользователя о произошедшем
    // событии выводом соответствующего сообщения
    // на консоль.
    Console.WriteLine("HandlerUp was called");
  }
  // Обработчик для второго события.
  public static void HandlerMouseDown()
  {
    // Уведомляем пользователя о произошедшем
    // событии выводом соответствующего сообщения
    // на консоль.
    Console.WriteLine("HandlerDown was called");
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
HandlerUp was called
HandlerDown was called
```

Как видно, нововведение никоим образом не отразилось на использовании класса извне. Была всего лишь изменена внутренняя организация компонента, несколько упростив работу программистам при помощи специализированного хранилища делегатов.

Стандартный делегат общей библиотеки

На протяжении главы всегда использовался собственноручно определенный делегат с пустым прототипом. Вот как он выглядел.

```
delegate void MyDelegate();
```

Многие классы стандартной библиотеки используют делегаты для уведомления о произошедших в них событиях. Соответственно, разработчики среды

.NET сочли, что будет целесообразно ввести общий стандартный тип делегата, который будет использоваться во всей библиотеке. Его прототип представлен ниже.

```
public delegate void EventHandler(
   // Ссылка на объект, вызвавший событие.
   object sender,
   // Параметры, описывающие событие.
   EventArgs e
);
```

Этот делегат принимает всего лишь два параметра, что явно маловато для универсального делегата. Но все же оказывается, параметров с лихвой хватает для передачи любой информации. Сам по себе класс `EventArgs` не содержит ни одного интересного члена, способного передавать какую бы то ни было информацию. Он введен лишь для обобщения. Когда необходимо передать дополнительную информацию, вводится новый класс, производный от `EventArgs`, в котором уже и вводятся поля, передающие необходимую информацию. В одной общей библиотеке у класса `EventArgs` 100 потомков, и их количество от версии к версии среды исполнения неуклонно растет.

При обращении к событию, имеющему тип данного делегата, в качестве параметра необходимо передать ссылку на текущий объект (`this`), а во втором параметре экземпляр класса `EventArgs` или производного от него.

```
SomeEvent(this, EventArgs.Empty);
```

где свойство `Empty` попросту возвращает пустой экземпляр типа `EventArgs`. В принципе, его можно создать и самому воспользовавшись оператором `new`.

```
SomeEvent(this, new EventArgs());
```

Правда, такой подход будет работать несколько медленнее, чем первый, поскольку здесь появилась дополнительная операция создания объекта. И причем необходимо отметить, что она далеко из не самых быстродействующих.

6.7. В заключение главы

Использование делегатов и событий в обычных управляемых приложениях несложно. В главе были рассмотрены все наиболее важные аспекты проблемы. Но при проектировании производительных и устойчивых приложений, можно задуматься о создании собственного механизма обращения к делегатам, с использованием многопоточной модели и более защищенных алгоритмов.

Глава 7

Обработка исключений

С незапамятных времен непрерывно растущая армия разработчиков программного обеспечения бьется над улучшением качества программ. Наиболее важной является задача повышения их устойчивости. Кому же нужна программа с красивым и удобным интерфейсом, но которая постоянно "падает" из-за ошибок или неадекватно себя ведет. Любой пользователь променяет ее на менее удобную, но более стабильно работающую программу.

Разработано огромное количество методик и технологий, позволяющих достигнуть желаемого. Обработка исключений — одна из таких технологий. Она позволяет сделать ваши программы более надежными, не концентрируя при этом все ваше внимание на отлове потенциальных ошибок. Фактически, технология обработки исключений предназначена для повышения устойчивости программы, абстрагируясь в основном коде от обработки ошибок и предсказания потенциальных проблем. В главе изложена суть технологии, приведены примеры, которые позволят вам создавать устойчивые приложения на ее основе.

7.1. Общие принципы работы

Технология обработки исключений построена на принципе защиты участков кода, без введения дополнительного кода обработки ошибок в сами эти участки. К защищенным участкам кода прикрепляются два вида обработчиков: первые вызываются при возникновении исключения, вторые выполняются в любом случае. В общем виде схема обработки исключений выглядит так.

```
void SomeFunction()
{
  //Защищенный участок кода.
  try
  {
    // Здесь располагается код, при исполнении которого
```

```
    // потенциально может возникнуть исключение.
    SomeFunction2();
  }
  //Обработчик исключительной ситуации.
  catch(Exception ex)
  {
    Console.WriteLine("There was exception here.");
  }
  finally
  {
    Console.WriteLine("Finally code");
    //Код, который выполняется в любом случае
  }
}
// Функция, в ходе работы которой происходит вывод
// исключения.
void SomeFunction2()
{
  // Производим вывод исключения.
  throw new Exception();
}
```

Обработка исключений предполагает использование в коде трех типов блоков: `try`, `catch` и `finally`. Рассмотрим их подробнее.

Try

Блок `try` обрамляет участок защищенного кода, при исполнении которого может возникнуть исключение. Но обратите внимание, что сам код не содержит операций, предназначенных для обработки ошибок. Ранее, когда не существовало исключений, приходилось в каждой строке кода проводить проверку корректности ее исполнения. Что было гораздо более трудоемко, чем использование исключений.

Достаточно интересным фактом является то, что блоки `try` могут быть вложенными. К примеру, так.

```
try /*Первый (верхний) защищенный блок*/
{
  ....
  try /*Второй (внутренний) вложенный защищенный блок*/
  {
```

```
    ...
  }
  catch() /*Обработчик второго защищенного блока*/
  {
    try /*Третий вложенный защищенный блок*/
    {
      ...
    }
    catch() /*Обработчик третьего защищенного блока*/
    {
      ...
    }
  }
  ....
}
catch() /*Обработчик первого защищенного блока*/
{
}
```

Хотя в реальной жизни такая схема обработки исключений используется не часто. Но все же наличие такой возможности не может не радовать.

Catch

Блок `catch` представляет собой обработчик заранее предусмотренной исключительной ситуации. Таких блоков может быть несколько. Каждый для своего типа исключения. Более подробно этот тип блоков будет рассмотрен далее.

Finally

В блоке `finally` представлен код, который будет выполнен всегда, независимо от того, произошло исключение или нет.

Примечание

Иногда `finally`-блоки называют блоками завершения. Но здесь данный термин использоваться не будет.

Такие блоки предназначены для выполнения кода, очищающего ресурсы приложения. Поскольку блок выполняется в любом случае, следовательно, ресурсы будут освобождены всегда. Ранее для организации подобной логики приходилось прибегать к достаточно сложным алгоритмам, которым приходилось следить за исполнением каждой строки кода. Теперь же все делается

по мановению волшебной палочки. Для большей наглядности приведем абстрактный пример использования `finally`-блока (листинг 7.1).

Листинг 7.1. Абстрактный пример использования `finally`-блока

```
/*
       Листинг 7.1
       File:   Some.cs
       Author: Дубовцев Алексей
*/
...
// Функция имитирует чтение некоторых данных
// из файла, имя которого передано ей в качестве первого
// параметра.
public static void ReadSomeData(string strFileName)
{
  // Опишем локальную переменную, видимую для всего кода функции.
  FileStream fs = null;
   // Защитим участок кода.
  try
  {
    // Откроем файл.
    fs = new FileStream("qq", FileMode.Open);
    // Создаем объект для чтения файла.
    StreamReader sr = new StreamReader(fs);
    // Здесь гипотетически может произойти исключение
    // типа EnfOfStreamException или другое
  }
  // Блок обработки исключения, отлавливающий все исключения,
  // включая EnfOfStreamException.
  catch (Exception ex)
  {
    // Сообщим пользователю о произошедшем исключении.
    Console.WriteLine("Some errors here {0}",ex.ToString());
  }
  // А здесь мы освободим использовавшиеся ресурсы,
  // то есть закроем файл.
  finally
  {
```

```
    //Закроем файл, предварительно проверив ссылку fs
    if (fs != null)
      fs.Close();
  }
...
```

Функция содержит единственный защищенный блок, в котором происходит работа с файлами. К этому защищенному участку прикреплен блок `finally`, в котором происходит закрытие файла. Даже если при работе с ним произойдет какая-либо непредвиденная ситуация, файл все равно будет закрыт. Таким образом, все ресурсы будут освобождены.

Примечание

Обратите внимание на переменную `fs`, описанную не в блоке `try`, а в самом начале тела функции. Сделано это из-за того, что блок `try` обладает одной неприятной особенностью — он ограничивает область видимости описанных в нем переменных. Вследствие этого, при определении переменной `fs` внутри защищенного блока, она будет не доступна из блока `finally`.

Прошу вас обратить особое внимание на код самого блока `finally`. Он содержит дополнительную проверку переменной `fs`. Проверка необходима для того, чтобы избежать обращения к нулевой ссылке. Такое может произойти, если файл вообще не будет открыт. К примеру, если его не существует или у программы нет соответствующих на то прав.

Дополнительные проверки в блоке `finally` крайне необходимы. Желательно всеми доступными средствами постараться предотвратить возникновение исключений в завершающих участках кода. Как правило, выброс исключения в `finally`-блоке непоправимо нарушает ход исполнения программы. Большинство программистов упускают из виду, что исключение может возникнуть на этом участке кода.

К примеру, в MSDN по представленным ссылкам

ms-help://MS.NETFrameworkSDK/cpguidenf/html/cpconthrowingexceptions.htm

`// Для новой версии SDK ссылка будет такой.`

ms-help://MS.NETFrameworkSDKv1.1/cpguidenf/html/cpconthrowingexceptions.htm

можно найти следующий пример (листинг 7.2). Код приведен без исправлений.

Листинг 7.2. Пример из MSDN, в котором допущена неявная ошибка

```
using System;
using System.IO;
public class ProcessFile
```

```
{
  public static void Main()
  {
    FileStream fs = null;
    try
    //Opens a text tile.
    {
      fs = new FileStream("data.txt", FileMode.Open);
      StreamReader sr = new StreamReader(fs);
      string line;
      //A value is read from the file and output to the console.
      line = sr.ReadLine();
      Console.WriteLine(line);
    }
    catch(FileNotFoundException e)
    {
      Console.WriteLine("[Data File Missing] {0}", e);
      throw new FileNotFoundException("[data.txt not in c:\\dev↵
      directory]",e);
    }
    finally
    {
      fs.Close();
    }
  }
}
```

Если скомпилировать и запустить приложение, то помимо `FileNotFoundException`, сработает незапланированное исключение `NullReferenceException`, которое произойдет при попытке закрытия файла в блоке `finally`.

Возникновения исключения

Понятно, что исключения сами по себе не возникают, они могут быть выброшены либо самой программой, либо средой исполнения.

Примечание

В отдельных случаях исключения выбрасываются самой операционной системой. При этом они преобразуются средой исполнения во внутренний формат, незаметно для пользователя. Таким образом, системные исключения можно приравнять к исключениям, выводимым средой исполнения.

Программа может выбросить исключение при помощи ключевого слова `throw`, аргументом которому передан объект, описывающий исключение. К примеру, таким образом:

```
// Выбрасываем исключение типа Exception().
throw new Exception();
// Или так.
// Описываем переменную для объекта, который будет обрабатывать
// исключение.
Exception ex;
// Создаем объект, который будет описывать исключение.
ex = new Exception();
// Выбрасываем исключение.
throw ex;
```

Необходимо четко понимать различие между самим механизмом обработки исключений и объектами, описывающими исключения. При возникновении события (исключения), среда исполнения запускает механизм обработки исключений. В его компетенцию входит: поиск подходящего `catch` блока, выполнение всех необходимых `finally`-блоков, а также, в особых случаях, вызов обработчика необработанных исключений (прошу извинить за невольный каламбур) или даже полного прекращения работы программы. Объект, который мы передаем в качестве первого параметра, описывает произошедшее событие (исключение). Он является чем-то вроде паспорта произошедшего исключения. Таким образом, исключение является событием, а объект паспортом, описывающим его.

При этом событие (исключение) не обязательно должно носить характер ошибки. Исключения можно использовать и в обычной логике программы. При возникновении исключения среда исполнения автоматически подыскивает подходящий `catch`-блок, который может обработать исключение. После того как блок найден, среда выполняет все необходимые `finally`-блоки, а затем передает управление `catch`-блоку.

7.2. Обработчики исключений (*catch*-блоки)

`Catch`-блоки служат для обработки потенциальных исключительных ситуаций. Каждый `catch`-блок состоит из двух частей: фильтра типа исключений, а также кода, отвечающего за обработку исключений этого типа. Рассмотрим пример.

```
...
try
{
  // Здесь располагается код, который потенциально может вызвать
```

```
  // исключение.
  ...
}
// Данный блок захватит все исключения типа FileNotFoundExcpetion, а
// также производные от него.
catch (FileNotFoundException ex)
{
  // Здесь необходимо поместить обработку данного типа исключения.
}
// Блок захватит все исключения типа Exception, а также
// производные от него.
catch (Exception ex)
{
  // Исключения этого типа лучше выбросить далее.
  throw ex;
}
// Этот фильтр захватывает абсолютно все исключения.
catch
{
  // Выбросим исключение дальше.
  throw;
}
```

Каждый фильтр обработчика исключения принимает в качестве параметра тип объекта исключения, который может обработать блок. Но, помимо данного типа, обработчик будет захватывать все производные (унаследованные) от него типы (рис. 7.1).

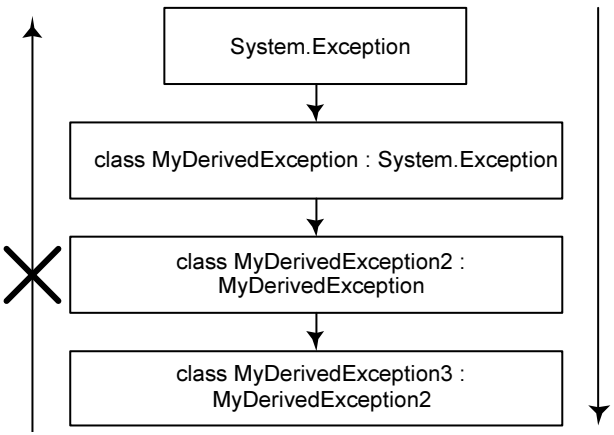


Рис. 7.1. Захват исключений по "наследству"

Стрелками на рисунке указана иерархия наследования классов. Таким образом, исключение типа `MyDerivedException2` может быть захвачено всеми фильтрами, типы которых стоят выше (`Exception`,`MyDerivedException`). Но типы, располагающиеся ниже в иерархии наследования, захватить его не могут (`MyDerivedException3`).

В соответствии с общеязыковой спецификацией (CLS, Common Language Specification), все типы исключений в среде .NET обязаны быть производными от `System.Exception`. Большинство высокоуровневых языков поддерживают работу только с такими исключениями.

В нашем примере, второй обработчик имел тип фильтра `Exception`. Такой фильтр захватит подавляющее большинство произошедших исключений, даже те, о которых разработчик может не подозревать. Следовательно, обработать корректно он их не сможет. Использование таких фильтров не рекомендуется, поскольку они могут нарушить логику хода исполнения программы. Если необходимость в таком фильтре все же возникла, то рекомендуется перевыбросить исключение, пришедшее в этот блок, для того чтобы оно было захвачено соответствующим блоком `catch`, располагающимся выше в стеке вызовов.

При работе с исключениями на CLS совместимом компиляторе, вы будете ограничены иерархией классов `Exception`. Исключения, в качестве базового которых выступал бы другой класс, использовать не удастся. Хотя фактически среда исполнения не накладывает столь строгих ограничений. Ей вообще безразлично, какой тип будет иметь объект, описывающий исключение. В чем можно легко удостовериться, используя любой компилятор, не придерживающийся правил CLS. К примеру, Managed Extensions for C++ или IL. Эти языки поддерживают работу с исключениями любого типа, даже отличного от `System.Exception`. В связи с этим, в C# был введен дополнительный тип фильтров, который позволяет захватывать исключения, не производные от `System.Exception`. В нашем примере подобный фильтр стоит последним в списке. Фильтр не содержит явного указания типа исключения, а соответственно, не позволяет получить доступ к объекту, описывающему исключение. Такое ограничение введено в соответствии с требованиями CLS. Таким образом, при помощи подобного фильтра возможно лишь узнать о том, что исключение произошло, но получить информацию о нем, к сожалению, не представляется возможным.

Рассмотрим, как работает фильтр данного типа. Компилятор автоматически использует тип `Object` для фильтра данного обработчика.

```
catch
{
  // Перевыбрасываем исключение, даже не получая доступа к нему.
  throw;
}
```

в результате работы компилятора получится следующий код.

```
catch (Object obj)
{
  throw obj;
}
```

Поскольку класс `Object` является базовым абсолютно для всех классов среды исполнения, обработчик с подобным фильтром еще опаснее, чем с типом `Exception`. Такой фильтр будет захватывать все подряд. Перевыбрасывание исключений для таких фильтров строго обязательно. Этот фильтр может перехватить исключение, которое сообщало о жизненно важном событии. А коль скоро мы его перехватили, то оно может и не дойти до адресата.

Помимо стандартных фильтров, принимающих в качестве параметра тип объекта исключения, некоторые языки поддерживают установку дополнительных условий. Приведем пример (листинг 7.3).

Листинг 7.3. Установка дополнительных условий при обработке исключения

```
'
' Листинг 7.3
' File:   Some.vb
' Author: Дубовцев Алексей
'
' Подключим основное пространство имен общей библиотеки классов.
Imports System
// Основной класс приложения.
Public Class App
  // Точка входа в приложение.
  Public Shared Sub Main()
   'Переменная, которая будет использоваться в фильтре
   'обработчика исключений,
   Dim a as String
   // Установим первоначальное значение переменной.
   a = "Hello, world!"
  'Защищенный блок
   Try
    Console.WriteLine("Try block.")
   'Изменим значение переменной
  a = "z'u z'u z'u"
  'Выбросим исключение нужного типа.
```

```
    Throw new Exception()
   'Блок обработки исключения типа Exception.
   'Но! Внимание! На данный блок установлено дополнительное
   'условие.
    Catch ex As Exception When a = "hello world"
     Console.WriteLine("catch block")
     '...
    End Try
  End Sub
End Class
```

Код, расположенный в защищенном блоке, выбрасывает исключение типа `Exception`. По идее, оно должно быть перехвачено обработчиком, расположенным ниже. Но он содержит условие, указанное при помощи ключевого слова `When`, которое не позволить выполнить код обработки. В результате работы приложения будет выброшено необработанное исключение `System.Exception`.

Сама среда исполнения не поддерживает подобной функциональности. Это искусственный метод, предоставляемый самим компилятором. Надо отметить, что при генерации кода, отвечающего за обработку условия, компилятор VB .NET оказался, мягко говоря, не на высоте. Код, создаваемый им, не оптимален. Сам код рассматривать здесь не имеет смысла, поскольку в последующих версиях все может измениться в лучшую сторону. Но вообще, серьезных программистов я бы предостерег в использовании данного языка.

Очень важным моментом является порядок выборки обработчиков, если к одному защищенному блоку (`try`) их прикреплено несколько. Для защищенного блока сканирование обработчиков происходит в порядке их следования в программе. Причем самое неприятное заключается в том, что если исключение будет перехвачено одним из блоков, стоящих выше, то в другие оно не попадет ни при каких условиях, даже если вам этого очень сильно захочется. При размещении обработчиков после `try`-блока, необходимо учитывать иерархию классов их фильтров. Рассмотрим гипотетический пример (листинг 7.4).

Листинг 7.4. Пример неправильного расположения обработчиков

```
/*
  Листинг 7.4
  File:   Some.cs
  Author: Дубовцев Алексей
*/
```

```
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Напомню, класс DivideByZeroException является производным
// от ArithmeticException.
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Вводим защищенный блок.
    try
    {
      // Выбрасываем исключение.
      throw new DivideByZeroException();
    }
    // Данный обработчик первым захватит выброшенное нами исключение,
    // поскольку класс ArithmeticException является предком
    // DivideByZeroException.
    catch (ArithmeticException ex)
    {
      Console.WriteLine("Exception catch");
    }
    // Здесь мы ожидаем поймать выброшенное ранее исключение
    // DivideByZeroException, но оно сюда попросту не дойдет.
    catch (DivideByZeroException ex)
    {
      Console.WriteLine("FileNotFoundException catch");
    }
  }
}
```

Пример гипотетический, поскольку компилятор откажется его собирать. Компилятор C# пытается изобразить интеллектуальность и мотивирует отказ от сборки данного исходного кода тем, что обработчик `DivideByZeroException` не будет вызван. Но, с точки зрения IL-кода, никаких ограничений на подобное построение кода не существует. И вполне не исключено, что менее интеллектуальные компиляторы пропустят подобную ошибку.

7.3. Обзор класса *System.Exception*

Класс является базовым в CLS модели исключений .NET. Предполагается, что любой объект, описывающий исключение, будет являться предком этого класса. Сам класс вполне прост, он содержит следующие члены (табл. 7.1).

Таблица 7.1. *Класс* `System.Exception`

Член класса	Описание
`Exception`	Конструктор класса, позволяющий создать объект-исключение
`HelpLink`	Ссылка на документацию, в которой описано исключение
`InnerException`	Содержит первоначальное исключение, на базе которого было создано текущее
`Message`	Сообщение, кратко описывающее исключение
`Source`	Имя приложения или объекта, вызвавшего исключение
`StackTrace`	Возвращает строку с именами всех методов, вызванных до возникновения исключения; строка формируется при помощи трассировки стека функций
`TargetSite`	Позволяет получить доступ к методу, который вызвал исключение
`HResult (protected)`	Содержит HRESULT код ошибки
`GetBaseException()`	Возвращает исходное исключение
`ToString()`	Возвращает строковое представление исключения

Перейдем к рассмотрению членов данного класса.

Конструкторы класса *Exception*

Всего у класса `Exception` существует четыре конструктора: три открытых и два закрытых. Три открытых обязательны для всех классов исключений. Они перечислены ниже.

```
// Конструктор класса, не принимающий параметров.
public Exception();
public Exception(
  // Строка, кратко описывающая причину произошедшего исключения.
```

```
  string message
);
// Конструктор, используемый при изменении типа объекта,
// описывающего исключение.
public Exception(
  // Строка, кратко описывающая исключение.
  string message,
  // Первоначальное исходное исключение.
  Exception innerException
);
```

Среди этих конструкторов особого внимания заслуживает последний. Он предназначен для изменения типа объекта исключения. Такое бывает необходимо для того, чтобы скрыть внутреннюю логику работы алгоритма. Рассмотрим на примере функции, которая осуществляет чтение каких-либо данных из файла. Предполагается, что файл имеет определенный, заранее известный формат. Если файл неожиданно кончается, то классы стандартной библиотеки выбрасывают исключение типа `EndOfStreamException`. Соответственно, можно предположить, что формат файла не верен. В таком случает, будет более правильно сообщить пользователю нашей функции не о достижении конца файла, а о его неверном формате. Для этого создадим исключение нового типа `InvalidFileFormatExcpetion` и перевыбросим его. Пример приведен в листинге 7.5.

Листинг 7.5. Изменение типа исключения

```
public static void ReadSomeData(string strFileName)
{
  // Вводим защищенный блок.
  try
  {
    // Откроем файл.
    FileStream fs = new FileStream(strFileName, FileMode.Open);
    //Создаем объект, необходимый для чтения файла.
    StreamReader sr = new StreamReader(fs);
    ...
    // Здесь гипотетически может произойти исключение
    // типа EnfOfStreamException.
    ...
  }
  // Введем обработчик, предусматривающий окончание файла.
```

```
  catch(EndOfStreamException ex)
  {
    // Создадим новый объект, описывающий исключение.
    // Это, по сути дела, и есть изменение типа исключения.
    InvalidFileFormatException myEx =
new InvalidFileFormatException( "Файл имеет неправильный формат", ex);
    // Перевыбросим исключение.
    throw ex;
}
```

В случае перевыброса исключения, необходимо оставить информацию об исходном объекте исключения. Она может пригодиться при отладке приложения. Для этого предназначен второй параметр конструктора. Впоследствии доступ к объекту, переданному через этот параметр, можно получить при помощи свойства `InnerException`.

В соответствии со спецификацией CLS все классы, производные от `Exception`, обязательно должны поддерживать все три конструктора. Это необходимо учитывать при создании собственных классов исключений.

Помимо трех открытых конструкторов, существует специальный закрытый конструктор. Его прототип представлен ниже.

```
protected Exception(
   // Объект, удерживающий данные объекта-исключения.
   SerializationInfo info,
   // Контекст сериализации.
   StreamingContext context
);
```

Конструктор используется для сериализации объектов исключений. Как ни удивительно, но такое происходит не так уж и редко. В модели удаленного исполнения кода .NET Remoting, исключения могут передаваться через контекст исполнения. Фактически, исключения, произошедшие на сервере, воспринимаются клиентом как собственные, и наоборот. Это возможно благодаря передаче объектов-исключений при помощи механизма сериализации.

Exception.HelpLink

Прототип

```
public virtual string HelpLink {get; set;}.
```

Свойство позволяет установить или получить доступ к ссылке на документацию, описывающей исключение данного типа. Ссылка должна быть задана в формате URL. К примеру, она может выглядеть так.

```
"file:///C:/Program Files/Cool App/Help/help.html#ZloException"
```

Любое уважающее себя приложение обязано предоставлять полную документацию, описывающую не только стандартные возможности, но также нестандартные ситуации и ошибки. Многим разработчикам, досконально знающим свою программу, может показаться такая информативность избыточной. Но, поверьте, если вам как разработчику понятно сообщение вроде **"неисправность пула подкачки ресурсов по пятому каналу"**, то обычного пользователя оно, скорее всего, приведет в ужас. Теперь, когда есть возможность ассоциировать с каждой ошибкой раздел документации, не стоит пренебрегать этим.

Exception.InnerException

Прототип

```
public Exception InnerException {get;}.
```

Возвращает объект, который был передан второму параметру конструктора `Exception`. То есть при помощи этого свойства можно получить объект, из которого преобразовывалось текущее исключение. Если для создания исключения использовался обычный конструктор, то значение поля будет равным `null`.

Exception.Message

Прототип

```
public virtual string Message {get;}.
```

Возвращает краткое описание произошедшего исключения. Информация очень полезная, поскольку она выводится отладчиком и позволяет оперативно разобраться в причине произошедшего исключения (рис. 7.2).

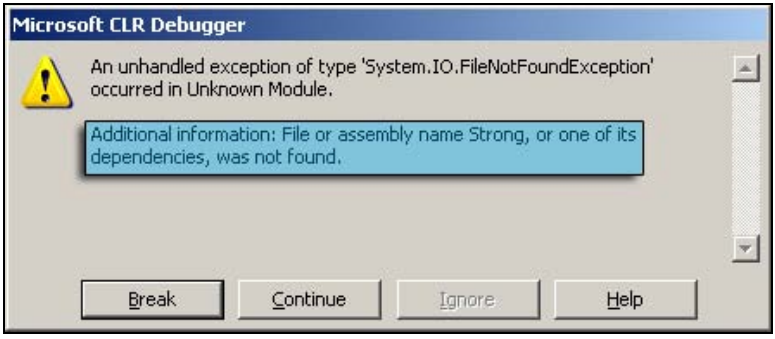


Рис. 7.2. Краткая информация произошедшего исключения, выводимая отладчиком

Exception.Source

Прототип

```
public virtual string Exception.Source {get; set;}.
```

Свойство позволяет узнать имя приложения или сборки, в котором произошло исключение. При разработке больших или распределенных сетевых приложений оно незаменимо.

Exception.StackTrace

Прототип

```
public virtual string Exception.StackTrace {get;}.
```

Свойство позволяет получить список методов, вызов которых привел к возникновению исключения. Более подробно о процессе трассировки стека будет рассказано далее.

Exception.TargetSite

Прототип

```
public MethodBase Exception.TargetSite {get;}.
```

Свойство позволяет определить метод, в котором произошло исключение. При помощи объекта `MethodBase`, возвращаемым им, можно получить полный доступ к методу. К его метаданным, отладочной информации и даже самому коду метода (IL-код) при помощи технологии отражения.

Exception.ToString

Прототип

```
public override string Exception.ToString();
```

Возвращает строку, содержащую общую информацию об исключении, включая тип исключения, строку описания и метод, в котором оно сработало. Приведем простой пример (листинг 7.6).

Листинг 7.6. Краткая информация об исключении

```
/*
  Листинг 7.6
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
```

```
using System;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Введем защищенный блок.
    try
    {
      // Создадим новый объект, представляющий исключение.
      Exception ex = new Exception("Раз, два, три");
      // Выбросим только что созданное исключение.
      throw ex;
    }
    // Обработчик нашего защищенного блока.
    catch(Exception ex)
    {
      // Выведем на консоль строковое представление
      // отловленного исключения.
      Console.WriteLine(ex.ToString());
    }
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
System.Exception: Раз, два, три
   at App.Main()
```

Метод позволяет легко и быстро получить полное представление о произошедшем исключении. Его использование крайне удобно в отладочных версиях приложений.

Exception.GetBaseException

Прототип

```
public virtual Exception Exception.GetBaseException();
```

При возникновении исключения, тип объекта, описывающего его, может быть неоднократно изменен и перенаправлен. Этот метод, в отличие от

InnerExcpetion, позволяет узнать тип не предыдущего исключения, а первоначального. Того самого, которое было выброшено первым.

7.4. Иерархия типов исключений в общей библиотеке классов

В документации от Microsoft сказано, что все исключения подразделяются на три вида: выбрасываемые самой средой исполнения (ее виртуальной машиной), общей библиотекой классов .NET, а также пользовательские исключения. Для первых двух предполагается наследование от класса SystemException, а для пользовательских от ApplicationException (табл. 7.2).

Таблица 7.2. *Иерархия различных видов исключений*

Для системы	Для приложений
System.Object	System.Object
System.Exception	System.Exception
SystemException	ApplicationException

У класса SystemException имеется более 65 потомков, поскольку он пользуется огромнейшей библиотекой классов .NET, а также самой средой исполнения. Было бы верно предположить, что у классов ApplicationException и Exception вообще не должно быть потомков. Но нет, Microsoft отступила от введенного ею правила, и несколько исключений все же являются их потомками. Их можно не перечислять, поскольку их не наберется даже с десяток. Весьма вероятно, что в будущих версиях библиотеки данные классы станут потомками SystemException. Это не должно внести коренных проблем обратной совместимости. Хотя в отдельных случаях все-таки может изменить логику некоторых программ.

Системные исключения

Отдельно хотелось бы рассказать про системные исключения, выбрасываемые самой средой исполнения, то есть ее виртуальной машиной. Основные и наиболее важные из этих исключений перечислены в табл. 7.3.

Таблица 7.3. *Список системных исключений, выбрасываемых самой средой исполнения*

Название класса	Базовый класс	Описание
Exception	Object	Базовый класс для всей системы исключений .NET

Таблица 7.3 *(окончание)*

Название класса	Базовый класс	Описание
`SystemException`	`Exception`	Базовый класс для исключений, выбрасываемых средой исполнения
`IndexOutOfRangeException`	`SystemException`	Выбрасывается в случае, если среда исполнения обнаруживает выход индекса за границы массива
`NullReferenceException`	`SystemException`	Выбрасывается при попытке использования нулевой ссылки
`InvalidOperationException`	`SystemException`	Выбрасывается при отсутствии возможности выполнения некоторой операции (к примеру, удаления элемента из пустой коллекции)
`ArgumentException`	`SystemException`	Базовый класс, для всех исключений, связанных с ошибками передачи аргументов в функции
`ArgumentNullException`	`SystemException` `ArgumentException`	Выбрасывается при передаче нулевой ссылки в качестве аргумента функции, когда это недопустимо
`ArgumentOutOfRangeException`	`SystemException` `ArgumentException`	Выбрасывается методами, которые проверяют выход аргумента за допустимые границы
`ExternalException`	`SystemException`	Класс является базовым для исключений, которые происходят за границами управляемого кода, подчиняющегося среде исполнения
`ComException`	`SystemException` `ExternalException`	Исключение, создаваемое при неудачной попытке вызова COM-метода
`SEHException`	`SystemException` `ExternalException`	Объект, представляющий произошедшее внешнее SEH (Structured Exception Handling, структурная обработка исключений) исключение

С этими типами исключений вам придется сталкиваться наиболее часто. Все они, кроме SEHException, просты в использовании и в отдельном обсуждении не нуждаются. А вот SEHException будет рассмотрен отдельно.

Внешние исключения (*SEHException*)

При возникновении SEH-исключения, во внешнем по отношению к среде исполнения коде, она перебрасывает его в управляемую среду в виде исключения типа `SEHException`. К примеру, если при вызове функции из внешней библиотеки DLL при помощи механизма `PInvoke` возникнет SEH исключение, то оно будет выброшено внутри управляемой среды в виде обычного .NET-исключения. Покажем это на примере, который будет состоять из двух файлов: динамической библиотеки и управляемого .NET-приложения, использующего функцию, экспортируемую этим приложением. Нетрудно догадаться, что функция из DLL будет выбрасывать SEH-исключение. Для начала рассмотрим код библиотеки (листинг 7.7).

Листинг 7.7. Динамическая библиотека, выдающая SEH-исключение

```
/*
  Листинг 7.7
  File:   Dll.cpp
  Author: Copyright (C) 2003 Dubovcev Aleksey
*/
// Подключим стандартный для Windows API заголовочный файл.
#include <windows.h>
// Функция, экспортируемая из DLL. При обращении к ней будет
// происходить исключение/
void DoSeh ()
{
  // Выполним попытку обращения по нулевому адресу. Там расположена
  // служебная отладочная страница со специальными атрибутами,
  // инициирующими выброс исключения STATUS_ACCESS_VIOLATION при любом
  // обращении к ней.
  *(int*)0 = 5; // Число 5 никакой роли не играет и выбрано случайно
}
```

Также для создания библиотеки необходим специальный `def`-файл с указанием имен экспортируемых функций (листинг 7.8).

Листинг 7.8. Файл, указывающий функции, экспортируемые из динамической библиотеки

```
;
;       Листинг 7.8
;       File:   Dll.def
;       Author: Дубовцев Алексей
;
; Имя динамической библиотеки.
LIBRARY Seh.dll
; Раздел со списком экспортируемых функций.
EXPORTS
  DoSeh
```

После создания этих файлов можно собрать динамическую библиотеку при помощи двух следующих команд.

```
cl dll.cpp -c
link Dll.obj /dll /def:dll.def /noentry /out:seh.dll /kernel32.lib
```

Итоговый размер динамической библиотеки будет приблизительно два с половиной Кбайта. Можно в принципе и меньше, но мы этого делать не будем. Теперь приступим к самому интересному: написанию программы под .NET, которая будет использовать библиотеку, а также отлавливать произошедшие в ней исключения. Пример программы приведен в листинге 7.9.

Листинг 7.9. Управляемое приложение, отлавливающее SEH-исключения

```
/*
  Листинг 7.9
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Здесь находятся классы, обеспечивающие взаимодействие
// с неуправляемым кодом.
using System.Runtime.InteropServices;
// Основной класс приложения.
```

```
class App
{
  // Импортируем функцию из ранее созданной библиотеки,
  // поместим ее для удобства внутрь нашего класса.
  [DllImport("seh.dll")]
  public static extern void DoSeh();
  // Точка входа в приложение.
  public static void Main()
  {
    // Объявляем защищенный блок.
    try
    {
      // Вызываем функцию из библиотеки.
      // ВНИМАНИЕ!!! Здесь произойдет исключение.
      DoSeh();
    }
    // Объявим обработчик защищенного блока, в качестве типа для которого
    // укажем класс SEHException, указывающий на то, что мы хотим
    // отловить Windows-исключение.
    catch (SEHException a)
    {
      // Выведем на консоль информацию об исключении.
      Console.WriteLine("To String = {0}",a.ToString());
    }
  }
}
```

Следуя логике программы, результатом работы приложения должна стать строка вида:

```
To String = ...,
```

выведенная на консоль. Но нет! Этого не произойдет. Я был крайне удивлен, когда, запустив приложение, обнаружил, что оно выдает исключение `NullReferenceException`. Оказалось, что среда исполнения преобразует внешние SEH-исключения во внутренние управляемые исключения. Причем этот факт никак не отражен в документации. Внутри исходных кодов среды исполнения я нашел таблицу преобразования SEH-исключений в управляемые исключения (табл. 7.4).

Таблица 7.4. *Таблица преобразования внешних SEH-исключений во внутренние*

Системный код исключения	Класс .NET	Описание
STATUS_FLOAT_INEXACT_RESULT STATUS_FLOAT_INVALID_OPERATION STATUS_FLOAT_STACK_CHECK STATUS_FLOAT_UNDERFLOW	System.ArithmeticException	Арифметическая ошибка
STATUS_FLOAT_OVERFLOW STATUS_INTEGER_OVERFLOW	System.OverflowException	Ошибка переполнения при выполнении арифметических операций
STATUS_FLOAT_DIVIDE_BY_ZERO STATUS_INTEGER_DIVIDE_BY_ZERO	System.DivideByZeroException	Ошибка деления на ноль
STATUS_FLOAT_DENORMAL_OPERAND	System.FormatException	Неправильный формат аргумента
STATUS_ACCESS_VIOLATION	System.NullReferenceException	Ошибка доступа к памяти
STATUS_ARRAY_BOUNDS_EXCEEDED	System.IndexOutOfRangeException	Выход за границу допустимых индексов
STATUS_NO_MEMORY	System.OutOfMemoryException	Нехватка памяти
STATUS_STACK_OVERFLOW	System.StackOverflowException	Переполнение стека

Здесь особо хотелось бы обсудить исключение STATUS_ACCESS_VIOLATION. Оно срабатывает при неудачной попытке обращения к памяти. В нашем примере после преобразования типа исключения, оно достаточно полно отражало суть проблемы. Класс NullReferenceException указывает на то, что произошло обращение к нулевому адресу памяти. Но дело в том, что исключение STATUS_ACCESS_VIOLATION может быть выброшено при обращении к абсолютно любой странице памяти. Все зависит от атрибутов защиты, выставленных для нее. Для того чтобы продемонстрировать это, можно изменить критический участок кода библиотеки на следующий.

```
*(int*)0xffffffff = 5;
```

Наш пример, как и раньше, выдаст исключение NullReferenceException. Что в новом свете выглядит несколько странновато. По большому счету, все

преобразования типов исключений не правомерны, поскольку могут запутать программиста. Среда исполнения должна была выбрасывать только `SEHException`-исключения, не проводя никаких преобразований.

С преобразованием исключений связан весьма неприятный момент. Он заключается в том, что все классы исключений, которые могут быть выброшены из неуправляемого кода, не объединены каким-либо общим предком. Следовательно, отловить все возможные исключения из внешнего неуправляемого кода будет весьма проблематично. Это в принципе можно сделать следующим образом (листинг 7.10).

Листинг 7.10. Шаблон отлова всевозможных исключений, выбрасываемых неуправляемым кодом

```
try
{
        ...
        // Вызов внешнего кода.
        ...
}
catch(ArithmeticException ex)
{
        ...
        // Здесь мы будем обрабатывать арифметические
        // ошибки.
        ...
}
catch(OverflowException ex)
{
        ...
        // Здесь мы будем обрабатывать арифметические
        // ошибки - переполнения
        ...
}
catch(DivideByZeroException ex)
{
        ...
        // Здесь мы будем обрабатывать ошибки деления
        // на ноль.
        ...
}
catch(FormatException ex)
```

```
{
        ...
        // Здесь мы будем обрабатывать ошибки,
        // связанные с неправильным форматом аргументов
        // функций.
        ...
}
catch(NullReferenceException ex)
{
        ...
        // Здесь мы будем обрабатывать ошибки неверного
        // доступа к памяти.
        ...
}
catch(IndexOutOfRangeException ex)
{
        ...
        // Здесь мы будем обрабатывать ошибки выхода
        // за допустимый индекс.
        ...
}
catch(OutOfMemoryException ex)
{
        ...
        // Здесь мы будем обрабатывать связанные
        // с нехваткой памяти.
        ...
}
catch(StackOverflowException ex)
{
        ...
        // Здесь мы будем обрабатывать проблему переполнения
        // стека.
        ...
}
catch(SEHException ex)
{
        ...
        // А здесь мы будем обрабатывать все остальные ошибки.
        ...
}
```

Но согласитесь, код выглядит несколько неуклюже. Хотелось бы чего-то более компактного, менее загромождающего код. Для этого придется воспользоваться фильтром на базе класса `Exception`. Выглядеть это будет так (листинг 7.11).

Листинг 7.11. Более компактный вариант предыдущего шаблона

```
try
{
}
catch (Exception ex)
{
  if (ex is ArithmeticException)
  {
    // Здесь мы будем обрабатывать арифметические
       // ошибки.
  }
  else
    if (...)
    {
      ...
    }
    else
      // Не забудем перевыбросить исключение, которое
      // не смогли обработать.
      throw ex;

}
```

Хотя этот вариант кода и выглядит довольно компактным, но все было бы намного проще, не проводи среда исполнения преобразования исключений.

Три ужасных исключения

Среди системных исключений существует три особых исключения, после возникновения которых восстановления работы программы невозможно.

`ExecutionEngineException`

`OutOfMemoryException`

`StackOverflowException`

Такие исключения используются только самой средой исполнения, прикладные программы не должны использовать их.

Исключение `ExecutionEngineException`, судя по документации, срабатывает только в случае повреждения виртуальной машины .NET. И при нормальных условиях использования среды .NET, вы не должны столкнуться с исключением этого типа. (Хотя надо отметить, что я его видел.) На мой взгляд, в документации данный момент отражен несколько неверно. Если вы обнаружили такое исключение, то, скорее всего, проблема кроется либо в поврежденных метаданных одной из прикладных программ или библиотечных сборок, либо в неисправности подсистемы COM, которая активно используется средой исполнения во внутренних целях.

Исключение `OutOfMemoryException` возникает в двух случаях: когда самой среде исполнения не хватает памяти для внутренних целей или когда сборщик мусора, произведя очередную очистку, не смог освободить достаточно памяти для новых объектов. Такое может происходить в двух случаях. Во-первых, при реальной нехватке оперативной памяти, а также при отсутствии файла подкачки. Во-вторых, при крупных утечках памяти, произошедших в программе.

Примечание

Утечкой памяти называется потеря контроля над некоторыми выделенными участками памяти. Фактически, утечка — это выделение все новых и новых областей памяти и неосвобождение уже использованных. Таким образом, размер памяти, используемый приложением, постоянно увеличивается. Достигнув некоторой критической точки, приложение больше не будет способно функционировать.

Чисто управляемые приложения принципиально не могут вызвать утечки памяти, поскольку они не допущены непосредственно к менеджеру памяти. Вся работа с памятью на уровне управляемого кода производится автоматически самой средой исполнения, прозрачно для программиста. А вот приложения, которые взаимодействуют с неуправляемым кодом, вполне могут вызвать пресловутые утечки памяти. Естественно, что работа с памятью внутри таких участков не контролируется средой исполнения. Это попросту невозможно. Соответственно, при написании приложений, необходимо стараться наиболее аккуратно взаимодействовать с внешним неуправляемым кодом.

Из нашей тройки осталось только одно исключение, — `StackOverflowException`. Исключение вызывается при переполнении стека вызова функций. В стеке располагаются параметры функций, а также их локальные переменные. В документации сказано, что это исключение происходит из-за большой вложенности функций. В общем случае это, конечно, правильно, но в реальной жизни в корне не верно. Так как девяносто девять и девять десятых процента от общего числа подобных исключений происходят по вине срыва функции или группы функции в бесконечную рекурсию. Для большей ясно-

сти произведем расчеты. Пускай размер стека будет равен 1 Мбайту, как у стандартных Win32-приложений. В среднем, каждая функция принимает 4 параметра, что составляет 16 байт для 32-разрядной архитектуры.

Для локальных переменных зарезервируем 200 байт, хотя на самом деле эта цифра намного меньше. Не забудем учесть адрес возврата — 4 байта. Сложим все это, округлим для ровного счета до 250. В нашей идеализированной модели столько байт использует в стеке каждая функция. А теперь узнаем, сколько функций сможет выдержать стек. Для этого разделим 1 Мбайт на 250 байт, получится 4000. То есть стек может выдержать вложенный вызов 4000 тысяч функций, при условии, что каждая из них будет использовать 250 байт. А теперь попытаемся представить себе такую программу. Я такого алгоритма никогда не видел, хотя приходилось сталкиваться и со сверхсложными и большими программами. Остается только вариант неуправляемой рекурсии. Для большей наглядности продемонстрируем на примере (листинг 7.12).

Листинг 7.12. Возникновение исключения `StackOverflowException` из-за срыва функции в бесконечную рекурсию

```
/*
  Листинг 7.12
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Вызовем рекурсивную функцию.
    Some(5);
  }
  // Эта функция будет обеспечивать срыв в бесконечную рекурсию и
  //вызывать переполнение стека.
    public static void Some(int a)
  {
    // Эта функция будет вызывать сама себя,
```

```
    // то есть будет рекурсивной.
    Some(5);
  }
};
```

Сердцем примера является функция `Some`. Она вызывает сама себя, обеспечивая срыв в бесконечную рекурсию.

Через некоторое время, после запуска приложения, на экране должно появиться диалоговое окно, сообщающее о возникновении исключения `StackOverflowException` (рис. 7.3).

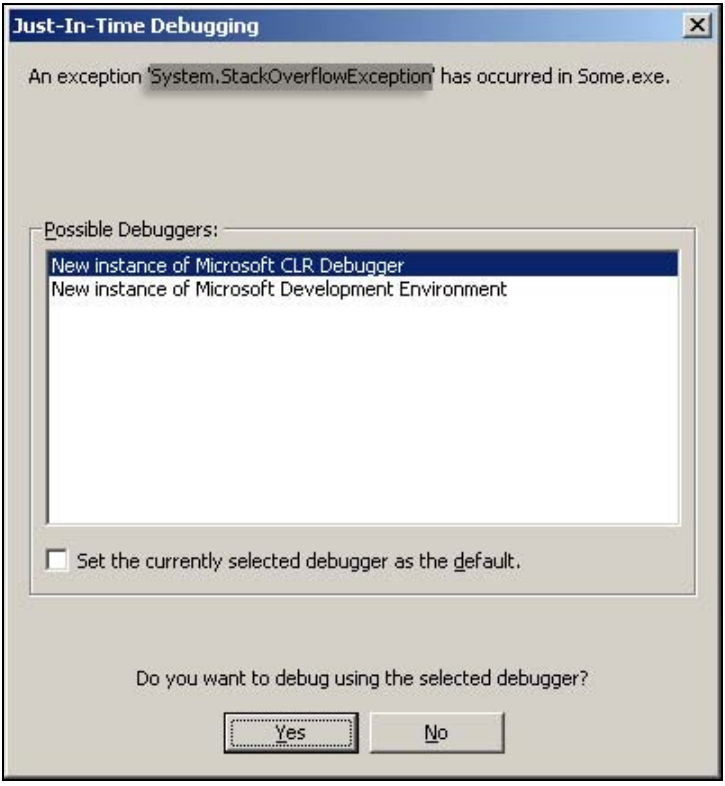


Рис. 7.3. Необработанное исключение `StackOverflowException`

Функция `Some` принимает один аргумент размером 4 байта, а также содержит несколько переменных, поэтому ждать возникновения исключения придется относительно долго. Для того чтобы увеличить скорость пожирания стека, можно увеличить количество параметров, принимаемых функцией, а также ввести дополнительные локальные переменные.

7.5. Создание пользовательских исключений

Прежде чем вводить собственный дополнительный класс исключения, внимательно изучите общую библиотеку классов, может, там уже есть подходящий тип. Но все же иногда бывает необходимо создать собственный класс исключения, хотя это бывает достаточно редко. Сам процесс создания пользовательского класса исключений крайне примитивен. Необходимо лишь учитывать несколько требований, диктуемых со стороны общеязыковой спецификации CLS.

- Все пользовательские классы исключений должны являться потомками класса `System.ApplicationException`.
- Любой класс исключений должен поддерживать три общедоступных конструктора. Они были подробно описаны ранее.
- Если приложение использует сервисы .NET Remoting, то все классы пользовательских исключений обязаны поддерживать сериализацию.
- Имя класса пользовательского исключения должно заканчиваться словом `Exception`.

Приведем пример определения пользовательского исключения (листинг 7.13).

Листинг 7.13. Определение нового пользовательского исключения

```
/*
  Листинг 7.13
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, содержащее сервисы поддержки
// сериализации.
using System.Runtime.Serialization;
// Пометим наш класс при помощи специального атрибута как
// сериализуемый.
[Serializable]
class MyException :
  // Наследуем наш класс от класса ApplicationException
  // и от интерфейса, отвечающего за сериализацию.
  ApplicationException, ISerializable
```

```
{
  // Сначала объявим три общедоступных конструктора.
  public MyException() :
    // Вызовем соответствующий конструктор базового класса.
    base()
  {
  }
  public MyException(String message) :
    // Вызовем соответствующий конструктор базового класса.
    base (message)
  {
  }
  public MyException(String message, Exception innerException) :
    // Вызовем соответствующий конструктор базового класса.
    base (message, innerException)
  {
  }
  // Это дополнительные данные об исключении, ради которых
  // и вводился пользовательский класс исключения.
  protected String myData;
  // Изменим поведение свойства Message, для того чтобы получать
  // полную информацию от метода ToString, который использует
  // это свойство.
  public override String Message
  {
    get {
      // Возвращаем новую строку, состоящую из обычного сообщения
      // и нашего
      return base.Message + " " + myData;
    }
  }
   // Опишем дополнительные конструкторы, для удобства
  // инициализации объекта данного класса.
  public MyException(String message, String _myData) :
    // Позаботимся об инициализации базового класса.
    this(message)
  {
    // Установим пользовательские данные.
    myData = _myData;
```

```
    }
    public MyException(String message, String _myData, Exception
_innerException) :
       // Позаботимся об инициализации базового класса.
       this(message, _innerException)
    {
      // Установим пользовательские данные.
      myData = _myData;
    }
    // Опишем два дополнительных метода, отвечающих за сериализацию
    // введенного объекта.
    // Закрытый конструктор, который используется в служебных целях.
    private MyException(SerializationInfo info, StreamingContext context)
      : base (info, context)
    {
      // Прочитаем пользовательские данные (строку).
      myData = info.GetString("myData");
    }
    void ISerializable.GetObjectData( SerializationInfo info, Streaming-
Context context)
    {
      // Запишем пользовательские данные.
      info.AddValue("myData", myData);
      // Запишем данные базового класса.
      base.GetObjectData(info,context);
    }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
  }
}
```

Если приложение не работает с .NET Remoting, то создание собственных классов исключений существенно упростится. Можно будет опустить часть кода, отвечающую за сериализацию объектов. Иногда бывает целесообразно вводить не один класс исключения, а целую иерархическую группу.

Это позволит более гибко настраивать обработчики исключений, так как некоторые из них смогут перехватывать сразу несколько типов исключений.

При создании пользовательских классов исключений обязательно необходимо помнить о том, что все пользовательские исключения должны быть потомками класса `System.ApplicationException`. Второе требование заключается в том, что любой класс исключения должен обязательно поддерживать три общедоступных конструктора, которые были подробно рассмотрены ранее.

7.6. Поиск необходимого обработчика исключения

Механизм поиска обработчика в пределах одного защищенного блока был описан ранее. Но что произойдет, если в защищенном блоке не окажется подходящих обработчиков или в функции вообще нет защищенного блока? В этом случае будет запущен процесс трассировки стека. Для того чтобы объяснить механизм работы этого процесса, сначала рассмотрим устройство и архитектуру стека функций.

Стек функций

Все без исключения приложения вызывают функции: внутренние и внешние. Под внутренними подразумеваются функции, расположенные в самой программе, а под внешними — экспортируемые из сторонних библиотек. Большинство функций принимают параметры, которые позволяют управлять их работой. Они помещаются в специальную область памяти — стек вызова функций.

Стек — это специальная область памяти, организованная по принципу рожка от автоматического оружия, к примеру, от автомата Калашникова. Перед тем как использовать автомат, вы помещаете пули в рожок, затем они поочередно в обратном порядке выталкиваются в ствол, дальнейшая их судьба нас интересовать не будет. Точно так же устроен и стек, для работы с ним есть два типа операций: помещающие в него параметр и извлекающие параметр.

Примечание

Операцию проталкивания параметра в стек назовем `push`, а операцию извлечения `pop`.

Также, перед вызовом функции, в стек помещается дополнительная информация об адресе возврата. Он позволит после исполнения функции вернуться обратно в данный участок кода, чтобы продолжить исполнение программы. Это схематично изображено на рис. 7.4.

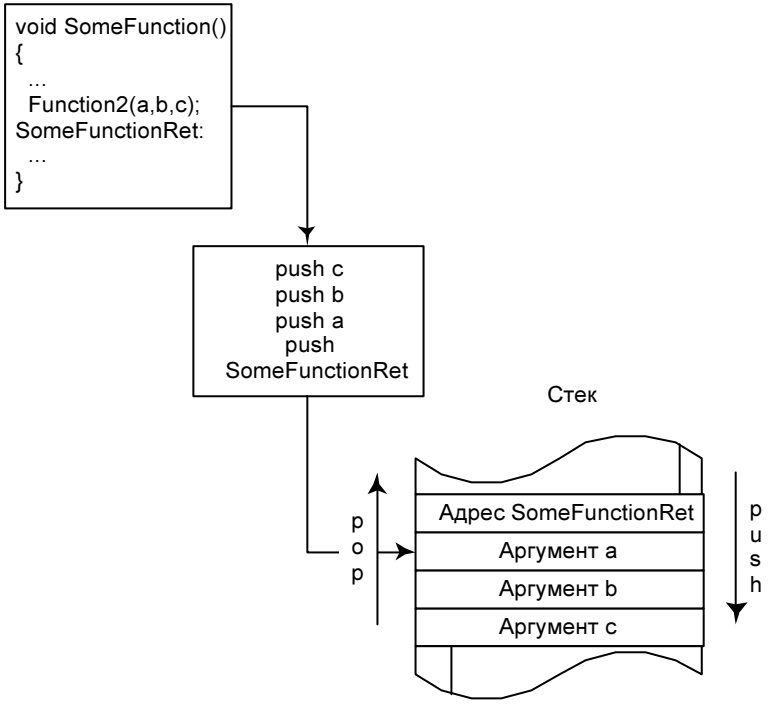


Рис. 7.4. Устройство стека

После того как вызванная функция отработает, она вытолкнет из стека все свои параметры, а также адрес возврата, по которому передаст управление. В нашем примере, после окончания работы функции `Function2`, она вытолкнет свои оба параметра из стека и передаст управление по адресу `SomeFunctionRet`.

К сожалению, организация стека в управляемых приложениях полностью не документирована. Поэтому о точном его строении можно только догадываться или проводить специальные трудоемкие исследования. Но это и не нужно, поскольку все сервисы для работы со стеком изначально предоставлены в общей библиотеке классов. Самому ничего писать не придется. Хотя об общем строении и устройстве стека знать обязательно нужно. В общем виде, он выглядит так (рис. 7.5).

Обратите внимание на адреса возврата, они косвенно указывают на функции, которые привели к вызову данной. Таким образом, можно просканировать стек и построить очередь вызова функций. Этот процесс называется трассировкой стека.

Примечание

Группа параметров функции совместно с адресом возврата называется стековым кадром. То есть весь стек разбит на кадры, каждый из которых соответствует некоторой функции.

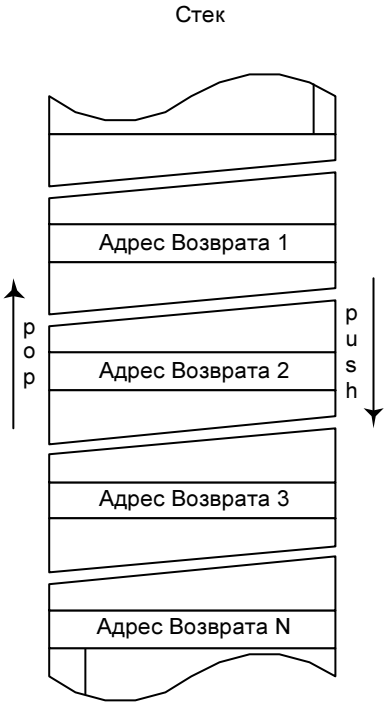


Рис. 7.5. Общий вид внутреннего устройства стека

Для большей наглядности продемонстрируем трассировку стека на примере (листинг 7.14). Воспользуемся классом `StackTrace`, который отвечает за трассировку стека.

Листинг 7.14. Пример трассировки стека

```
/*
  Листинг 7.14
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Подключаем пространство имен, в котором расположены сервисы отладки
// и диагностирования приложений.
using System.Diagnostics;
// Подключаем пространство имен, в котором расположены сервисы,
// поддерживающие технологию отражения.
```

```
using System.Reflection;
// Основной класс приложения.
class Application
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Сразу передаем управление вложенной функции.
    Some1();
  }
  // Далее введено несколько дополнительных функций
  // для того, чтобы организовать стек вызова.
  public static void Some1()
  {
    try
    {
      Some2();
    }
    catch (Exception ex)
    {
    }
  }
  public static void Some2()
  {
    Console.WriteLine("Method Some2 was called.");
    Some3();
  }
  // Эта функция будет располагаться на вершине стека
  // вызовов, именно она и будет проводить трассировку стека.
  public static void Some3()
  {
    // Создаем новый объект класса StackTrace.
    StackTrace st = new StackTrace();
    Console.WriteLine("\nStack trace result");
    // Проходим по всем стековым кадрам.
    for ( int i = 0; i < st.FrameCount; i++)
    {
      // Запрашиваем текущий кадр стека
      StackFrame sf = st.GetFrame(i);
```

```
        // Определяем, к какому методу относится данный фрейм
        MethodBase mb = sf.GetMethod();
        // Выводим информацию по данному методу
        Console.WriteLine("Method {0}",mb.ToString());
      }
    }
};
```

В результате работы программы на консоль будут выведены следующие строки.

```
Method Some2 was called
Stack trace result
Method Void Some3()
Method Void Some1()
Method Void Main()
```

Правда, здесь можно усмотреть одну неточность. В списке отсутствует метод `Some2`, который вывел на консоль первую из приведенных строк.

Примечание

Весь код этой программы можно заменить одним-единственным обращением к свойству `System.Environment.StackTrace`. Оно вернет строку, в которой будет указан результат трассировки стека.

По умолчанию при трассировке методов пропускаются все методы, не содержащие защищенных участков кода, за исключением первого (`Main`) и последнего метода (`Some3`). Для того чтобы получить информацию обо всех методах, необходимо построить отладочную версию приложения. Сделать это можно, указав компилятору ключ `/debug`:

```
csc Some.cs /debug
```

После этого результат работы станет следующим.

```
Method Some2 was called

Stack trace result
Method Void Some3()
Method Void Some2()
Method Void Some1()
Method Void Main()
```

Лог уже стал гораздо более информативен. Но мы этим не ограничимся. Каждое приложение в отладочной версии содержит обширную информацию

об исходном коде данной программы. Нас будут интересовать номера строк методов, а также файлы, в которых они описаны. Для этого придется воспользоваться методами `GetFileName()` и `GetFileLineNumber()` класса `StackFrame`. Модифицируем предыдущий пример (листинг 7.15).

Листинг 7.15. Более полная информация по трассировке стека

```
/*
  Листинг 7.15
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Подключаем пространство имен, в котором расположены сервисы отладки
// и диагностирования приложений.
using System.Diagnostics;
// Подключаем пространство имен, в котором расположены сервисы,
// поддерживающие технологию отражения.
using System.Reflection;
// Основной класс приложения.
class Application
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Сразу передаем управление вложенной функции.
    Some1();
  }
  // Далее введены несколько дополнительных функций,
  // чтобы организовать стек вызова.
  public static void Some1()
  {
    try
    {
      Some2();
    }
    catch (Exception ex)
```

```
    {
    }
  }
  public static void Some2()
  {
    Console.WriteLine("Method Some2 was called.");
    Some3();
  }
  // Эта функция будет располагаться на вершине стека
  // вызовов, именно она и будет проводить трассировку стека.
  public static void Some3()
  {
    // Создаем новый объект класса StackTrace.
    StackTrace st = new StackTrace();
    Console.WriteLine("\nStack trace result");
    // Проходим по всем стековым кадрам.
    for ( int i = 0; i < st.FrameCount; i++)
    {
      // Запрашиваем текущий кадр стека.
      StackFrame sf = st.GetFrame(i);
      // Определяем, к какому методу относится данный фрейм.
      MethodBase mb = sf.GetMethod();
      // Выводим информацию по этому методу.
      Console.WriteLine("Method {0}",mb.ToString());
      // Выводим информацию о том, в каком файле расположен
      // метод.
      Console.WriteLine("FileName:{0}",sf.GetFileName());
      // Выведем номер строки, в которой следует искать метод.
      Console.WriteLine("LineNumber:{0}\n",sf.GetFileLineNumber());
    }
  }
};
```

В результате работы программы будет получен следующий лог.

```
Method Some2 was called
Stack trace result
Method Void Some3()
FileName:d:\...\Some.cs
LineNumber:31
```

```
Method Void Some2()
FileName:d:\ ...\Some.cs
LineNumber:27
Method Void Some1()
FileName:d:\ ...\Some.cs
LineNumber:16
Method Void Main()
FileName:d:\ ...\Some.cs
LineNumber:10
```

А что будет, если запустить нашу программу в Release версии построения? В этом случае, вид лога кардинально изменится.

```
Method Some2 was called

Stack trace result
Method Void Some3()
FileName:
LineNumber:0

Method Void Some1()
FileName:
LineNumber:0

Method Void Main()
FileName:
LineNumber:0
```

Причем если пропажа имен файлов и номеров строк ясна, то почему в лог не попадает функция `Some2`, совершенно неясно. Ответ довольно прост. Трассировка в процессе обработки исключений используется для поиска необходимых обработчиков исключений. Если в текущей функции обработчик найден не будет, то будет запущен процесс трассировки стека. Среда исполнения будет просматривать все вышележащие функции на предмет поиска подходящего обработчика. Естественно, что будут учитываться только те функции, которые содержат защищенные блоки. Другие рассматривать не имеет смысла, поскольку они не могут ничем помочь в обработке исключения. Именно поэтому из лога исчез метод `Some2()`, он не содержал обработчика исключения. А вот метод `Some1` имел его, хотя ничем другим от него не отличался.

Процесс сканирования стека функций на предмет поиска обработчика довольно прост, исключая один неприятный момент. В процессе поиска обработчика

среде исполнения приходится учитывать попадание обращения к функциям в контекст защищенного блока. Приведем пример.

```
void Some1()
{
  try
  {
    ...
  }
  catch (Exception ex)
  {
    ...
  }
  Some2();
}
void Some2()
{
  throw new Exception();
}
```

Функция `Some2` не попадает в контекст защищенного блока, расположенного в функции `Some1`. Среда исполнения должна будет определить, захватывает такой защищенный блок исключение или нет. К счастью программистов, этот механизм глубоко скрыт в недрах среды исполнения и им непосредственно с ним сталкиваться не придется. Хотя знать, как он работает, довольно полезно, это может сильно помочь при устранении проблем во время работы с исключениями.

Фильтр необработанных исключений

Если среда исполнения, просканировав стек функций, не обнаружит подходящего обработчика, то будет вызван фильтр необработанных исключений. По умолчанию он вызывает всем знакомое диалоговое окно, сообщающее о произошедшем исключении (рис. 7.6).

Такое диалоговое окно будет показано только в случае, если приложение управляемое. Для Win32-приложений вид окна будет несколько другим (рис. 7.7).

Или же таким, для систем класса 9x (рис. 7.8).

Любое из представленных окон, вызывает у большинства пользователей малоприятные чувства, начиная от легкого недоумения и заканчивая ужасом. Мечта любого программиста добиться того, чтобы никто и никогда не увидел такого окна при работе с его программами. Оказывается, этого не так уж

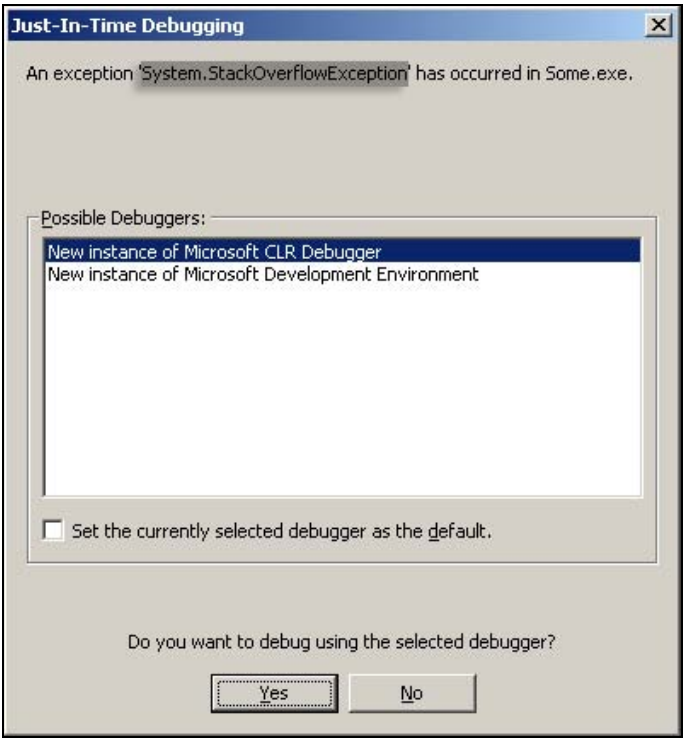


Рис. 7.6. Окно, выводимое фильтром необработанных исключений

Рис. 7.7. Необработанное исключение для классических Win32-приложений в Windows XP

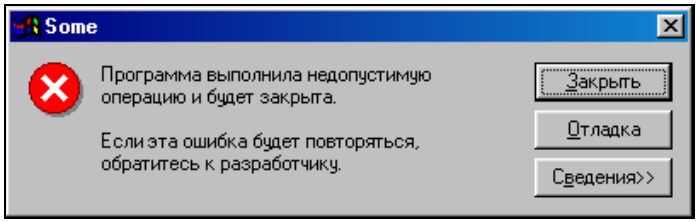


Рис. 7.8. Необработанное исключение в классических Win32-приложениях в системах класса 9x

и сложно добиться. Всего существует два способа. Первый — написать абсолютно защищенное приложение, в котором нет ошибок, а также предусмотрены все гадости, которые может подкинуть операционная система, внешние модули и сами пользователи. Второй — зарегистрировать свой обработчик в фильтре необработанных исключений и в нем подавить вывод диалогового окна на экран. Каким из путей идти, выбирать будете вы сами. Здесь будет обсуждаться лишь второй вариант.

Для этого класс `AppDomain` предоставляет событие `UnhandledException`. Вы сможете перехватывать необработанные события, установив свой обработчик на это событие. Прототип события представлен ниже.

```
public event UnhandledExceptionEventHandler AppDomain.UnhandledException;
```

А прототип делегата события следующий.

```
public delegate void UnhandledExceptionEventHandler(
  // Объект, который вызвал исключение.
  object sender,
  // Объект, содержащий информацию об исключении.
  UnhandledExceptionEventArgse
)
```

Ключевым здесь является второй параметр делегата. Это переменная класса `UnhandledExceptionEventArgs`, описанного в табл. 7.5.

Таблица 7.5. *Класс* `System.UnhandledExceptionEventArgs`

Член класса		Описание
	`ExceptionObject`	Объект, описывающий произошедшее необработанное исключение
	`IsTerminating`	Позволяет определить, будет ли закрыта программа после исполнения обработчика

Свойство ExceptionObject

```
public object UnhandledExceptionEventArgs.ExceptionObject {get;}
```

Свойство позволяет получить объект, описывающий исключение. Это тот самый объект, который был указан в качестве параметра оператора `throw` или был сформирован средой исполнения автоматически, непосредственно перед выбросом исключения.

Свойство IsTerminating

```
public bool UnhandledExceptionEventArgs.IsTerminating {get;}
```

Свойство позволяет определить, будет ли закрыта программа после выхода из функции обработчика необработанного исключения. С первого взгляда может удивить наличие такого свойства, поскольку на момент вызова функции обработчика стек функций разобран, а все объекты, находящиеся в нем, уничтожены. Казалось бы, обратного пути нет и дальнейшее исполнение приложения бессмысленно. Однако существуют многопоточные приложения, для которых закрытие одного потока не является критичным. Возникновение необработанного исключения в любом потоке, кроме главного, приведет к вызову обработчика и последующему закрытию потока. Но при этом само приложение продолжит свою работу. Далее будет показан пример такого приложения, подвергающегося атаке одного из ужасных исключений — `StackOverflowException` и продолжающего свою работу.

Использование фильтра необработанных исключений

А теперь рассмотрим простейший пример использования фильтра необработанных исключений (листинг 7.16).

Листинг 7.16. Простейший пример установки фильтра необработанных исключений

```
/*
  Листинг 7.16
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
```

```
  // Функция, которая будет отвечать за работу с необработанными
  // исключениями.
  // UEF — это сокращение от английского Unhandled Exception Filter
  // (фильтр необработанных исключений).
  public static void UEF(object sender,UnhandledExceptionEventArgs args)
  {
    // Сообщим пользователю о том, что произошло необработанное
    // исключение.
    Console.WriteLine("Unhandled Exception detected.");
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Устанавливаем обработчик для "необработанных" исключений,
    // его будет обслуживать функция UEF нашего приложения.
    AppDomain.CurrentDomain.UnhandledException +=
      new UnhandledExceptionEventHandler(UEF);
    // Вызываем исключение, которое не будет явно обрабатываться
    // нашим приложением.
    throw new Exception();
  }
};
```

После выброса исключения в функции `Main` нашей программы, среда исполнения попытается найти обработчик исключения в данной функции. Но там его не окажется, тогда она запустит процедуру трассировку стека, что также будет бессмысленным, поскольку функция находится на дне стека вызовов.

Примечание

На самом деле, `Main` находится далеко не на дне стека вызовов функции, под ней находятся внутренние сервисы среды исполнения. Но с пользовательской точки зрения, она является нижней.

Придя к выводу о том, что необходимых явных обработчиков в программе не присутствует, среда исполнения передаст управление фильтру необработанных исключений. По идее, в результате работы программы, на консоль должна быть выведена следующая строка.

```
Unhandled Exception detected.
```

На этом работа приложения должна завершиться. Но вместо этого на экране появляется окно, сообщающее о произошедшем исключении. А установленный

в программе обработчик вызывается только в случае отказа от отладки приложения. На системах класса 9x ситуация еще интересней. После возникновения необработанного исключения на экране появилось диалоговое окно, предлагающее либо завершить работу приложения, либо приступить к его отладке (рис. 7.9).

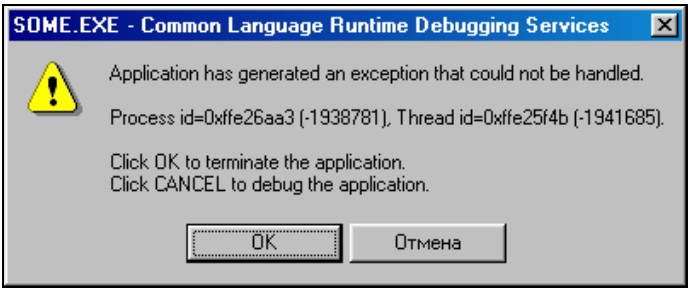


Рис. 7.9. Сообщение о необработанном исключении в управляемых приложениях, работающих под управлением операционной системы класса 9x

Предложение об отладке выглядело весьма странным, поскольку отладчик в системе установлен не был.

Примечание

Для тестирования использовалась чистая система Windows 98. Единственной установленной на нее программой была среда исполнения.

После выбора отладки приложения (кнопка **Отмена**), среда исполнения сообщила о том, что отладчик не найден (рис. 7.10).

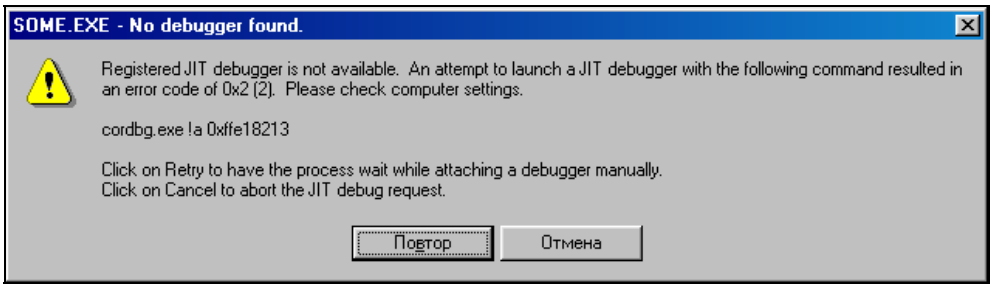


Рис. 7.10. Предложение среды исполнения подключить отладчик вручную

Диалоговое окно предлагает заморозить процесс, а отладчик подключить вручную после или немедленно завершить процесс, прервав запрос на отладку.

Таким образом, фильтрация необработанных исключений в приложениях .NET теряет первоначальный смысл. Прямым путем скрыть от пользователя

возникновение необработанного исключения не удастся. А пользователи, как известно, на появление подобных диалоговых окон реагируют крайне отрицательно. Следовательно, желательно предотвратить появление таких предупреждений.

Для приложений, написанных на WinAPI, проблем нет, для них можно с легкостью подавить появление ненавистного окна. Дабы подтвердить справедливость вышесказанного, приведем пример (листинг 7.17).

Листинг 7.17. Подавления сообщения о возникших необработанных исключениях

```
/*
  Листинг 7.17
  File:   Some.cpp
  Author: Дубовцев Алексей
*/
// Подключаем стандартные заголовочные файлы Windows API.
#include <windows.h>
// Подключаем сервисы ввода/вывода из стандартной библиотеки.
#include <stdio.h>
// Функция фильтр необработанных исключений.
LONG ExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo)
{
  // Сообщаем пользователю о произошедшем исключении
  printf("Unhandled Exception detected.");
  // Сообщаем операционной системе о том, что мы обработали
  // исключение и ей больше не надо заботиться о нем и показывать
  // любые диалоговые окна
  return EXCEPTION_EXECUTE_HANDLER;
}
// Точка входа в приложение
void main()
{
  // Устанавливаем фильтр необработанных исключений.
  SetUnhandledExceptionFilter(
    (LPTOP_LEVEL_EXCEPTION_FILTER)ExceptionFilter);
  // Выбросим исключение, которое не будет явным образом перехвачено
   RaiseException(5,EXCEPTION_NONCONTINUABLE,0,0);
   // Предыдущую строку безболезненно можно заменить на что-нибудь
   // вроде
   *(int*)0xffffff = 5;
}
```

В результате запуска приложения на консоль будет выведена строка.

```
Unhandled Exception detected.
```

После чего работа приложения будет завершена. Но никаких сообщений операционной системы выведено не будет.

Добиться такого поведения от управляемых .NET-приложений возможно, если знать об одном нюансе. Вся хитрость заключается в том, что среда исполнения по-разному относится к необработанным исключениям, возникшим в основном потоке приложения и в остальных потоках. Если необработанное исключение возникает в основном потоке приложения или в одном из неуправляемых, среда исполнения выводит диалоговое окно, вызывает фильтр необработанных исключений, а затем закрывает приложение. Если же исключение происходит в одном из других потоков, то будет закрыт только он, а само приложение будет работать дальше, без этого потока. Рассмотрим многопоточный пример, реализующий такую схему обработки исключений (листинг 7.18).

Листинг 7.18. Многопоточная схема обработки исключений

```
/*
  Листинг 7.18
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Подключаем пространство имен, в котором содержатся классы для работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Фильтр необработанных исключений.
  public static void UEF(object sender,UnhandledExceptionEventArgs args)
  {
    // Сообщим пользователю о произошедшем необработанном
    // исключении.
    Console.WriteLine("Unhandled Exception detected.");
    // Сообщим пользователю, будет закрыто все приложение или нет.
    Console.WriteLine("IsTerminating = {0}",args.IsTerminating);
  }
```

```
  // Функция фонового потока.
  static void ThreadProc()
  {
    Console.WriteLine("Second thread started.");
    // Выбрасываем исключение, которое не будет обработано.
    throw new System.Exception();
  }
  // Точка входа в приложение.
  static void Main()
  {
    // Создаем объект потока.
    Thread th = new Thread( new ThreadStart(ThreadProc) );
    // Присоединяем функцию обработчик к событию.
    Thread.GetDomain().UnhandledException +=
new UnhandledExceptionEventHandler(UEF);
    // Запускаем второй поток
    th.Start();
    // Делаем нечто, для того чтобы показать, что приложение еще
    // работает.
    for (;;)
    {
      Console.WriteLine("Hello, World!");
      // Чуть-чуть задержимся, дабы не засорять консоль лишней
      // информацией.
      Thread.Sleep(2000);
    }
  }
};
```

После запуска приложения на консоль будут выведены следующие строки.

```
Second thread started
Unhandled Exception detected
IsTerminating = False
Hello, World!
Hello, World!
^C
```

Обратите внимание на последнюю строку "^C", она автоматически выводится при нажатии комбинации клавиш **Ctrl-Break**. Данная комбинация необходима для завершения работы приложения, поскольку оно будет с проме-

жутком в две секунды безостановочно выводить на консоль строку "Hello, World!". В первых строках лога отражены самые интересные моменты из жизни приложения. Сначала запускается второй поток, затем в нем происходит необработанное исключение, в результате которого второй поток прекращает свою работу. Но тем временем, основной поток приложения, как ни в чем ни бывало, продолжает свою работу.

Подавление ужасного исключения

Ранее рассказывалось о трех ужасных исключениях, при возникновении которых дальнейшая работа приложений невозможна. На самом деле, одно их этих исключений все-таки можно подавить. Это `StackOverflowException`. Исключение возникает при переполнении стека функций. А поскольку, в многопоточных приложениях каждому потоку выделен собственный стек, то можно закрыть проблемный поток и продолжить работу основного приложения. Пример подавления исключения `StackOverflowException` приведен в листинге 7.19.

Листинг 7.19. Подавление исключения `StackOverflowException` для многопоточного приложения

```
/*
  Листинг 7.19
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Подключаем пространство имен, в котором содержатся классы для работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Функция фильтр необработанных исключений.
  public static void UEF(object sender,UnhandledExceptionEventArgs args)
  {
    // Сообщим пользователю о том, что произошло необработанное
    // исключение.
    Console.WriteLine("Unhandled Exception detected.");
  }
```

```
  public static void Hello(int i)
  {
    // Войдем в бесконечную рекурсию, для того чтобы вызвать
    // исключение StackOverflowException.
    Hello(i);
  }
  public static void ThreadProc()
  {
    // Обратимся к функции, обеспечивающий вход в бесконечную
    // рекурсию.
    Hello(4);
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим новый объект потока.
    Thread th = new Thread(new ThreadStart(ThreadProc));
    // Установим фильтр необработанных исключений.
    AppDomain.CurrentDomain.UnhandledException +=
      new UnhandledExceptionEventHandler(UEF);
    // Запустим второй поток.
    th.Start();
    // Имитируем работу, чтобы пользователь знал,
    // что основной поток приложения еще работает.
    for (;;)
    {
      Console.WriteLine("Hello, World!");
      // Чуть-чуть задержимся, дабы не засорять консоль лишней
      // информацией.
      Thread.Sleep(3000);
    }
  }
}
```

В результате запуска приложения на консоль будет выведена строка.

```
Unhandled Exception detected
```

А основной поток приложения, несмотря на это, будет постоянно выводить на консоль сообщение.

```
Hello, World!
```

Метод, безусловно, хорош, только использовать его нужно очень осторожно. При закрытии любого потока необходимо учитывать его персональные особенности. Если некоторые их потоков можно безболезненно перезапустить и работа приложения нарушена не будет, то другие более критичные потоки трогать не следует. Все зависит от конкретной архитектуры приложения. Общие рекомендации здесь дать крайне сложно.

7.7. Внутри механизма обработки исключений

Механизм обработки исключений в среде .NET построен на SEH-исключениях операционной системы. Благодаря этому факту можно перехватить управляемые исключения из неуправляемого кода. Продемонстрируем это на примере, для чего понадобится двухмодульное приложение: управляемое приложение, плюс динамическая библиотека, написанная на родном Windows API, без использования технологий .NET. Управляемое приложение будет обращаться к динамической библиотеке и вызывать управляемое исключение, а та, в свою очередь, будет перехватывать управляемое исключение. Код приложения представлен в листинге 7.20.

Листинг 7.20. Приложение, демонстрирующее внутренний механизм работы управляемых исключений

```
/*
  Листинг 7.20
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключаем основное пространство имен общей библиотеки классов.
using System;
// Подключаем пространство имен, содержащее сервисы для взаимодействия
// с операционной системой.
using System.Runtime.InteropServices;
// Опишем делегат для обратного взаимодействия с динамической
//библиотекой, то есть для того чтобы передать ссылку
// на управляемую функцию.
public delegate void CallBack();
// Основной класс приложения.
class App
{
  // Импортируем функцию из динамической библиотеки.
```

```
  [DllImport("Some.dll")]
  public static extern void SEHCatch(CallBack param);
  // Точка входа в приложение.
  public static void Main()
  {
    // Создаем делегат, ссылающийся на функцию, выбрасывающую управляемое
    // исключение.
    CallBack hello = new CallBack(App.ThrowException);
    // Вызываем из динамической библиотеки неуправляемую функцию.
    SEHCatch(hello);
  }
  // Назначение функции — выбросить управляемое .NET-исключение.
  public static void ThrowException()
  {
  // Выбрасываем тривиальное .NET-исключение, основного базового типа.
    throw new Exception();
  }
};
```

То есть взаимодействие с неуправляемым кодом будет производиться по схеме, представленной на рис. 7.11.

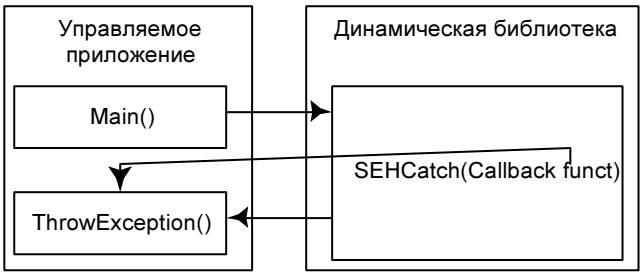


Рис. 7.11. Схема взаимодействия между управляемым приложением и динамической библиотекой

Управляемое приложение вызовет функцию `SEHCatch` из динамической библиотеки и передаст ей ссылку на управляемую функцию `ThrowException`, основное назначение которой — выбрасывать исключение. А неуправляемая функция `SEHCatch` вызовет `ThrowException` и перехватит управляемое исключение. Как это будет сделано, показано в листинге 7.21.

Листинг 7.21. Ортодоксальная динамическая библиотека, перехватывающая управляемое исключение

```
/*
  Листинг 7.21
  File:   Some.cpp
  Author: Copyright (C) 2002 Dubovcev Aleksey
*/
// Подключим стандартный заголовочный файл для Windows API.
#include <windows.h>
#include <stdio.h>
// Функция фильтр для исключений.
int MyFilter(LPEXCEPTION_POINTERS lpExInfo)
{
  // Выведем на консоль количество параметров данного исключения.
  printf("Number of parameters %d\n",
 lpExInfo->ExceptionRecord->NumberParameters);
   // Выведем значения этих параметров, если они присутствуют.
   for (int i = 0; i < lpExInfo->ExceptionRecord->NumberParameters; i++)
    printf("0x%x\n",lpExInfo->ExceptionRecord->ExceptionInformation[i]);
  // А здесь есть две альтернативы.
  // Обработаем исключения сами.
  return EXCEPTION_EXECUTE_HANDLER;
  // Пропустим исключение дальше.
  return EXCEPTION_CONTINUE_SEARCH;
}
// Неуправляемая функция, которая будет вызвана из управляемого кода.
void SEHCatch(DWORD dwAddress)
{
  // Введем защищенный блок.
  __try
  {
    __asm {
      // Вызовем управляемую функцию ThrowException.
      call dwAddress
    }
  }
  // Обработчик исключений.
  __except(MyFilter(GetExceptionInformation()))
```

```
  {
    // В случае если фильтр примет решение об обработке
    // исключения, информируем пользователя о том, что
    // имело место быть исключение,
    // иначе код попросту не получит управления.
    printf("Exception has been catched\n");
  }
}
```

Помимо исходного кода, для компиляции библиотеки понадобится специальный def-файл, в котором будут указаны экспортируемые функции (листинг 7.22).

Листинг 7.22. Файл, указывающий функции, которые должны быть экспортированы из динамической библиотеки

```
;
; Листинг 7.22
; File:   Some.def
; Author: Дубовцев Алексей
;
; Имя динамической библиотеки.
LIBRARY Some.dll
; Раздел со списком экспортируемых функций.
EXPORTS
  SEHCatch
```

Скомпилировать библиотеку можно при помощи двух следующих команд.

```
cl Some.cpp /c
link Some.obj /dll /def:Some.def
```

В результате работы программы на консоль будут выведены следующие строки.

```
Number of parameters 0
Exceptin has been cathced
```

Исключение было успешно обработано. Можно попытаться достать информацию о произошедшем исключении. При помощи стандартных параметров, передаваемых в структуре `EXCEPTION_POINTERS`, однако этого сделать не удастся, о чем сигнализирует строка **"Number of parameters 0"**. Также можно попытаться узнать информацию об исключении при помощи интерфейса `IErrorInfo`, запрошенного при помощи функции `GetErrorInfo`. Но это

также не приведет ни к чему хорошему, поскольку информация подобного типа не предоставляется. На самом деле, это просчет программистов, создававших виртуальную машину .NET. Они просто забыли сделать преобразование информации об исключении для такого случая. Им просто в голову не пришло, что из динамической библиотеки кто-то может запросить эту информацию, ведь для объектов COM такая возможность была реализована.

При взаимодействии с приложениями COM, происходящее исключение автоматически преобразуется в специальный код ошибки HRESULT, по которому можно определить, что за проблема возникла. Также предоставляется информация при помощи интерфейса `IErrorInfo`. Таблица соответствий кодов HRESULT и классов исключений приведена ниже.

Таблица 7.6. *Соответствие между классами исключений общей библиотеки классов и кодами ошибок COM*

Код HRESULT	Тип исключения .NET
MSEE_E_APPDOMAINUNLOADED	AppDomainUnloadedException
COR_E_APPLICATION	ApplicationException
COR_E_ARGUMENT или E_INVALIDARG	ArgumentException
COR_E_ARGUMENTOUTOFRANGE	ArgumentOutOfRangeException
COR_E_ARITHMETIC ERROR_ARITHMETIC_OVERFLOW	ArithmeticException
COR_E_ARRAYTYPEMISMATCH	ArrayTypeMismatchException
COR_E_BADIMAGEFORMAT	BadImageFormatException
ERROR_BAD_FORMAT	
COR_E_COMEMULATE_ERROR	COMEmulateException
COR_E_CONTEXTMARSHAL	ContextMarshalException
COR_E_CORE	CoreException
NTE_FAIL	CryptographicException
COR_E_DIRECTORYNOTFOUND	DirectoryNotFoundException
ERROR_PATH_NOT_FOUND	
COR_E_DIVIDEBYZERO	DivideByZeroException
COR_E_DUPLICATEWAITOBJECT	DuplicateWaitObjectException
COR_E_ENDOFSTREAM	EndOfStreamException
COR_E_TYPELOAD	EntryPointNotFoundException
COR_E_EXCEPTION	Exception
COR_E_EXECUTIONENGINE	ExecutionEngineException

Таблица 7.6 *(продолжение)*

Код HRESULT	Тип исключения .NET
COR_E_FIELDACCESS	FieldAccessException
COR_E_FIELDACCESS	FileNotFoundException
COR_E_FILENOTFOUND ERROR_FILE_NOT_FOUND	FileNotFoundException
COR_E_FORMAT	FormatException
COR_E_INDEXOUTOFRANGE	IndexOutOfRangeException
COR_E_INVALIDCAST E_NOINTERFACE	InvalidCastException
COR_E_INVALIDCOMOBJECT	InvalidComObjectException
COR_E_INVALIDFILTERCRITERIA	InvalidFilterCriteriaException
COR_E_INVALIDOLEVARIANTTYPE	InvalidOleVariantTypeException
COR_E_INVALIDOPERATION	InvalidOperationException
COR_E_IO	IOException
COR_E_MEMBERACCESS	AccessException
COR_E_METHODACCESS	MethodAccessException
COR_E_MISSINGFIELD	MissingFieldException
COR_E_MISSINGMANIFESTRESOURCE	MissingManifestResourceException
COR_E_MISSINGMEMBER	MissingMemberException
COR_E_MISSINGMETHOD	MissingMethodException
COR_E_MULTICASTNOTSUPPORTED	MulticastNotSupportedException
COR_E_NOTFINITENUMBER	NotFiniteNumberException
E_NOTIMPL	NotImplementedException
COR_E_NOTSUPPORTED	NotSupportedException
COR_E_NULLREFERENCE E_POINTER	NullReferenceException
COR_E_OUTOFMEMORY E_OUTOFMEMORY	OutOfMemoryException
COR_E_OVERFLOW	OverflowException
COR_E_PATHTOOLONG ERROR_FILENAME_EXCED_RANGE	PathTooLongException
COR_E_RANK	RankException
COR_E_REFLECTIONTYPELOAD	ReflectionTypeLoadException

Таблица 7.6 (окончание)

Код HRESULT	Тип исключения .NET
COR_E_REMOTING	RemotingException
COR_E_SAFEARRAYTYPEMISMATCH	SafeArrayTypeMismatchException
COR_E_SECURITY	SecurityException
COR_E_SERIALIZATION	SerializationException
COR_E_STACKOVERFLOW ERROR_STACK_OVERFLOW	StackOverflowException
COR_E_SYNCHRONIZATIONLOCK	SynchronizationLockException
COR_E_SYSTEM	SystemException
COR_E_TARGET	TargetException
COR_E_TARGETINVOCATION	TargetInvocationException
COR_E_TARGETPARAMCOUNT	TargetParameterCountException
COR_E_THREADABORTED	ThreadAbortException
COR_E_THREADINTERRUPTED	ThreadInterruptedException
COR_E_THREADSTATE	ThreadStateException
COR_E_THREADSTOP	ThreadStopException
COR_E_TYPELOAD	TypeLoadException
COR_E_TYPEINITIALIZATION	TypeInitializationException
COR_E_VERIFICATION	VerificationException
COR_E_WEAKREFERENCE	WeakReferenceException
COR_E_VTABLECALLSNOTSUPPORTED	VTableCallsNotSupportedException
Все другие коды HRESULT	COMException

Данные коды ошибок определены в заголовочном файле CorError.h, поставляемом вместе с Framework SDK. Их очень удобно использовать при взаимодействии с управляемым кодом из классических COM-объектов.

7.8. На что следует обратить внимание при работе с исключениями

В разделе будет дано несколько рекомендаций по работе с исключениями. Хочется обратить внимание, что это рекомендации, а не строгие предписания, которые необходимо беспрекословно выполнять. В отдельных случаях можно смело отступать от данных здесь советов, только при этом необходимо быть предельно внимательным и хорошенько продумать последствия.

Сохранение состояний

Если работа метода объекта была прервана выбросом исключения, то общее состояние объекта не должно измениться. В обработчике исключения, расположенном внутри метода, обязательно необходимо восстановить состояние объекта, которое наблюдалось до вызова метода. Если этого не будет сделано, то повторный вызов метода может привести к непредсказуемым результатам. Если данные объекта будут лишь частично модифицированы методом, прерванным исключением, то ждать корректного поведения от объекта не стоит.

Проверяйте ссылки

Если при использовании какой-либо ссылки существует вероятность того, что она не проинициализирована, то лучше предварительно проверить ее доступность при помощи оператора `if`. Это позволит избежать выброса исключения `NullReferenceException`.

```
if (SomeObj != null)
        SomeObj.Close();
```

Типы исключений

При использовании собственных типов исключений, которые будут использоваться внешними приложениями, обратите внимание на доступность типов. Типы, которые вы используете для описания своих исключений, обязательно должны быть доступны внешним приложениям. Иначе их обработка будет невозможна.

Три конструктора

При создании собственных типов исключений на основе `System.Exception` не забудьте переопределить три общедоступных конструктора `System.Exception`.

Локализация сообщений об ошибках

Не ленитесь создавать локализованные версии сообщений об ошибках, которые будут читать пользователи. Пишите сообщения об ошибках как можно более грамотно и понятно. Исходите из того, что сообщение будет читать пользователь, мало знакомый с проблемой.

Файлы помощи

С каждым исключением можно сопоставить ссылку, указывающую на файл помощи. Пользователям будет очень приятно, если при возникновении

некоторой ошибки они смогут прочитать подробную документацию, а не одну скупую строку пояснения.

Не используйте кодов возврата

Настоятельно рекомендую, не используйте кодов возврата, это устаревшая технология. В .NET все ошибки следует обрабатывать при помощи исключений, которые предоставляют полную информацию без необходимости обращения к дополнительным сервисам.

Заканчивайте ... на *Exception*

Имена собственных классов исключений старайтесь заканчивать словом `Exception`. Хотя данное требование не обязательно, оно позволяет сразу же отличить класс исключения среди множества других. Это очень удобно.

7.9. Заключение

Многое в данной главе, наверное, было сложным и непонятным. Если нет, то я склоняю перед вами голову. Сам я, конечно же, понял не все из того, что написал ☺. Но искренне надеюсь, что вы сможете в этом разобраться.

Глава 8

Управление памятью

Проблема управления памятью существует с момента написания первых программ. Издревле программисты бьются в поисках компромисса между скоростью и надежностью работы с памятью. К сожалению, решение этой задачи является "палкой о двух концах". И перед программистами встает буриданова[1] проблема: скорость или надежность.

В этой главе пойдет речь о технологии сборки мусора, при помощи которой Microsoft попыталась решить проблемы управления памятью.

8.1. Проблемы и задачи

В рамках управления памятью можно выделить три основные проблемы: неверные указатели, утечки памяти, а также неэффективное распределение участков памяти. Рассмотрим каждую из этих проблем более подробно.

Неверные указатели

Прежде чем рассказывать о неверных указателях, необходимо обратиться к сути самих указателей и ответить на вопрос, а что вообще такое указатель? Указатель — это адрес некоторой области памяти, где адрес представляет собой обыкновенное число, указывающее местоположение данных в памяти. Причем разрядность числа напрямую зависит от разрядности системы. На 32-разрядных системах используются 32-разрядные указатели, а на 64-битных — уже 64-битные указатели. Таким образом, местоположение в памяти задается 32/64-битным линейным числом.

---

[1] Некий парадокс об осле, не решающемся выбрать ни одну из двух одинаковых и равноудаленных от него охапок сена и потому обреченному на голодную смерть. Авторство парадокса приписывается французскому философу Ж. Буридану — 14 век.

Примечание

На самом деле линейная организация памяти в современных системах является не более чем иллюзией, за которой скрываются сложные механизмы преобразования линейного адреса в физический (настоящий).

В обычных неуправляемых средах программист волен распоряжаться с данным числом (указателем) по своему усмотрению. Он может складывать и вычитать такие числа, заносить в них произвольные значения, ну и конечно разыменовывать указатели любыми доступными типами.

Операцию разыменования указателей хотелось бы рассмотреть более подробно, поскольку она далеко не очевидна. Данная операция подразумевает возможность рассмотрения области памяти, адресуемой указателем, как объект заданного типа. Таким образом, мы можем представить любую область памяти как объект любого известного нам типа. К примеру, можно использовать один и тот же участок памяти и как структуру, и как простой целочисленный тип или вообще как таблицу интерфейсов. Поскольку приведение типов указателей никоим образом не контролируется со стороны среды операционной системы и процессора, то никаких принципиальных ограничений в данной не наблюдается. В этом состоит и сила и слабость неуправляемых приложений. С одной стороны, мы получаем необычайную гибкость при использовании памяти, но с другой — полностью теряем безопасность и надежность. Любой неверный шаг может привести к неотвратимым последствиям. Работу с указателями можно сравнить с ходьбой по минному полю — никогда не знаешь, где рванет.

При использовании классических указателей можно усмотреть две проблемы: попытка доступа к закрытым областям и использование неверных данных. Первая проблема возникает, когда указатель ссылается на одну из закрытых областей (страниц) памяти. При попытке доступа к подобной области операционная система выбросит исключение, и если приложение его не перехватит, то оно будет закрыто. Данная ситуация наверняка знакома всем читателям: когда на экране появляется диалоговое окно с сообщением: "Программа выполнила недопустимую операцию и будет закрыта". В качестве примера закрытой области можно указать нулевую страницу памяти — атрибуты которой в отладочных целях полностью запрещают какой бы то ни было доступ к ней. Если попытаться прочитать или записать что-либо в нулевую (а также немного выше) область памяти, то будет выброшено исключение.

Как бы странно не звучало, но проблемы подобного рода являются наиболее благоприятными. Да, конечно, будет выброшено исключение, и приложение на этом, скорее всего, закончит свою работу, но зато у нас будет возможность точно локализовать произошедшую проблему. Благо, исключение предоставит нам всю необходимую для этого информацию. Гораздо хуже, если указатель будет разыменован неверным типом или в неверной области

памяти. Первый случай связан с проблемой, когда мы попытаемся получить доступ к объекту одного типа при помощи другого — к примеру, к типу int, при помощи float (в классических неуправляемых языках это вполне возможно). Во втором случае, хотя и используется верный тип, но указатель адресует неверную область памяти, то есть область, в которой и не пахло необходимым объектом заданного типа. Самое страшное состоит в том, что в обоих этих случаях удастся получить некие данные, которые будут рассматриваться программой как верные, но, к сожалению, таковыми являться, конечно же, не будут. Проблемы данного типа относят к наиболее опасным, потому как локализовать их стандартными средствами не представляется возможным — необходима тщательная и кропотливая отладка при непосредственном участии самого программиста. При возникновении подобных ошибок программа, скорее всего, будет продолжать работать, только вот работать она будет неправильно и предсказать ее поведение будет крайне сложно.

Автоматическое, а тем более прозрачное решение проблем, напрямую связанных с указателями, в классических неуправляемых средах является невозможным. Для того чтобы решить эту проблему, необходимо контролировать все указатели, а это попросту невозможно, поскольку противоречит самой идеологии указателей, которая представляет их как обычные числа.

В языке C++ для предотвращения случайной порчи указателей были введены более строгие правила преобразования типов, которые начисто исключают случайное приведение указателей к неверному типу. Но эта искусственная мера скорее похожа на крик отчаяния, чем на реальный подход к решению проблемы.

Для того чтобы решить проблему неверных указателей полностью, разработчикам .NET пришлось внести коррективы в саму идеологию программирования, отказавшись от указателей и заменив их ссылками. Это было единственное возможное решение, которое позволило бы контролировать обращения к памяти абсолютно прозрачно для самих программистов. На первый взгляд, отказ от указателей выглядит весьма удручающее — но, разобравшись в механизме работы ссылок, понимаешь, что преимущества, предоставляемые ими, вполне оправдывают вносимые ими неудобства.

Ссылки

Ссылки являются специальными объектами среды исполнения .NET, инкапсулирующими реальные указатели на объекты. С указателями, расположенными внутри ссылок, можно работать только косвенно, при помощи команд среды исполнения.

Примечание

В принципе, со ссылками можно работать как с классическими указателями. Но в этом случае, код будет помечен как небезопасный и в некоторых случаях его исполнение может быть запрещено политикой безопасности.

Естественно, что при этом среда исполнения контролирует все операции, проводимые над указателями. При таком тотальном контроле ошибку при использовании ссылок совершить попросту невозможно. При этом необходимо обязательно отметить, что над ссылками запрещено проводить классические арифметические операции. Ссылки можно лишь копировать (операция =), а также получать с их помощью доступ к объекту. Таким образом, неудивительно, что с введением столь строгих ограничений, разработчикам .NET удалось добиться практически полной отказоустойчивости и безопасности при работе со ссылками.

Утечки памяти

В классических неуправляемых средах предполагается ручное управление памятью. Программист обязан самостоятельно следить за выделением и освобождением памяти. Если был запрошен некоторый участок памяти, то он обязательно должен быть освобожден, иначе данный участок повиснет в виде неиспользуемого груза.

Рассмотрим механизм получения памяти в неуправляемой среде. Программист запрашивает операционную систему выделить ему область памяти некоторого размера. Это можно сделать при помощи следующих функций API.

```
VirtualAlloc
LocalAlloc
HeapAlloc
```

Все они возвращают указатель на выделенную область памяти, то есть ее адрес. Программист сохраняет данный адрес в одной из переменных, при помощи которой получает доступ к выделенному участку. При этом для того чтобы передвигаться по данному участку памяти, он может применять адресную арифметику, используя значение данного указателя. К примеру, можно прибавить к указателю число два, а затем сосчитать или записать значение второго байта в выделенной области. Это очень удобно и потрясающее эффективно и гибко, но использование адресной арифметики сопряжено с опасностью неверного доступа к памяти и к тому же требует немалого опыта работы в данной области.

После того как программист закончит использовать выделенную область памяти, он должен вернуть ее операционной системе. Но зачастую программисты не делают этого, потому как механизм использования динамической памяти не является очевидным. Для чего необходимо выделять память, понятно всем программистам. Не выделишь память под объект — не получится его использовать. А вот для чего освобождать объект, совершенно непонятно, поскольку и без этого все как бы отлично работает.

Выделяя область памяти, программисты чаще всего сохраняют указатель на него в локальной переменной. Затем, забыв освободить выделенную ранее

память либо вовсе забывают о данной переменной, либо инициализируют ее другим значением. Так и происходят утечки памяти. Продемонстрируем на примере (листинг 8.1).

Листинг 8.1. Демонстрация утечек памяти

```
/*
  Листинг 8.1
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим стандартный заголовочный файл.
#include <windows.h>
// В этой функции произойдет утечка памяти.
void allocMem()
{
  void *pSomeMem;
  // Выделим память и поместим полученный указатель
  // в локальную переменную.
  pSomeMem = (void*)malloc(1024);
  // Здесь надо было освободить выделенную память,
  // но поскольку указатель на нее сохранен в локальной переменной,
  // сделать этого не удастся, так как его значение будет потеряно.
}
// Точка входа в приложение.
void main()
{
  allocMem();
  // Здесь тоже произойдет утечка памяти.
  void *pSomeMem2;
  // Первый раз выделим память и получим указатель на выделенную
  // область памяти.
  pSomeMem2 = (void*)malloc(1024);
  // Повторно запросим выделить участок памяти и при этом
  // "затрем" старое значение указателя,
  // что сделает невозможным освобождение первого
  // участка памяти.
  pSomeMem2 = (void*)malloc(1024);
}
```

В ходе работы программы произойдет две утечки памяти. В функции `allocMem` выделяется один Кбайт динамической памяти, указатель на которую помещается в локальную переменную `pSomeMem`. Но поскольку данная переменная располагается в стеке вызова, то после выхода из функции ее значение будет потеряно. А следовательно, освободить данную область памяти будет принципиально невозможно, в отсутствии информации о ее местоположении. Вторая утечка произойдет в функции `main`, — здесь последовательно выделяется два участка памяти, причем указатели на них сохраняются в одну и ту же переменную `pSomeMem2`. И второе значение, естественно, затирает первое. Таким образом, у нас образуется два недостижимых выделенных участка памяти, указателей на которых в программе нет. Они как бы повисают между операционной системой и программой. Операционная система считает, что они используются программой, а программа о них уже и думать забыла.

Видно, что такой важный момент далеко не очевиден и может ускользнуть из внимания даже опытного программиста.

Казалось бы, проблема не так уж и страшна, ведь памяти много, — потеряем указатели на пару областей, ну и пусть себе висят, все равно, когда программа завершит работу, эти области будут возвращены операционной системе. Однако ситуация гораздо серьезнее, чем кажется. Если память будет утекать в циклических многократно исполняющихся алгоритмах, то даже при малых утечках через некоторое время производительность приложения существенно снизится. Всем, наверное, знакомо неадекватное поведение многих программ после нескольких часов непрерывной работы с ними. Раскрою секрет, чаще всего это происходит из-за утечек памяти. Диву даешься, когда небольшая прикладная программа после нескольких часов работы занимает в памяти несколько сот Мбайт — а такое далеко не редкость.

Тут, естественно, встает вопрос, а почему операционная система не отслеживает указатели, используемые в программе, и при потере последнего указателя на некоторую область памяти не освобождает ее. Оказывается, это принципиально невозможно. Операционная система не может отслеживать указатели, поскольку из-за возможности вольного обращения с оными невозможно определить, что является указателем, а что обычными переменными. При использовании адресной арифметики указатели зачастую адресуют не начальные области памяти, а могут смещаться в произвольные участки — в этом заключается их основная сила. В неуправляемой среде полностью решить проблему с утечками памяти не удается. Правда, существует гибридное решение с использованием специальных объектов — интеллектуальных указателей. Они представляют собой классы, инкапсулирующие реальные указатели. Основная идея заключается в том, что память, на которую адресуют данные указатели, автоматически освобождается из деструктора, который в свою очередь вызывается языковой средой при выходе объекта из области видимости. Но проектирование и использование данных объектов, как

ни странно, является далеко не тривиальной задачей. Поскольку обязательно необходимо отслеживать копирование указателей, которое зачастую может происходить не явно. Соответственно при возникновении подобных ситуаций необходимо вручную указать объекту интеллектуальной ссылке на то, что произошло дублирование указателя. Естественно, что это неудобно и большинство программистов предпочитают не использовать подобные технологии. Поскольку все равно толком не понимают, как ими пользоваться. Интеллектуальные указатели нашли распространение только при использовании COM-интерфейсов, там удалось создать достаточно гибкую систему, которая позволяла контролировать время жизни объектов без участия программиста.

Но с отказом от указателей и переходом на ссылки, ситуация кардинально меняется, теперь исполняющая среда может отследить все используемые указатели, а соответственно и организовать автоматическое управление памятью. В .NET для этого применяется технология сборки мусора (GC, garbage collection). Ее суть состоит в том, что теперь программисты отвечают только за выделение памяти, а за освобождение неиспользуемых областей отвечает сама исполняющая среда. Когда в программе теряется последняя ссылка, указывающая на объект, среда исполнения помечает его как мусор и во время очередной операции по сборке мусора данная область автоматически освобождается. Мало того, что данная модель гораздо нагляднее, чем классическая, так она еще и полностью исключает возникновение утечек памяти.

Далее будет подробно описана технология сборки мусора, а также сопутствующие ему механизмы.

Эффективное распределение участков памяти

Мечта любого уважающего себя программиста предполагает быструю и высокопроизводительную работу написанных им приложений. А производительность, равно как и скорость, напрямую зависит от эффективности использования памяти.

Здесь пойдет речь о внутреннем распределении участков памяти, механизм которой в большинстве случаев сокрыт от обычных прикладных программистов. Во время работы программы постоянно происходят выделения и освобождения участков памяти. Естественно, что после некоторого времени работы распределение занятых и свободных участков будет неравномерно. То есть произойдет фрагментация памяти.

Примечание

Фрагментация — происходит от английского слова fragmentation, что в переводе означает: дробление, разбиение, разделение, расщепление. Дефрагментация — прямо противоположное понятие, означающее: уплотнение, устранение фрагментации.

Для большей ясности продемонстрируем на примере. Допустим, мы выделяем 100 байт динамической памяти. Затем через некоторое время решаем, что все четные байты нам не нужны и освобождаем их (рис. 8.1). Также сделаем допущение, что нашей программе предоставлено в распоряжение только эти сто байт и ни байтом больше. Фактически у нас есть пятьдесят свободных байт памяти, а на деле оказывается, что выделить их не удастся. Попытка выделения любого количества байт больше одного закончится неудачей.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68
69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85

Рис. 8.1. Фрагментация памяти

В стандартных классических системах (Windows, ANSI C), не поддерживающих дефрагментацию памяти, при запросе на выделение некоторого количества памяти, менеджер просматривает с самого начала все свободные участки для того, чтобы подыскать область подходящего размера.

Примечание

В защиту менеджера памяти Windows все же хотелось бы сказать несколько слов. Хотя возможность дефрагментации памяти в его среде принципиально невозможна, все же менеджер позволяет произвести частичную дефрагментацию памяти, объединив соседние неиспользуемые области. Это позволяет несколько повысить скорость запроса на выделение памяти. Эта операция осуществляется при помощи API-функции `HeapCompact`.

Но как мы видели, в силу фрагментации памяти необходимый участок может быть недоступен, даже если суммарное количество свободной памяти явно больше, чем объем запрашиваемой области.

По идее, было бы хорошо иметь возможность дефрагментации всей памяти программы. В ходе которой все выделенные участи собирались бы в одно место, а свободные в другое (рис. 8.2).

Выгода от дефрагментации ясна даже без дополнительных объяснений, при взгляде на схему сразу же бросается в глаза большая серая область, которая представляет собой участок свободной памяти. А из нее уже можно выделять объекты, размер которых сопоставим с реальным объемом свободной памяти.

К сожалению, в классических неуправляемых средах дефрагментация памяти принципиально невозможна, по причине отсутствия обратной связи с указа-

телями. Ведь если мы сдвинем некоторые области памяти, соответственно необходимо обновить адреса указателей, ссылающихся на них. Если этого не сделать, то указатели будут адресовать совершенно другие области памяти, что, безусловно, приведет к непоправимым последствиям.

1	3	5	7	9	11	13	15	17	18	20	22	24	26	28	30	32
34	35	37	39	41	43	45	47	49	51	52	54	56	58	60	62	64
66	68	69	71	73	75	77	79	81	83	85	2	4	6	8	10	12
14	16	19	21	23	25	27	29	31	33	36	38	40	42	44	46	48
50	53	55	57	59	61	63	65	67	70	72	74	76	78	80	82	84

Рис. 8.2. Дефрагментация памяти

В .NET ситуация совершенная иная. Поскольку исполняющая среда знает обо всех ссылках, используемых в программах, соответственно появляется возможность обновить их, а как следствие становится возможной дефрагментация памяти. С радостью сообщу вам, что дефрагментация является неотъемлемой частью процесса сборки мусора. Более подробно мы обсудим этот момент далее.

Теперь, после того как обозначены основные проблемы и задачи технологий управления памятью, приступим к непосредственному рассмотрению механизма управления памятью в среде .NET.

8.2. Управление памятью в среде .NET

Управление памятью в среде .NET полностью автоматизировано и является прозрачным для рядовых прикладных программистов процессом. Тем не менее, рекомендуется ознакомиться с механизмами работы менеджера памяти среды .NET, дабы создавать эффективные и высокопроизводительные приложения.

Модель памяти, используемая в .NET, является наиболее наглядной из известных мне. Здесь программист отвечает только за создание объектов, а освобождаются они автоматически средой исполнения. Явного механизма детерминированного уничтожения объектов, наподобие оператора `delete` языка C++, не существует.

Механизм освобождения объекта следующий. При потере последней ссылки, указывающей на выделенный ранее объект, он помечается как мусорный. При этом удаления объекта не происходит, он лишь помечается как мусорный — система функционирует как ни в чем не бывало. Но вот при попытке выделения памяти под один из новых объектов, среда исполнения .NET

решает, что общий объем выделенной памяти достиг критического порога, и инициирует процесс сборки мусора. Который сводится к банальному уплотнению (дефрагментации) памяти и обновлению ссылок на объекты (рис. 8.3).

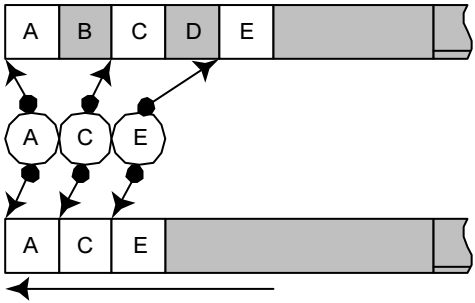


Рис. 8.3. Обновление ссылок на объекты во время сборки мусора

Серым помечены объекты, которые считаются мусором, а также девственно свободная нетронутая область памяти, которая будет использоваться для выделения новых объектов. Как видно из схемы, сборка мусора, прежде всего, представляет собой дефрагментацию памяти. Все используемые объекты сдвигаются в самое начало, при этом замечательно то, что для удаления "мусорных" объектов не требуется предпринимать дополнительных действий. Помимо обращения к их "деструкторам", если, конечно, это требуется.

Так же естественно, что после сборки мусора обновляются значения ссылок на объекты. Происходит данный процесс автоматически, без какого бы то ни было участия со стороны программиста.

Устройство механизма выделения памяти

Можно смело заявить, что данный механизм выделения памяти является наиболее быстрым из существующих. По скорости он даже превосходит работу сервисов операционной системы. Что кажется нереальным, но это действительно так. Все-таки оказывается, что в отдельных случаях можно прыгнуть выше головы.

Секрет заключается в следующем, — перед началом работы приложения менеджер памяти .NET выделяет для программы большую непрерывную область, которая естественно на момент запуска приложения будет пуста. Также он инициализирует специальный внутренний указатель начала этой свободной области, — изначально он равен нулю (рис. 8.4). Назовем данный указатель `pFreeBegin`, потому как он всегда указывает на начало свободной области памяти.

Когда программа запросит создание объекта, менеджер памяти просто вернет ей ссылку, указывающую на данную область памяти (`pFreeBegin`), после чего сдвинет `pFreeBegin` вперед на размер выделенного объекта (рис. 8.5).

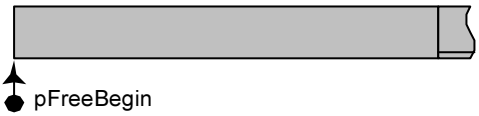


Рис. 8.4. Указатель на свободную область памяти

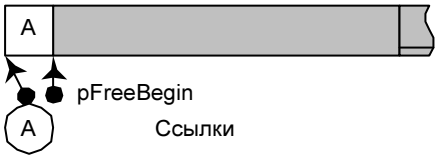


Рис. 8.5. Механизм выделения памяти для нового объекта

Обратите внимание, что программист не может запросить выделение не типизированного участка памяти произвольного размера. Он может лишь создать объект определенного типа, а среда исполнения уже сама будет определять, сколько памяти под него требуется, а также заниматься ее резервированием. Таким образом, выделение памяти представляет собой линейный процесс, и сводится к банальному складыванию адресов и возврату соответствующих ссылок.

Создание объектов в управляемой среде происходит при участии инструкции `newobj`, которая принимает тип объекта, прототип конструктора, а также набор необходимых параметров, которые необходимо передать выбранному конструктору. Соответственно мы имеем возможность автоматической инициализации вновь создаваемых объектов, для этого всего лишь необходимо определить конструктор типа. Приведем пример.

```
class Some
{
  // Конструктор типа
  public Some()
  {
    // Инициализируем поле объекта начальным значением
    x = "Hello, World!";
  }
  public String x;
}
```

При запросе на создание объекта данного типа, среда исполнения сначала выделит память под объект, затем вызовет конструктор, неявно передав ему ссылку на выделенную область памяти, а только затем вернет ссылку пользователю.

Особо хотелось бы отметить, что устройство конструкторов принципиального исключает возможность возвращения результата работы этого метода классическим способом. При создании объектов в большинстве языков используется следующая конструкция.

```
somereference = new Some();
```

При этом в переменную `somereference` помещается ссылка на только что созданный объект типа `Some`().

Если же при использовании конструктора все-таки понадобилась обратная связь, то ее можно наладить при помощи параметров, передаваемых по ссылке. А в конструкторе изменять их значение, затем после выполнения оного считывать новые значения параметров. Приведем пример, в котором демонстрируется возвращение строки, через специальный параметр конструктора (листинг 8.2).

Листинг 8.2. Создание обратной связи при работе с конструктором

```
/*
  Листинг 8.2
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Опишем специальный тестовый класс, над конструктором которого будем
// экспериментировать.
class Some
{
  // Конструктор класса.
  public Some(ref string s)
  {
    s = "Hello, World!";
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
```

```
    // Опишем переменную ссылку, через которую будем получать
    // необходимую информацию.
    String str = null;
    new Some(ref str);
    // Выведем на консоль значение переменной str.
    Console.WriteLine(str);
  }
}
```

В результате работы приложения на консоль будет выведена следующая строка.

```
Hello, World!
```

Правда, полезность такой возможности весьма сомнительна, поскольку результат работы конструктора можно передавать через общедоступные члены объекта, доступ к которым станет возможным сразу же после создания экземпляра данного класса.

Особо хотелось бы отметить возможность предотвращения создания объекта из конструктора искомого типа. Суть состоит в том, что если в конструкторе объекта выбросить исключение, то объект не будет создан. По крайней мере, так будет казаться прикладному программисту верхнего уровня. В действительности же дело обстоит несколько иначе. На самом деле объект будет создан, просто при его создании конструктор выбросит исключение, и операция установки ссылки на объект будет попросту пропущена в поисках подходящего обработчика исключения. Но прошу обратить ваше внимание на то, что объект все-таки будет создан, правда его дееспособность, возможно, будет зависеть от степени завершенности работы конструктора. Если такая ситуация возникнет, то подобный объект будет автоматически помечен как мусорный. Произойдет это потому, что на него не будет указывать ни одна ссылка в программе.

Для того, чтобы доказать, что объект все-таки создается, приведем пример, который демонстрирует это довольно не очевидным способом. В примере будет рассмотрен класс с хитрым конструктором, который сохраняет ссылку на только что созданный объект во временной переменной, при помощи которой можно будет впоследствии получить доступ к объекту (листинг 8.3).

Листинг 8.3. Предотвращение создания объекта из его конструктора

```
/*
  Листинг 8.3
  File:   Some.cs
  Author: Дубовцев Алексей
```

```
*/
// Подключим основное пространство имен общей библиотеки классов
using System;
// Введем дополнительный класс с конструктором,
// демонстрирующим создание объекта, даже несмотря на исключение,
// выброшенное в самом конструкторе, непосредственно во время
// создания объекта.
class Some
{
  // А это и есть особо хитрый конструктор класса.
  // Он принимает в качестве параметров ссылку на объект
  // типа App, там он будет сохранять ссылку на только, что
  // созданный объект, данного типа, а во второй
  // переменной будет передаваться контрольная строка,
  // позволяющая убедиться в создании именно этого объекта.
  public Some(App a, string str)
  {
    // Сохраним ссылку на только что созданный объект
    // в специальной временной переменной.
    a.someTemp = this;
    m_String = str;
    // Для большей надежности выведем на консоль хеш-код вновь созданного
    // объекта.
    Console.WriteLine("Hash of this object is = {0}",this.GetHashCode());
    // Выбросим исключение, попытаясь предотвратить создание объекта.
    throw new Exception("Can't create Some object");
  }
  // Специальный закрытый член класса, который
  // инициализируется контрольной строкой, переданной конструктору
  // класса.
  private string m_String;
  // Метод, выводящий на консоль значение поля m_String.
  // Необходим для того, чтобы удостовериться в целостности объекта.
  public void SayHello()
  {
    Console.WriteLine("Some::SayHello Mehto invoke m_String = {0}",
  m_String);
  }
}
```

```
// Основной класс приложения.
class App
{
  // Временная дополнительная ссылка на созданный объект, которая
  // будет обновляться самим объектом из его конструктора.
  public Some someTemp;
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим экземпляр основного класса приложения, он необходим
    // для хранения ссылки на объект типа Some.
    App appTemp = new App();
    // Проинициализируем значение ссылки.
    Some refToSome = null;
    // Введем защищенный блок, который позволит перехватить
    // исключение, возникающее в конструкторе.
    try
    {
      // Попытаемся создать экземпляр типа Some.
      refToSome = new Some(appTemp,"Hello, World!");
    }
    // Опишем перехватчик исключения.
    catch(Exception ex)
    {
      // Выведем информацию по произошедшему исключению.
      Console.WriteLine("Exception Detected {0}",ex.ToString());
    }
    // Проверим, удалось ли создание объекта,
    // и в случае неудачи выведем сообщение об этом на консоль.
    if (refToSome == null)
      Console.WriteLine("Can't use refToSome it's null");
    // Обратимся к методу объекта, создание которого как бы не удалось.
    appTemp.someTemp.SayHello();
    // Выведем на консоль хеш как бы не созданного объекта
    Console.WriteLine("Hash code of someTemp object is = {0}",
appTemp.someTemp.GetHashCode());
  }
}
```

В результате работы программы на консоль будут выведены следующие строки.

```
Hash of this object is = 1
Exception Detected System.Exception: Can't create Some object
   at Some..ctor(App a, String str)
   at App.Main()
Can't use refToSome it's null
Some::SayHello Mehto invoke m_String = Hello World
Hash code of someTemp object is = 1
```

Лог показывает, что создание объекта произошло. Во-первых, это можно определить по приветствию "Hello, World!", которое выводит на консоль метод `SayHello`. Во-вторых, хеш-коды объектов совпадают, что говорит об их идентичности.

Примечание

Об уникальности хеш-кодов в данном случае можно говорить без зазрения совести, поскольку по умолчанию они равны порядковому номеру объекта в управляемой куче. А порядковый номер может измениться только после сборки мусора, которая в нашем приложении произойдет только при его завершении.

Пример на самом деле является демонстрацией модифицированной технологии воскрешения, о которой мы поговорим более подробно несколько позднее.

Правда о деструкторах или когда уходят объекты

Скорее всего, я удивлю вас, заявив, что среда исполнения не поддерживает деструкторы как таковые. Методов по свойствам, схожим деструкторам, в среде исполнения нет!! "Но как же так, ведь синтаксис C# поддерживает определение деструкторов", — могут сказать наиболее просвещенные читатели. Мы без проблем можем объявить следующий тип, поддерживающий деструктор.

```
class Some
{
  ~Some()
  {
    // Здесь по идее должна проводиться деинициализация объекта.
  }
}
```

Человеку непосвященному может показаться, что мы действительно объявили деструктор, но на самом деле это не деструктор, а финализатор. Деструк-

тор является лишь удобным синтаксисом определения финализатора — при компиляции он будет заменен методом `Finalize()`. Финализатор вызывается системой перед непосредственным уничтожением объекта, то есть во время процесса сборки мусора. В этом его принципиальное отличие от деструктора, который вызывается при выходе объекта из области видимости. Приведем пример с тем же классом `Some`.

```
class Some
{
  ~Some()
  {
    Console.WriteLine("Hello, World!");
  }
}
class App
{
  void Function()
  {
    Some s = new Some();
    ...
  }
...
```

Логично было бы предположить, что после завершения исполнения функции `Function` класса `App`, будет вызван деструктор `~Some()`, потому что при выходе из функции теряется последняя ссылка на вновь созданный объект `Some()` и он помечается как мусорный. Но в том-то и дело, что объект лишь помечается как мусорный, но его удаление не происходит. А финализатор будет вызван лишь в процессе сборки мусора, который может быть инициирован в любое заранее непредсказуемое время.

Очередь финализации

Также с финализаторами связан еще один неприятный момент, объяснить который можно, лишь обратившись к внутреннему механизму их работы. При создании любого объекта среда исполнения проверяет, реализован ли в нем метод `Finalize`, если это так, то объект автоматически добавляется в очередь финализации. Исключение делается только для объектов типа `Object`, хотя они и содержат метод `Finalize`, но он необходим лишь для вызова из классов потомков.

Во время сборки мусора среда исполнения начинает последовательно вызывать методы из очереди финализации. Самое интересное заключается в том, что среда исполнения не гарантирует последовательность вызова финализаторов.

Примечание

В простых однопоточных приложениях финализаторы все-таки вызываются, как положено в обратном порядке создания объектов, к которым они прикреплены. Правда, гарантировать такое же поведение для более сложных приложений я не возьмусь.

Что, конечно же, недопустимо для классических деструкторов, поскольку это может привести к коллизиям доступа, так как будет невозможно предсказать, какие из объектов живы на момент вызова деструктора, а какие нет. Как бы странно это не звучало, но именно так и обстоит дело с финализаторами. Дабы избежать проблем при их использовании, Microsoft рекомендует не обращаться при их вызове к ссылкам на другие объекты, что само по себе выглядит несколько абсурдно. Спрашивается, для чего вообще тогда были введены финализаторы.

Вызов финализаторов

С весьма странным порядком вызова финализаторов мы уже разобрались, но это еще полбеды, самое неприятное впереди. Оказывается, среда исполнения вызывает финализаторы в отдельном высокоприоритетном потоке, который исполняется параллельно основным потокам приложения. Соответственно, необходимо ждать неприятностей, связанных с многопоточными блокировками и совместном доступе к объектам.

Продемонстрируем многопоточность модели освобождения ресурсов примером, который в деструкторе тестового объекта будет выводить информации о потоке, в котором тот исполняется (листинг 8.4).

Листинг 8.4. Демонстрация многопоточности механизма освобождения ресурсов

```
/*
  Листинг 8.4
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу с потоками.
using System.Threading;
// Введем дополнительный класс, над деструктором
// которого будем экспериментировать.
class TestC
{
  // Введем деструктор, который выведет информацию о потоке, в котором
```

```
  // будет исполняться.
  ~TestC()
  {
    // Для начала сообщим пользователю о том, что финализатор
    // вообще был вызван.
    Console.WriteLine("Hello, World: from TestC.Finalize\n");
    // Выведем на консоль хеш-код объекта, представляющего текущий
    // поток.
    Console.WriteLine("Destructor thread's hash code -  {0}",
      Thread.CurrentThread.GetHashCode());
    // Выведем на консоль приоритет текущего потока.
    Console.WriteLine("Thread Priority - {0}",
Thread.CurrentThread.Priority.ToString());
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  static void Main()
  {
    // Выведем на консоль хеш-код объекта, представляющего текущий
    // поток.
    Console.WriteLine("Main thread's hash code - {0}\n",
Thread.CurrentThread.GetHashCode());
    // Создадим объект, ссылка на который будет потеряна
    // еще до завершения работы данной функции.
    new TestC();
  }
};
```

В результате работы приложения на консоль были выведены следующие строки.

```
Main thread's hash code — 1
Hello, World: from TestC.Finalize
  Destructor thread's hash code -  2
  Thread Priority — Highest
```

Лог отчетливо показывает, что исполнение финализатора происходило в отдельном потоке, а также то, что приоритет данного потока был повышен.

Подведем итог

Таким образом, финализаторы никоим образом не подходят для детерминированного уничтожения объектов. То есть для такого уничтожения, когда вам точно необходимо знать, когда и в какой последовательности вызовутся деструкторы созданных вами объектов. Здесь же вам гарантируется лишь то, что финализатор действительно будет вызван.

Если вы все же решили воспользоваться финализаторами, то вам небезынтересно будет узнать об одной особенности их проектирования. При объявлении финализатора компилятор автоматически вставляет обращение к финализатору базового класса. Это необходимо для правильной очистки сложных унаследованных объектов. Для того чтобы гарантировать вызов базового финализатора, компилятор оборачивает код в защищенный участок, в блоке `finally` которого обращается к базовому финализатору. Выглядит это примерно так:

```
Finalize()
{
  try
  {
    // Код финализатора.
  }
  finally
  {
        base.Finalize();
}
```

Это правильный подход, поскольку гарантирует стопроцентную очистку объектов с наследованием. Но некоторые компиляторы не делают этого, к примеру MC++ не только не оборачивает код финализатора в защищенный блок, но он даже не вызывает базовый финализатор. Для создания правильно спроектированных объектов вам придется сделать это самостоятельно.

```
__gc class Some
{
public:
  ~Some()
  {
    __try
    {
      // Код финализатора.
    }
    __finally
```

```
    {
      Object::Finalize();
    }
  }
};
```

Теперь очищение объекта произойдет правильно.

Также хотелось бы предупредить об использовании в финализаторах операций, которые явно или косвенно могут блокировать поток, в котором они выполняются. Если поток будет все-таки блокирован, то среда исполнения подождет около 2 секунд, после чего перестанет вызывать финализаторы и завершит работу приложения.

Настоящие деструкторы — детерминированное уничтожение объектов

Недостатки финализаторов, замаскированных в высокоуровневых языках под деструкторы, становятся очевидны сразу же после того, как понимаешь механизм их работы. Они совершенно не оправдывают себя, когда требуется детерминированное уничтожение объектов. То есть, когда требуется освободить объекты в определенном порядке в заданное время, использование финализаторов абсолютно неприемлемо.

Именно поэтому Microsoft пришлось ввести дополнительную модель освобождения объектов. Она базируется на интерфейсе `IDisposable`, который содержит один-единственный метод `Dispose`.

```
void Dispose();
```

Объекты, поддерживающие эту модель уничтожения, обязаны реализовывать этот интерфейс, где метод `Dispose` по сути дела является деструктором объекта. Приведем пример объекта, поддерживающего детерминированное уничтожение.

```
class Some : IDisposable
{
  void IDisposable.Dispose()
  {
    Console.WriteLine("Hello, World - Disposing object");
    // Здесь мы должны произвести очистку
    // необходимых ресурсов.
  }
};
```

Основная идея состоит в том, что теперь данный объект поддерживает явное уничтожение, хотя и в ручном режиме. Когда программист решит, что пришло

время уничтожить объект, он обратится к методу `Dispose`, который выполнит необходимые операции. При этом прошу обратить внимание на то, что сам объект физически уничтожаться не будет, произойдет лишь его логическое уничтожение, после которого его использование будет бессмысленным.

Метод `Dispose` активно используется в классах, являющимися обертками для объектов операционной системы. К примеру, можно указать класс `FileStream` — очевидно, что данный объект работает с файловыми объектами операционной системы. При вызове метода `Dispose` данный объект попросту закрывает ассоциированный с ним файловый объект при помощи API-вызова `CloseHandle`. После него использование естественно будет бессмысленным — файл-то все равно закрыт. Хотя сам объект `FileStream` никуда не денется и останется вполне целым, с формальной точки зрения.

Шаблон реализации *IDisposable*

Ниже будет представлен шаблон (пример) реализации поддержки интерфейса `IDisposable`, для объекта, поддерживающего детерминированное уничтожение. К подобным классам предъявляются серьезные требования.

- При вызове любых операций они обязаны отслеживать, находится ли объект в рабочем состоянии или уже был деинициализирован при помощи метода `Dispose`. Согласитесь, было бы довольно глупо проводить какие бы то ни было операции над объектом, если фактически он является недееспособным.
- Необходимо предусмотреть автоматическую деинициализацию объекта из финализатора, на тот случай, если программист забудет обратиться к методу Dispose или в силу некоторых причин сочтет это ненужным. Также необходимо учесть то, что при вызове финализатора объект уже может быть деинициализирован.
- При определении метода `Dispose` рекомендуется проводить многопоточную синхронизацию, дабы избежать возможных проблем связанных с блокировкой потоков.
- Для интуитивного удобства деинициализации объекта, рекомендуется определять дополнительный метод `Close`, который перенаправляет вызов к функции `Dispose`.

Все эти требования реализованы в объекте `Some` (листинг 8.5).

Листинг 8.5. Шаблон использования модели детерминированного уничтожения, основанной на интерфейсе `IDisposable`

```
/*
  Листинг 8.5
  File:   Some.cs
```

```
  Author: Дубовцев Алексей
*/
// Объект, поддерживающий модель детерминированного уничтожения.
public class Some: IDisposable
{
  // Некоторый внутренний объект операционной системы,
  // требующий закрытия после своего использования.
  private IntPtr handle;
  // А это уже управляемые ресурсы среды .NET
  private Component Components;
  // Специальный внутренний флаг, определяющий была ли произведена
  // деинициализация объекта.
  private bool m_disposed = false;
  // Конструктор данного класса.
  public Some()
  {
    // Здесь будет размещаться инициализирующий
    // код объекта.
  }
   // Опишем метод Dispose интерфейса IDisposable
   // ВНИМАНИЕ! Не делайте это метод виртуальным,
   // чтобы потомки не смогли перекрыть его.
   public void Dispose()
   {
     // Поместим код в критическую секцию, чтобы
     // избежать возможных неприятностей, если метод будет вызван
     // сразу же из двух потоков.
     lock(typeof(BaseResource))
     {
       Dispose(true);
       // Предотвращаем автоматический вызов Finalize
       // при сборке мусора, чтобы метод Dispose
       // не был вызван повторно.
       GC.SuppressFinalize(this);
     }
   }
   // Следующий метод является служебным и закрытым. Обращения к нему
   // происходят только из функции Dispose с параметром true, а также
   // из финализатора (Finalize/"деструктор") с параметром false.
```

```
// Если передан параметр true, значит функция вызвана пользователем
// и следует освободить как управляемые, так и неуправляемые ресурсы.
// Если параметр равен false, то функция вызвана финализатором
// и нужно освободить только неуправляемые ресурсы,
// так как управляемые
// будут автоматически освобождены самим сборщиком мусора.
// Данный метод специально объявлен виртуальным, так как он собственно
// и занимается реальным освобождением ресурсов.
protected virtual void Dispose(bool disposing)
{
  // Проверим, вызывался ли метод ранее.
  // Это всего лишь предосторожность, чтобы не получить ошибку
  // при повторном вызове метода Dispose.
  // То есть проверим, в каком состоянии находится наш объект
  // в текущий момент.
  if(!this.m_disposed)
  {
    // Если параметр, переданный методу = true,
    // то освобождаем управляемые
    // ресурсы, так как настоящий деструктор никогда не  вызовется,
    // поскольку мы подавили его, вызвав при помощи
    // метода GC.SuppressFinalize(this);
    if(disposing)
    {
      // Освобождаем управляемые ресурсы.
      Components.Dispose();
    }
    // Теперь освобождаем неуправляемые (уровня операционной ситемы)
    // ресурсы, к примеру вызовем CloseHandle.
    CloseHandle(handle);
    // На всякий случай обнулим описатель.
    handle = IntPtr.Zero;
  }
  // Устанавливаем флаг, говорящий о том, что ресурсы уже освобождены
  // и объект более не доступен.
  m_disposed = true;
}
// Объявим финализатор, который будет вызван только в том
// случае, если до этого не был вызван метод Dispose.
```

```
  ~Some()
  {
    // В самом деструкторе не стоит очищать ресурсы,
    // вместо этого лучше централизованно поместить весь
    // код очистки в метод Dispose.
    // Вызываем Dispose с параметром false, который говорит
    // о том, что нужно очистить только неуправляемые ресурсы.
    Dispose(false);
  }
  // Внимание! Не делайте этот метод виртуальным, чтобы класс
  // потомок не смог перекрыть его и изменить его поведение.
  public void Close()
  {
    // Переадресуем вызов к методу Dispose.
    Dispose();
  }
  // Некоторый метод, который производит некоторую операцию,
  // предоставляемую классом.
  public void DoSomething()
  {
    // Проверяем, не был ли удален наш объект.
    if(this.disposed)
    {
      // Если объекта больше не существует, выбрасываем
      // соответствующее исключение.
      throw new ObjectDisposedException();
    }
    // Здесь выполняется соответствующая операция.
  }
}
```

Данный шаблон рекомендуется самой Microsoft. Он оптимален, поскольку при вызове метода `Dispose` деинициализируются все ресурсы, используемые объектом, в том числе и управляемые. Правда, здесь необходимо отметить, что они всего лишь деинициализуются, реально освобождены они будут только после вызова процесса сборки мусора.

В случае если от класса, реализующего интерфейс `IDisposable`, необходимо объявить класс-потомок, то стоит придерживаться следующего шаблона, который гарантирует правильную деинициализацию ресурсов (листинг 8.6).

Листинг 8.6. Шаблон, предназначенный для классов-потомков

```
/*
  Листинг 8.6
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Класс, являющийся потомком класса, поддерживающего детерминированное
// уничтожение, основанное на вышеприведенном шаблоне.
public class MyResourceWrapper: Some
{
  // Некоторые управляемые ресурсы, которые необходимы
  // классу для работы.
  private ManagedResource addedManaged;
  // А это некие неуправляемые ресурсы, которые предстоит вернуть
  // операционной системе.
  private NativeResource addedNative;
  // Специальный внутренний флаг, определяющий, была ли произведена
  // деинициализация объекта.
  private bool m_disposed = false;
  // Обычный конструктор.
  public MyResourceWrapper()
  {
    // Конструктор базового класса будет вызван автоматически.
    // Здесь предполагается размещение инициализирующего кода.
  }

  // Метод очистки Dispose, перекрывает аналогичный
  // в классе-предке. Это достигается за счет использования модификатора
  // override, а также потому, что, данный в классе-предке Some,
  // он описан как виртуальный
  protected override void Dispose(bool disposing)
  {
    // Проверим, не освобождался ли данный объект ранее.
    if(!this.m_disposed)
    {
      // Введем защищенный блок, для того чтобы гарантировать
      // очистку ресурсов, зарезервированных базовым классом.
      try
```

```
      {
        // Если происходит ручное уничтожение объекта,
        // освободим управляемые ресурсы.
        if(disposing)
        {
          addedManaged.Dispose();
        }
        // Теперь освободим неуправляемые ресурсы.
        CloseHandle(addedNative);
        // Установим флаг, говорящий о том, что объект был уничтожен.
         this.m_disposed = true;
      }
      // Блок завершения.
      finally
      {
        // Освобождаем ресурсы, зарезервированные базовым классом.
        base.Dispose(disposing);
      }
   }
}
```

Обратите внимание на то, что очистка объекта в методе `Dispose` происходит абсолютно безопасно, с гарантированным вызовом метода `Dispose` базового класса. Это достигается за счет использования защищенного участка, с блоком завершения `finally`, который гарантирует обращение к методу `Dispose` базового класса.

Обратите внимание, в классе-потомке `MyResourceWrapper` не вводится собственный метод `Close`. Версии введенной в классе `Some` вполне хватает для полноценной переадресации к методу `Dispose`.

Особо бы хотелось отметить, что объект обязан выбрасывать исключение `ObjectDisposedException` в случае, если пользователь пытается провести операцию на деинициализированном объекте.

using — оптимизация работы с методом *Dispose*

Поскольку `Dispose` является обычным методом, соответственно работать с ним предполагается вручную. Причем при работе с методом необходимо учитывать возникновение нештатных ситуаций (выброс исключений), которые могут помешать его вызову. Следовательно, необходимо использовать специальные защищенные участки с блоками завершения `finally`, из которых

уже обращаться к методу Dipose. Что, конечно же, неудобно, потому как загромождает код излишними подробностями. Это, безусловно, понимали и создатели среды .NET. Они предложили ввести в язык C# специальный оператор using, позволяющий оптимизировать работу с методом Dispose. Работает он следующим образом: в качестве параметра он принимает некоторый объект, поддерживающий модель детерминированного уничтожения (IDisposable), а в свое тело позволяет включить набор операций, после выполнения которых будет вызван метод Dipsose. Использование данного оператора выглядит следующим образом.

```
using (Some s = new Some())
{
        s. DoSomething();
}
```

Конструкция будет транслирована компилятором в следующую (для упрощения исходный код представлен на языке C#):

```
Some s = null;
try
{
  s = new Some();
  s.DoSomething();
}
finally
{
  if (s != null)
    s.Dispose();
}
```

Даже если операция DoSomething вызовет исключение, объект все равно будет уничтожен.

Краткий итог

Подводя краткий итог, можно сделать вывод о том, что среда исполнения предоставляет два механизма уничтожения объектов, не имеющих между собой ничего общего. Причем их использование не является интуитивно понятным и простым. Что, естественно, приведет к появлению ошибок не только у начинающих программистов, но даже и у опытных людей. Само по себе введение дуальной модели освобождения ресурсов еще простить можно, но вот безответственность при проектировании языков высокого уровня, из-за которой финализаторы были вероломно замаскированы под деструкторы, понять, а тем более простить никак нельзя. Деструктор — это давно устоявшееся понятие, требующее от объекта совершенно иного поведения,

нежели чем от финализатора. Многие программисты, знакомые с понятием деструктора, будут крайне удивлены, когда обнаружат, что в среде .NET они таковыми вовсе и не являются. И хорошо, если эта "особенность" их поведения будет обнаружена ими на стадии разработки программ, а не пользователями, приходящими в ужас от неверной работы приложения.

8.3. Внутреннее устройство сборщика мусора

Рассмотрим класс `GC`, предоставляющий сервисы для взаимодействия со сборщиком мусора, а затем обратимся к теоретическому описанию его внутренних механизмов.

Класс *GC*

В табл. 8.1 представлено краткое описание членов класса `GC`.

Таблица 8.1. *Класс* `GC`

Член класса	Описание
Свойства	
`MaxGeneration`	Возвращает максимальное количество поколений (всегда возвращает 3)
Методы	
`Collect`	Инициирует внеочередную сборку мусора
`GetGeneration`	Позволяет узнать номер поколения заданного объекта
`GetTotalMemory`	Возвращает суммарный объем памяти, зарезервированный приложением
`KeepAlive`	Позволяет предотвратить уничтожение объекта во время сборки мусора, даже если на него не осталось ни одной ссылки из управляемого кода
`ReRegisterForFinalize`	Добавляет финализатор указанного объекта в очередь завершения
`SuppressFinalize`	Убирает финализатор указанного объекта из очереди завершения
`WaitForPendingFinalizers`	Приостанавливает текущий поток до тех пор, пока не будет вызван последний финализатор

Главной особенностью класса является то, что все его методы определены статическими. По сути дела, он является лишь оберткой для функций, управляющих памятью. И действительно, ведь менеджер памяти принципиально может быть только один, следовательно, введение объекта GC есть всего лишь дань объектно-ориентированной моде. Что, в общем-то, не так уж и плохо, поскольку четко позволят указать назначение методов.

GC.Collect

Метод инициирует внеочередную сборку мусора

```
public static void Collect();
public static void Collect(
       // Глубина поколений.
   int generation
);
```

С первым методом все понятно, он вызывает сборку мусора. А второй метод позволяет произвести неполную сборку мусора с указанной глубиной выборки поколений. Про поколения будет рассказано далее.

GC.GetGeneration

Возвращает номер поколения для указанного объекта.

```
public static int GetGeneration(
   object obj
);
public static int GetGeneration(
   // Слабая ссылка.
   WeakReference wo
);
```

Метод также позволяет получить номер поколения для объекта, на который указывает слабая ссылка.

GC.GetTotalMemory

Возвращает общее количество байт, зарезервированных приложением.

```
public static long GetTotalMemory(
   bool forceFullCollection
);
```

В случае, если метод будет вызван во время сборки мусора и параметр forceFullCollection будет равен true, метод подождет некоторое время, пока сборщик мусора не освободит значительную часть памяти. Время ожидания данного метода будет зависеть от количества внутренних циклов

сборщика мусора, а также от количества памяти, освобождаемой за каждый цикл. Что самое интересное, данный метод может и не дождаться окончания сборки мусора, если решит, что время задержки велико.

GC.KeepAlive

Позволяет предотвратить уничтожение объекта во время сборки мусора, даже если все ссылки на него потеряны.

```
public static void KeepAlive(
   object obj
);
```

Данный метод крайне удобен, когда в управляемой среде все ссылки на объект потеряны, но четко знаешь, что из неуправляемого кода он еще используется. Соответственно будет крайне неприятно, если сборщик мусора уничтожит важный объект.

GC.ReRegisterForFinalize

Регистрирует финализатор указанного объекта в специальной внутренней очереди сборщика мусора.

```
public static void ReRegisterForFinalize(
   object obj
);
```

Особенность метода заключается в том, что сколько раз при помощи него вы зарегистрируете финализатор объекта в этой очереди, столько раз он и будет вызван во время сборки мусора. Продемонстрируем на примере, в котором объявим финализатор, выводящий на консоль приветствие, а затем дважды зарегистрируем его в очереди завершения (листинг 8.7). После чего будем ждать сборки мусора, которая будет проведена при закрытии приложения.

Листинг 8.7. Множественный вызов финализатора во время сборки мусора

```
/*
  Листинг 8.7
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем экспериментальный класс с финализатором.
class Some
{
```

```
  // Это финализатор класса, он будет
  // выводить сообщение на консоль перед уничтожением данного объекта.
  ~Some()
  {
    Console.WriteLine("Hello, World!");
  }
}
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим новый экземпляр типа Some.
    // Обратите внимание, что здесь произойдет первая
    // регистрация финализатора объекта s в очереди завершения.
    Some s = new Some();
    // Зарегистрируем финализатор в очереди завершения еще пару раз.
    GC.ReRegisterForFinalize(s);
    GC.ReRegisterForFinalize(s);
  }
}
```

В результате работы данного приложения на консоль будут выведены следующие строки.

```
Hello, World!
Hello, World!
Hello, World!
```

С первого взгляда может удивить, ведь мы регистрировали финализатор лишь два раза, а сообщение было выведено три. Но если вспомнить, что финализатор автоматически регистрируется в очереди завершения при создании объекта (`new`), то все становится на свои места.

GC.SupressFinalize

Метод изымает из очереди завершения финализатор указанного объекта.

```
public static void SuppressFinalize(
   object obj
);
```

Прошу обратить внимание на то, что если финализатор некоторого объекта был добавлен в очередь завершения более одного раза, то для того чтобы полностью отказаться от его вызова, придется ровно столько раз обратиться к методу `SupressFinalize`.

GC.WaitForPendingFinalizers

Позволяет дождаться завершения работы всех финализаторов, вызываемых сборщиком мусора в параллельном потоке.

```
public static void WaitForPendingFinalizers();
```

При использовании метода необходимо опасаться окончательного и бесповоротного блокирования потока, в котором он был вызван, которое может пройти по причине блокирования одного из финализатора. То есть возникнет взаимная блокировка (`deadlock`), при которой работа приложения намертво застопорится. Продемонстрируем на примере (листинг 8.8), в котором финализатор одного из объектов будет блокировать процесс сборки мусора.

Листинг 8.8. Блокировка сборки мусора одним из финализаторов

```
/*
  Листинг 8.8
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Введем экспериментальный класс, финализатор которого
// будет самоблокироваться.
class Some
{
   // Финализатор данного класса.
  ~Some()
  {
    // Подождем немного (~27 часов).
    Thread.Sleep(100000000);
  }
};
```

```
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Создаем новый объект, который сразу же будет
    // помечен как мусорный потому, что не сохраняем ссылку на него.
    new Some();
    // Инициируем сборку мусора.
    GC.Collect();
    // Подождем, пока не завершится сборка мусора и вызов всех
    // финализаторов.
    GC.WaitForPendingFinalizers();
    // Выведем на консоль приветствие, сообщающее от том,
    // что главный поток приложения вышел из блокировки.
    Console.WriteLine("Hello, World!");
  }
};
```

Хотя в документации указано, что поток, в котором исполняются финализаторы, может блокироваться не более чем на 40 секунд, по истечении этого срока менеджер памяти среды исполнения должен обнаруживать блокировку и завершать поток, в котором вызываются финализаторы. Но, к моему удивлению, этого не произошло, приложение было намертво заблокировано, и мне так и не удалось дождаться снятия блокировки.

У меня возникло предположение, что взаимная блокировка не обнаруживается средой исполнения из-за особенностей работы функции `Sleep`, и я попробовал использовать вместо нее явную взаимную блокировку на основном потоке приложения. Код новой модификации блокировки представлен в листинге 8.9.

Листинг 8.9. Взаимная блокировка финализатора на главном потоке приложения

```
/*
  Листинг 8.9
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
```

```
using System;
// Подключим пространство имен, отвечающее за поддержку
// работы с потоками.
using System.Threading;
// Введем класс, финализатор которого будет вызвать
// явную взаимную блокировку на главном потоке приложения.
class Some
{
  // Объект, предоставляющий доступ к основному
  // потоку приложения.
  private Thread mainThread;
  // Конструктор-класс, необходимый для инициализации
  // поля mainThread.
  public Some(Thread th)
  {
    mainThread = th;
  }
  // Финализатор данного класса.
  ~Some()
  {
    // Ожидаем завершения главного потока приложения, вызывая
    // тем самым взаимную блокировку.
    mainThread.Join();
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим новый объект, который будет сразу же помечен как
    // мусорный, потому что мы не сохраняем ссылку на него.
    // Передадим его конструктору ссылку на объект, представляющий
    // текущий поток.
    new Some(Thread.CurrentThread);
    // Инициируем сборку мусора.
    GC.Collect();
    // Дождемся завершения вызова всех финализаторов.
```

```
    GC.WaitForPendingFinalizers();
    // Выведем на консоль приветствие, сообщающее о том,
    // что главный поток приложения вышел из блокировки
    Console.WriteLine("Hello, World!");
  }
}
```

Это приложение, точно как и предыдущее, было заблокировано в вечном ожидании снятия взаимной блокировки.

Детерминированное уничтожение объектов при помощи финализаторов

Рассказ о финализаторах был бы неполным, если не скрыть от читателя возможность детерминированного (явного) уничтожения объектов при помощи финализаторов. Как было рассказано — исполнение данной задачи является принципиально невозможным. Но как говорится — когда нельзя, но очень хочется, то можно. Идея заключается в следующем: необходимо уничтожить все ссылки на объект и вручную инициировать сборку мусора. Делается это следующим образом.

```
// Создаем объект.
Some refSome = new Some();
...
// Здесь мы работаем с объектом.
...
// Уничтожаем ссылку на объект.
refSome = null;
// Инициируем внеочередную сборку мусора.
GC.Collect();
```

Единственный недостаток подхода заключается в том, что помимо данного объекта сборщик мусора будет вынужден уничтожить также все остальные мусорные объекты. Что может вызвать недопустимые задержки при вызове метода `GC.Collect`. Поэтому лучше все-таки пользоваться методом `Dispose`. Хотя если его нет, то данный подход будет единственным выходом их сложившегося положения.

Воскрешение объектов

Во время вызова финализатора объекта, формально сам объект еще существует, хотя и находится в стадии уничтожения. Следовательно, есть надежда, что объект все-таки можно восстановить, предотвратив его уничтожение.

Но как это сделать? Оказывается, все тривиально просто, необходимо лишь восстановить хотя бы одну ссылку на объект. Поскольку наличие ссылок, указывающих на объект, среда исполнения проверяет непосредственно перед его удалением, следовательно, объект не будет уничтожен. Приведем пример, который продемонстрирует это (листинг 8.10).

Листинг 8.10. Воскрешение объектов через финализаторы

```
/*
  Листинг 8.10
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;

// Введем дополнительный экспериментальный класс, объект
// которого будем затем восстанавливать.
class Some
{
  // Конструктор класса, инициализирующий объект.
  public Some(string str)
  {
    m_str = str;
  }
  // Финализатор, воскрешающий объект.
  ~Some()
  {
    // Восстанавливаем ссылку на объект, тем самым воскрешая его
    App.m_Some = this;
    // Сообщим пользователю о том, что был вызван финализатор
    // (деструктор) объекта.
    Console.WriteLine("~Some was invoked");
  }
  // Служебное внутреннее поле необходимо для хранения сообщения,
  // которое позволит идентифицировать целостность объекта после
  // его воскрешения.
  String m_str;
```

```
  // Метод, выводящий приветствие, хранимое внутри объекта.
  // Если воскрешение объекта не удастся, то вызов метода
  // приведет к непредсказуемым последствиям.
  public void SayHello()
  {
    Console.WriteLine(m_str);
  }
}
// Основной класс приложения.
class App
{
  // Ссылка на объект, который будет воскрешаться; она объявлена
  // за пределами метода для того, чтобы сделать ее доступной
  // для финализатора класса Some.
  public static Some m_Some;
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим новый экземпляр объекта Some.
    m_Some = new Some("Hello, World!");
    // Уничтожим последнюю ссылку на объект.
    m_Some = null;
    // Инициируем внеочередную сборку мусора.
    GC.Collect();
    // Сделаем невозможное — обратимся к несуществуемому объекту.
    // Двумя строками ранее мы явно уничтожили последнюю ссылку на него.
    m_Some.SayHello();
  }
}
```

Удивительно, но в результате работы программы, на консоль будут выведены следующие строки.

```
~Some was invoked
Hello, World!
```

Ранее в этой главе мы уже использовали технологии воскрешения. Правда, там объект воскрешал себя из конструктора, а затем вызвал исключение, не давая программе сохранить на него ссылку. Тем самым, с одной стороны, он препятствовал своему созданию, а с другой стороны, сам же себя и воскрешал.

Поколения как технология оптимизации сборки мусора

Естественно, что при разработке среды исполнения ее создатели шли на всяческие ухищрения, дабы повысить скорость ее работы. Не обошло стороной это стремление и создателей сборщика мусора. Они применили специальную технологию поколений, для оптимизации процесса сборки мусора. Фокус заключается в предложении ввести поколения объектов, по аналогии со старением людей. По определению, поколение всех вновь выделяемых объектов равно нулю. То есть все вновь выделяемые объекты являются молодыми. После очередной сборки мусора, поколение выживших объектов увеличивается на единицу — они стареют. Всего поколений в .NET может быть три: нулевое (0), первое (1) и второе (2). Большее количество поколений вводить смысла не имело. "А где же оптимизация?" — спросите вы. Оптимизация заключается в том, что когда количество поколений больше одного, сборка мусора производится только для младших поколений. И только в том случае, если в ходе сборки мусора среди младших поколений не удалось освободить необходимое количество памяти, тогда будет произведена сборка мусора среди старших поколений.

Основная идея данного подхода заключается в том, что если уж объект пережил одну или несколько сборок мусора, то вероятность того, что он станет мусорным, гораздо меньше. Следовательно, можно пренебречь малыми потерями памяти среди старших поколений. Сказанное иллюстрируется схемой на рис. 8.6.

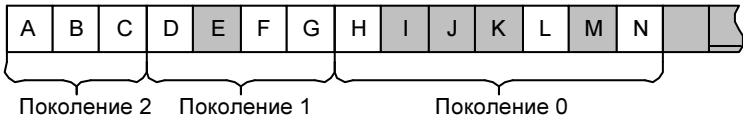


Рис. 8.6. Плотность потерь памяти в различных поколениях

Как видно из схемы, основные потери памяти, то есть возникновение мусорных объектов, наиболее вероятны в младших поколениях. Именно поэтому имеет смысл собирать мусор именно в них. Безусловно, что мусор будет образовываться и в старших поколениях. Но в них объекты уже как бы устоялись, и вероятность их уничтожения гораздо меньше, чем недавно выделенных.

Также довольно интересен механизм увеличения поколений. Как будет показано дальше, он практически не накладывает на сборщика мусора никакой нагрузки.

По определению, все выделяемые объекты помечаются как представители нулевого поколения. Таким образом, на первом этапе работы программы ситуация будет выглядеть, как представлено на рис. 8.7.

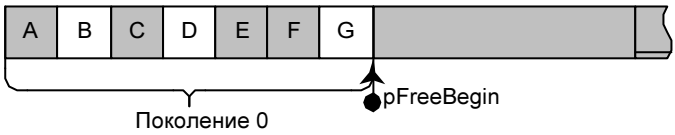


Рис. 8.7. Представители нулевого поколения в самом начале работы приложения

Затем, когда порог выделенной памени превысит некоторый заданный предел, будет произведена сборка мусора, в результате которой каждый из выживших объектов постареет на одно поколение (рис. 8.8).

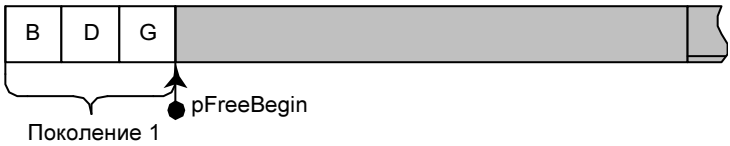


Рис. 8.8. Первая сборка мусор и старение объектов

После этого объекты поколения 1 на некоторое время будут оставлены в покое, среда исполнения будет следить лишь за вновь прибывшими объектами. Помечая их как объекты нулевого поколения (рис. 8.9).

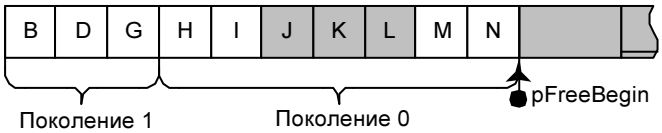


Рис. 8.9. Второй этап в работе приложения, где присутствуют уже два поколения

Точно так же, как и в первом случае, после переполнения критического объема занимаемым нулевым поколением будет произведена сборка мусора. Единственное отличие от предыдущего случая будет заключаться в том, что среда исполнения просканирует также остальные выжившие объекты и повысит их номера поколений. Если, конечно, это возможно, поскольку предельный номер поколения равен двум. После очередной сборки мусора ситуация будет выглядеть следующим образом (рис. 8.10).

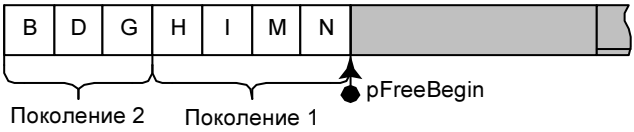


Рис. 8.10. Положение дел после второй сборки мусора

Для старших поколений существуют свои допустимые пределы, но, в отличие от нулевого поколения, проверяются не при каждом выделении памяти под объекты, а только в процессе сборки мусора.

В завершении раздела приведем пример, демонстрирующий увеличение поколений объекта при проведении сборок мусора (листинг 8.11). После каждой сборки мусора, на консоль будет выводиться номер поколения выжившего объекта. Естественно, что сборка мусора будет инициирована искусственно.

Листинг 8.11. Демонстрация увеличения поколения объектов во время сборки мусора

```
/*
  Листинг 8.11
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем дополнительный экспериментальный класс,
// объекты которого будут сообщать номера поколений, к которым они
// принадлежат.
class GenObj
{
  // Финализатор класса поможет нам уловить момент уничтожения объекта.
  ~GenObj()
  {
    // Информируем пользователя об уничтожении объекта
    Console.WriteLine("GenObj destructor was called");
  }
  // Данный метод будет сообщать нам о поколении экземпляра
  // данного класса, для которого он вызван.
  public void DisplayGen()
  {
    Console.WriteLine("Generation of this object equal = {0}",↳
GC.GetGeneration(this));
  }
};
// Основной класс приложения.
class App
{
```

```
  // Точка входа в приложение.
  public static void Main()
  {
    // Для начала сообщим пользователю о максимальном числе
    // поколений, поддерживаемых средой исполнения.
    Console.WriteLine("Maximum GC generations: {0}", GC.MaxGeneration);
    // Создадим тестовый объект.
    GenObj p = new GenObj();
    // Выведем на консоль информацию о поколении данного объекта.
    p.DisplayGen();
    // Проведем сборку мусора, вследствие чего поколение нашего
    // объекта увеличится на единицу.
    GC.Collect();
    // Опять проверим поколение нашего объекта.
    p.DisplayGen();
    // Повторим вышеописанную операцию еще несколько раз.
    GC.Collect();
    p.DisplayGen();
    GC.Collect();
    p.DisplayGen();
  }
};
```

В результате работы данного приложения на консоль будут выведены следующие строки.

```
Maximum GC generations: 2
Generation of this object equal = 0
Generation of this object equal = 1
Generation of this object equal = 2
Generation of this object equal = 2
Gen destructor was called
```

Видно, что номер поколения не может превысить числа 2, сколько бы раз ни производилась сборка мусора.

Большие объекты

Объекты, занимающие в памяти более 20 Кбайт, считаются средой исполнения большими объектами и не участвуют в общем "круговороте объектов". Такие объекты резервируются в специальной области памяти, и к ним не применяется механизм поколений. Сделано это исходя из соображений

производительности. Поскольку при каждой сборке мусора происходит передвижение многих объектов, при помощи банального копирования памяти. А применительно к большим объектам это было бы весьма накладно.

8.4. Слабые ссылки *WeakReference*

Многие программы резервируют большое количество участков памяти различного размера. Начиная от маленьких и заканчивая очень большими. В целях оптимизации, было бы удобно некоторые из таких участков выгружать, когда в них нет непосредственной необходимости. Но тут перед разработчиками встает вопрос — ведь надо разрабатывать дополнительную систему контроля памяти, динамически выгружающие ненужные участки. А стоит ли оно того? Ведь надо будет затратить определенные усилия на создание и отладку данной системы, оправдает ли она себя? К счастью, разработчики среды .NET заранее позаботились о нас и создали подобный механизм, позволяющий автоматически контролировать распределение памяти. Этот механизм представлен единственным классом — `WeakReference`. Его конструктор представлен ниже.

```
public WeakReference (object target);
public WeakReference (object target, bool trackResurrection );
```

Объекты данного класса представляют собой так называемые слабые ссылки. Идея заключается в следующем: если на объект указывает лишь только одна слабая ссылка, то он помечается как мусорный. Соответственно, во время ближайшей сборки мусора он будет уничтожен. Тем не менее, в любой момент через слабую ссылку можно получить нормальную строгую ссылку на объект. Если на этот момент он еще не будет уничтожен, то с него будет снят флаг "мусорности". Получение ссылки производится при помощи свойства `Target`.

Данный механизм чрезвычайно удобен для хранения ссылок на редко используемые объекты, содержащие в своих недрах большие массивы данных. В случае если память, занимаемая одним из таких объектов, понадобится для других нужд программы, то он будет безболезненно удален, без какого бы то ни было участия со стороны программиста. Когда же программе понадобится данный объект и она попытается получить к нему доступ, то ей будет возвращена пустая ссылка. В чем, к счастью, нет ничего страшного, поскольку это будет говорить лишь о том, что необходимо переинициализировать объект.

Продемонстрируем работу технологии на примере. В нем будет введен квазибольшой объект `BigTable`. Квази потому, что большим он лишь подразумевается, на самом же деле таковым не является. Тем не менее, наш пример будет обращаться с ним по всем правилам больших объектов, — для доступа к нему будем использовать слабую ссылку. В ходе работы приложения

специально инициируем сборку мусора, которая уничтожит интересующий нас объект. После этого для того чтобы воспользоваться объектом, нам придется воссоздать объект вновь. Исходный код примера вы найдете в листинге 8.12.

Листинг 8.12. Демонстрация использования слабых ссылок для работы с большими объектами

```
/*
  Листинг 8.12
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Введем специальный класс BigTable, который будет представлять
// квазибольшой объект. Он редко используется, и его
// не обязательно все время хранить в памяти.
class BigTable
{
  // Введем данные, которые легко инициализировать и которые
  // будут реально храниться вместе объектом.
  private String internalString = "DoIt method was called";
  // Введем метод, который подтвердит работоспособность объекта.
  public void DoIt()
  {
    Console.WriteLine(internalString);
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Создаем квазибольшой объект.
    BigTable p = new BigTable();
    // Создаем слабую ссылку, указывающую на объект.
    WeakReference wr = new WeakReference(p);
    // Уничтожаем последнюю строгую ссылку на наш объект.
    // Теперь его больше ничто не держит в памяти, хотя мы еще
    // можем получить доступ к нему при помощи слабой ссылки,
```

```
    // созданной чуть раньше.
    p = null;
    // Инициируем сборку мусора, в ходе которой объект будет уничтожен.
    // На него будет указывать лишь слабая ссылка.
    GC.Collect();
    // Попытаемся получить строгую сильную ссылку через слабую.
    p = (BigTable)wr.Target;
    // ОБЯЗАТЕЛЬНО проверим, удалось ли нам получить сильную ссылку.
    // Иными словами, проверим, существует ли еще на данный
    // момент наш объект в памяти. Если нет, то придется
    // воссоздать его.
    if (p == null)
    {
      // Сообщим пользователю о том, что объект уничтожен
      // и будет создан заново.
      Console.WriteLine("Object destructed!\n Creating new");
      // Создаем объект.
      p = new BigTable();
      // Обновим слабую ссылку.
      wr = new WeakReference(p);
    }
    // Проверим работоспособность нашего объекта.
    p.DoIt();
  }
}
```

В результате работы приложения, на консоль будут выведены следующие строки.

```
Object destructed!
 Creating new
DoIt method was called
```

Особо хотелось бы обратить ваше внимание на подход к использованию слабой ссылки (листинг 8.13).

Листинг 8.13. Обобщенный шаблон работы со слабыми ссылками

```
/*
  Листинг 8.13
  File:   Some.cs
  Author: Дубовцев Алексей
```

```
*/
// Попытаемся получить строгую сильную ссылку через слабую.
p = (BigTable)wr.Target;
// ОБЯЗАТЕЛЬНО проверим, удалось ли нам получить сильную ссылку,
// иными словами, проверим существует ли еще на данный
// момент наш объект в памяти. Если не существует, то придется
// воссоздать его.
if (p == null)
{
  // Сообщим пользователю о том, что объект уничтожен
  // и будет создан заново.
  Console.WriteLine("Object destructed!\n Creating new");
  // Создаем объект.
  p = new BigTable();
  // Обновим слабую ссылку.
  wr = new WeakReference(p);
}
```

Сначала получаем строгую ссылку. Затем необходимо проверить, удался ли этот запрос. В случае неудачи воссоздаем объект заново, не забыв обновить слабую ссылку на него. Эти четыре шага строго обязательны, именно в той последовательности, в которой они здесь представлены. Они представляют собой не просто предосторожность, а скорее отражают всю сущность механизма работы слабых ссылок.

Таким образом, в среде .NET имеется два вида ссылок: слабые и строгие (рис. 8.11).

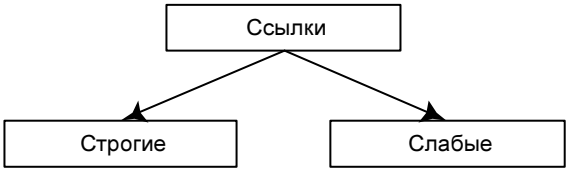


Рис. 8.11. Два вида ссылок, используемых в среде .NET

По крайней мере, подобной терминологии придерживается Mircosoft. Ей, конечно, виднее — она ведь все-таки как никак является создателем данной технологии. Тем не менее, мне кажется несколько неверным, что слабые ссылки вообще причислены к классу ссылок. И вот по каким соображениям, — априори предполагается, что ссылка может быть использована для доступа к объекту. Но по отношению к слабым ссылкам данное правило не работает, что может ввести в заблуждение неискушенного пользователя. Он ведь ожидает при помощи ссылки получить доступ к объекту.

Слабые ссылки, по сути дела, ссылками вообще не являются, они представляют собой механизм, позволяющий держать объект на грани жизни и смерти. Для доступа к нему все равно придется использовать обыкновенную строгую ссылку, другого пути нет!

Также необходимо сказать пару слов о самих объектах, при использовании которых следует использовать слабые ссылки. Прежде всего, к таким объектам предъявляются весьма строгие требования по скорости создания. Объект, на который будет указывать слабая ссылка, может многократно создаваться вновь и вновь во время работы программы. Соответственно, если на его создание и инициализацию уходит много времени, то это может существенно затормозить работу приложения. Также необходимо учесть уникальность данных объекта, они должны быть абсолютно точно восстановлены в случае уничтожения объекта, которое весьма вероятно при использовании слабых ссылок.

Короткие и длинные ссылки

Помимо строгих и слабых ссылок существуют еще два вида ссылок: короткие и длинные. Данное деление производится среди слабых ссылок (рис. 8.12).

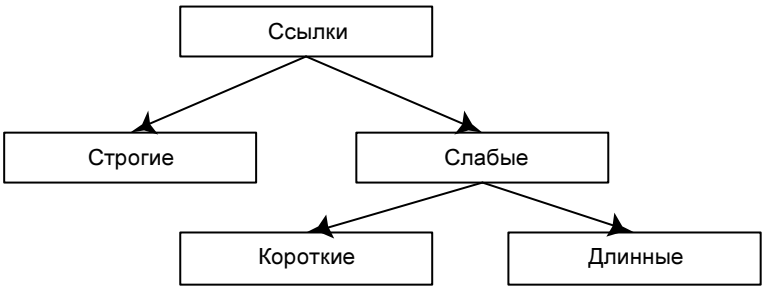


Рис. 8.12. Все виды ссылок, используемых в среде .NET

Различие между данными видами ссылок с первого взгляда весьма несущественно. Короткие ссылки "зануляются" непосредственно после начала операции по уничтожению объекта, на который они указывают. Длинные же ссылки, в отличие от коротких, принимают нулевое значение только перед самим уничтожением объекта. Для того чтобы понять различие, вспомним, что операция уничтожения объекта многоступенчата: прежде всего, время уходит на вызов финализатора и уж совсем мизер на само физическое уничтожение объекта. И если над объектом уже колдует сборщик мусора, вызывая его финализатор и собираясь в ближайшем будущем уничтожить его, то существует вероятность, что мы еще успеем выхватить объект из его лап при помощи длинной ссылки. Что, конечно же, не удастся сделать, используя короткую ссылку.

Подобная функциональность весьма сомнительна, поскольку, получив объект при помощи длинной ссылки, мы не можем знать, в каком состоянии он находится. Весьма вероятно, что он уже был безвозвратно деинициализирован своим финализатором. Соответственно, использовать данную возможность надо с крайней осторожностью.

Класс *WeakReference*

В заключение подраздела приведем краткое описание класса `WeakReference` (табл. 8.2).

Таблица 8.2. *Класс WeakReference*

Член класса	Описание
Свойства	
`IsAlive`	Позволяет определить, доступен ли объект, на который указывает слабая ссылка
`Target`	Возвращает объект, на который указывает слабая ссылка
`TrackRessurection`	Позволяет определить, является ли ссылка длинной или короткой

Данный класс не имеет собственных методов, он вводит лишь свойства, при помощи которых можно взаимодействовать со слабыми ссылками.

Класс имеет два конструктора

```
public WeakReference (object target );
public WeakReference (object target, bool trackResurrection );
```

Первый позволяет создавать исключительно короткие ссылки, указывающие на объект, переданный в качестве первого параметра. Второй позволяет создавать как короткие, так и длинные ссылки, для этого лишь необходимо передать либо false, либо true соответственно.

WeakReference.IsAlive

Позволяет определить, не уничтожен ли еще объект, на который указывает слабая ссылка.

```
public virtual bool IsAlive {get;}
```

При помощи данного метода можно слегка модифицировать стандартный шаблон использования слабых ссылок. Изменение будет заключаться в том, что проверка доступности реального объекта по слабой ссылке будет проверяться с использованием данного метода.

```
// ОБАЗАТЕЛЬНО проверим, доступен ли объект
```

```
if (wr.IsAlive())
  // Если он еще не уничтожен, получим строгую ссылку на него,
  p = (BigTable)wr.Target;
// иначе придется воссоздать объект
else
{
  // Создаем объект
  p = new BigTable();
  // Обновим слабую ссылку
  wr = new WeakReference(p);
}
```

Принципиальное отличие данного подхода заключается в том, что для проверки целостности объекта не используется строгая ссылка, после получения которой объект автоматически перестает быть мусорным. В приведенном выше примере работы со слабой ссылкой, с использованием `IsAlive`, есть одно очень слабое место. Опасность заключается в том, что сборка мусора может произойти непосредственно после вызова метода `IsAlive`. Таким образом, проверка удастся — программа заключит, что объект уцелел и передаст управление коду, запрашивающему строгую ссылку. Но именно в этот момент объект будет уничтожен, и вместо строгой ссылки будет возвращена пустая ссылка. Что, конечно же, приведет к непоправимым последствиям — выбросу исключения `NullReferenceException`, которое сделает дальнейшую работу программы невозможной.

WeakReference.Target

Свойство возвращает строгую ссылку на объект

```
public virtual object Target {get; set;}
```

Обратите внимание, что ссылка возвращается в виде объекта класса `Object`. Для того чтобы получить доступ, настоящему объекту придется воспользоваться операцией приведения типов.

WeakReference.TrackRessurection

Позволяет определить, является ли данная слабая ссылка длинной или короткой.

```
public virtual bool TrackResurrection {get;}
```

В случае если ссылка короткая, будет возвращено значение false, если длинная — true.

Внутреннее устройство слабых ссылок

У пытливых читателей может возникнуть вопрос, но каким же образом слабые ссылки сохраняют связь с объектом, ведь они вроде бы являются экземплярами обычного класса. А все ссылки из подобных объектов должны рассматриваться как строгие, что, безусловно, будет предотвращать посягательства на данный объект со стороны сборщика мусора. И действительно, класс `WeakReference` является обычным классом, не имеющим особой поддержки со стороны виртуальной машины .NET. А для того чтобы создать подобную функциональность, он использует вполне легальные документированные механизмы, предоставляемые классом `GCHandle`, которые и позволяют обмануть пресловутый сборщик мусора. Класс `WeakReference` запрашивает у данного класса описатель объекта, являющийся не чем иным, как его обычным адресом в памяти. Когда же к слабой ссылке обращаются за строгой, она запрашивает строгую ссылку у все того же класса `GCHandle`, при помощи ранее полученного указателя.

Таким образом, внутренний механизм поддержки слабых ссылок использует классические указатели без обратной связи, которые в данном случае оказались незаменимыми.

8.5. Блокировка объектов в памяти

При взаимодействии с управляемой средой, что, кстати говоря, происходит довольно часто, среде исполнения приходится отказаться от ссылок и использовать настоящие указатели на объекты. Другого выхода нет, ведь неуправляемый код не имеет ни малейшего представления о ссылках, а соответственно, не умеет с ними работать. А как мы уже выяснили, во время сборки мусора многие объекты меняют свое положение в памяти, что не так уж и страшно, когда применяются ссылки и совершенно недопустимо при использовании указателей. Для решения данной проблемы, был создан механизм блокировки объектов в памяти. Данная технология называется Memory Pinning, что в дословном переводе означает, — фиксация памяти.

Примечание

В переводе с английского слово pin означает — булавка, кнопка, шпилька, прищепка. Отсюда и название технологии.

Суть заключается в том, что программист по собственному желанию может заблокировать объект, в памяти запретив сборщику мусора передвигать. После чего можно работать с таким объектом при помощи классических указателей, не опасаясь того, что он будет перемещен.

Продемонстрируем использование данной технологии на примере. Закрепив объект в памяти, будем использовать его поле через классический указатель, поиграв при этом со сборкой мусора (см. листинг 8.14).

Листинг 8.14. Демонстрация технологии блокирования объектов в памяти

```
/*
  Листинг 8.14
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки классов
using System;

// Данный класс будет располагаться в управляемой области памяти
// и подчиняться всем правилам, по которым играет сборщик мусора
class ClassInHeap
{
  // На этот член класса мы будем ссылаться через указатель
  public int Value;
};

// Основной класс приложения
public class App
{
  // Объявим специальный модификатор unsafe, для того, что бы получить
  // возможность работы с указателями, а не только со ссылками
  unsafe static void SomeProc ( int* p)
  {
    // Удвоим значение переменной, на которую ссылается данный
    // указатель, прибавив ее саму к себе
    *p += *p;
  }

  // Точка входа в приложения
  // Объявим специальный модификатор unsafe, для того чтобы получить
  // возможность работы с указателями, а не только со ссылками
  unsafe public static void Main()
  {
    // Специально создадим мусорный объект,
```

```
    // для того чтобы гарантировать перемещение следующих за
    // ним в памяти, в процессе сборки мусора
    new ClassInHeap();

    // Создадим экземпляр класса
    ClassInHeap cih = new ClassInHeap();

    // Устанавливаем начальное значение члена только что созданного
    // экземпляра
    cih.Value = 4;

    // Фиксируем в памяти положение члена класса
    // и объявляем указатель на него
    fixed (int* ptr = &cih.Value)
    {
      // Произведем сборку мусора, в результате которой
      // объект cih должен был переместиться в памяти, если бы
      // ранее не был фиксирован
      GC.Collect();

      // Вызываем нашу функцию, передав ей указатель на член
      SomeProc(ptr);
    }

    // А теперь убедимся в том, что значение реального члена было
    // изменено через указатель
    Console.WriteLine("Value of Point.x = {0}",cih.Value);
  }
};
```

В результате работы данной программы на консоль будет выведена следующая строка.

```
Value of Point.x = 8
```

Особо необходимо обратить внимание, что для использования указателей пришлось воспользоваться ключевым словом unsafe. Оно говорит о том, что в коде используются небезопасные операции. А это не совсем хорошо, поскольку исполнение подобного кода может быть запрещено политикой безопасности.

8.6. Производительность

Уверен, что большинство программистов при использовании технологий .NET прежде всего будет волновать вопрос производительности. Как бы ни была хороша и удобна технология, но если она медленна и малопроизводительна, то естественно, что ее использование будет не оправдано.

Приятно отметить, что программисты, создававшие виртуальную машину .NET, изрядно потрудились, особенно пристальное внимание они уделили проектированию и реализации подсистемы памяти. Они прекрасно понимали, что память является основным ключом к производительности .NET-приложений и приложили максимум усилий, дабы оптимизировать работу с ней .NET-приложений.

Сборка мусора и потоки

Интуитивно понятно, что во время самого процесса сборки мусора исполнение любых потоков крайне нежелательно, поскольку может привести к коллизиям обращения к памяти. Наиболее простым решением является приостановка всех потоков приложения. Но, как очевидно, это крайне неоптимально, поскольку приложение фактически будет заморожено на протяжении всего процесса сборки мусора. К тому же приостанавливать работу неуправляемых потоков отнюдь не безопасное занятие, поскольку может кардинально нарушить логику их работу.

Для того чтобы решить эти проблемы, были разработаны две технологии: взлом (Hijacking) и безопасные точки (Safe Points). Первая применяется при контролировании неуправляемых потоков, а вторая для работы с управляемыми.

- Взлом. Если один из потоков находится в неуправляемой области кода, то среда исполнения сканирует стек данного потока и встраивает туда специальную заглушку, перехватывающую контроль после возвращения в управляемую область. Как только управление передается этой заглушке, она приостанавливает работу потока, вследствие чего становится возможна безопасная сборка мусора.
- Безопасные точки. Для каждой функции компилятор JIT встраивает в управляемый код специальные безопасные точки, на которых работа кода может быть в любой момент приостановлена и передана под контроль среды исполнения. Безопасные точки вставляются с тем учетом, чтобы можно было максимально быстро приостановить работу потока с наименьшими задержками.

Тем не менее, необходимо отметить, что сборка мусора так или иначе нуждается в приостановке работы потоков, что может несколько замедлить

работу приложения. Но данные издержки столь незначительны, что ими вполне можно пренебречь.

Исполнение на мультипроцессорных системах

При исполнении на мультипроцессорных системах возможны две конфигурации среды исполнения: обычная и серверная.

В обычном режиме для каждого из потоков вводится специальная область памяти, в которой сохраняются все объекты, выделенные в данном потоке и имеющие нулевое поколение. А для объектов старших поколений используется общее хранилище (рис. 8.13).

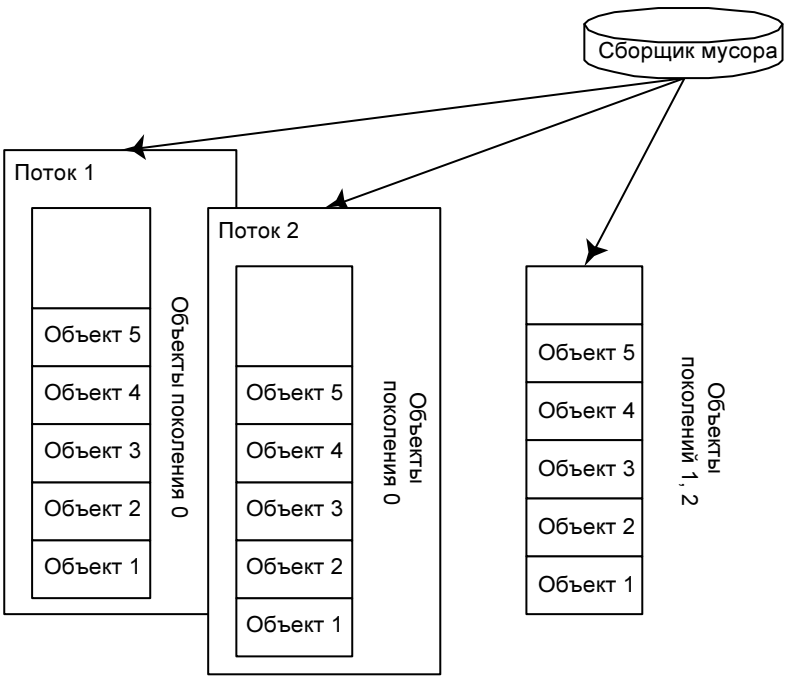


Рис. 8.13. Среда исполнения на мультипроцессорной системе, работающая в обычном режиме

В серверном режиме для каждого процессора существует свой экземпляр сборщика мусора, который отвечает за потоки, исполняющиеся на данном процессоре (рис. 8.14).

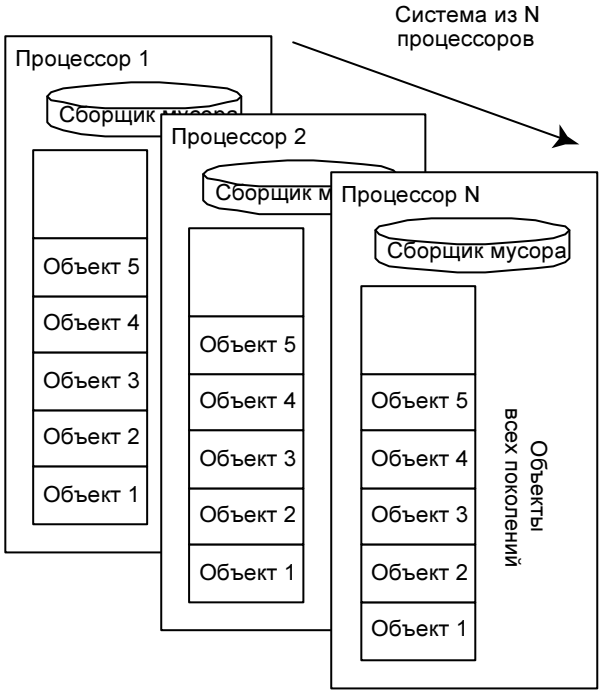


Рис. 8.14. Среда исполнения на мультипроцессорной системе, работающая в серверном режиме

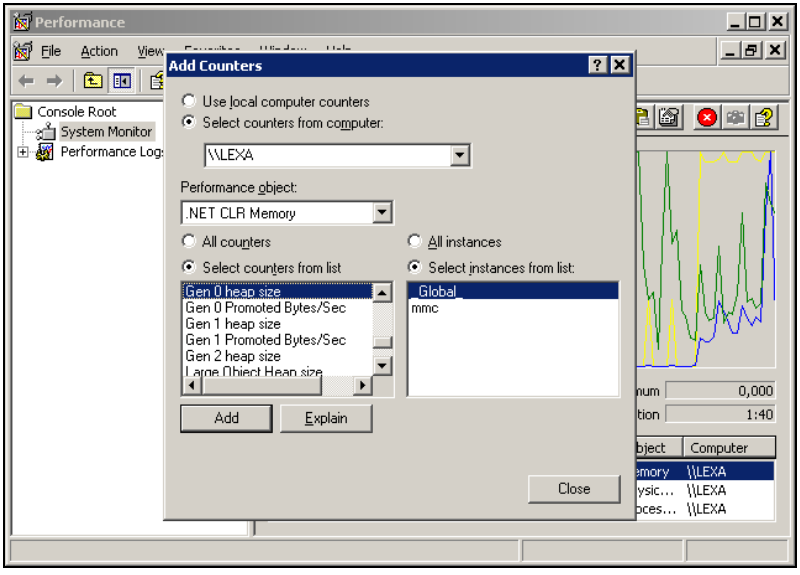


Рис. 8.15. Датчики производительности, регистрирующие внутренние показания процессов, происходящих внутри среды .NET

Слежение за программой

Для особо пытливых разработчиков и пользователей, чрезмерно заботящихся о качестве своих приложений, среда исполнения представляет набор датчиков производительности. Они помогут получить информацию о различных параметрах процессов, происходящих внутри среды исполнения.

Наиболее простой и удобный способ получить доступ к датчикам предполагает использование оснастки Performance консоли управления, которая, к сожалению, будет доступна только пользователям Windows 2000/XP (рис. 8.15).

Среда исполнения предоставляет множество датчиков, но в данном случае нас будет интересовать лишь группа .NET CLR Memory.

Каждый из доступных датчиков кратко описан в табл. 8.3.

Таблица 8.3. *Набор датчиков производительности для снятия показаний с памяти*

Название датчика	Описание
# Bytes in all Heaps	Общее количество памяти, зарезервированное программами
# GC Handles	Количество описателей (GCHandle), используемых сборщиком мусора
# Gen X Collections	Количество объектов поколения X
# Induced GC	Количество вынужденных сборок мусора, вызванных нехваткой памяти при попытке создать новый объект
# of Pinned Objects	Количество закрепленных (фиксированных) в памяти объектов
# of Sink Block in use	Объекты синхронизации используют специальные блоки подключения. Датчик показывает количество таких блоков
# Total committed Bytes	Общее количество реально используемых байт памяти
# Total reserved Bytes	Общее количество зарезервированных байт
% Time GC *	Отношение времени, проведенное программой для сборки мусора, к времени исполнения других потоков
Allocated Bytes/sec *	Количество байт, резервируемых за секунду. Правда, с одной оговоркой, данное значение обновляется только при сборке мусора
Finalizations Survivors	Количество воскрешенных объектов
Gen X heap size	Общий объем, занимаемый объектами поколения X
Gen X Promoted Bytes/sec *	Объем памяти, занимаемой объектами, которые увеличили поколение во время текущей сборки мусора

Таблица 8.3 *(окончание)*

Название датчика	Описание
Large Object Heap size	Общий объем памяти, занимаемый объектами, размер которых превышает 20 Кбайт
Promoted Memory from Gen 0	Количество памяти, занимаемое объектами, которые пережили сборку мусора и увеличили свое поколение с 0 на 1
Promoted Memory from Gen 1	Количество памяти, занимаемое объектами, которые пережили сборку мусора и увеличили свое поколение с 1 на 2

Наиболее интересно добавить все датчики разом и следить за их изменениями не на графике, а в режиме отчета, когда выводятся их точные показания.

8.7. Заключение

При первом знакомстве с технологиями управления памятью, используемыми в среде .NET, меня достаточно сильно обрадовало название "сборщик мусора", сразу же вспомнилось что-то нехорошее, подкрепленное смутными воспоминаниями о фильме "Газонокосильщик".

Глава 9

Потоки

В главе рассмотрены принципы и механизмы работы потоков в среде .NET, которые несколько отличаются от принятых в операционной системе. Несмотря на кажущуюся простоту описываемых здесь сервисов и механизмов, эта тема является наиболее сложной в прикладном программировании. Поэтому от читателя потребуются значительные усилия для усвоения материала.

Введение

Все знают, что операционная система Windows способна одновременно исполнять несколько приложений на одном компьютере. Но мало кому известно, что настоящая многозадачность осуществима только на многопроцессорных системах, а на обычных компьютерах она возможна только благодаря хитроумному механизму потоков, основанному на принципе вытеснения.

Windows является операционной системой с вытесняющей многозадачностью. Вытеснение заключается в том, что операционная система в рамках процесса планирования потоков выделяет каждому их них квант процессорного времени. По истечении этого времени поток принудительно вытесняется операционный системой и процессор переключается на исполнение кода другого потока. Следует отметить, что это не единственная схема реализации многозадачных систем. К примеру, существуют системы, в которых задача планирования потоков полностью лежит на самих прикладных программах, а не на операционной системе. Рассмотрение подобных систем выходит за рамки данной книги.

На уровне операционной системы существуют два фундаментальных понятия: процесс и поток. Поток — это код, реально исполняющийся на процессоре, а процесс — это абстрактный контейнер, объединяющий несколько потоков в единое приложение и предоставляющее им персональное закрытое адресное пространство. Таким образом, процессы являются контейнерами, содержащими потоки (рис. 9.1).

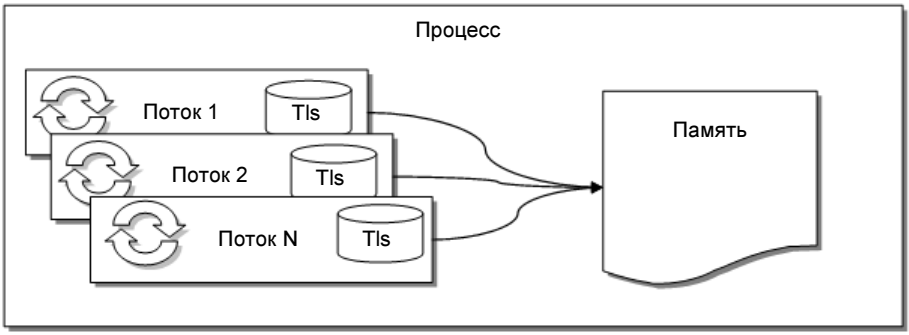


Рис. 9.1. Общая схема устройства процессов и потоков

Важно понять, что процессы ничего не выполняют, это всего лишь обертки для потоков, которые уже реально исполняют код приложения.

Все потоки имеют доступ к общей памяти приложения, выделенной им процессом, но помимо этого у каждого из них есть собственная локальная память. Она так и называется — локальная память потока (Thread Local Storage, TLS). К этой памяти может получить доступ только сам поток, что зачастую бывает не только удобно, но даже необходимо.

9.1. Приступим к делу

Все потоки в приложениях .NET можно разделить на два вида: управляемые и неуправляемые. Первые создаются самим .NET-приложением или средой исполнения, а вторые — неуправляемым кодом. Неуправляемые потоки чаще всего создаются неявно, при вызове некоторых внешних сервисов операционной системы или при взаимодействии с внешними COM-объектами. Каждый поток представлен объектом типа `System.Threading.Thread`, независимо от того, является он управляемым или нет.

Любые операции, производимые над потоками, контролируются средой исполнения. Любые изменения сразу будут отражены в соответствующем объекте `Thread`. Оказывается, виртуальная машина .NET перехватывает все сервисы управления потоками и ей не составляет труда их контролировать.

Читателям, знакомым с программированием потоков для Windows, могут оказаться непонятными причины введения новой модели управления потоками. Не проще ли было создать классы-обертки для уже существующего Windows Threading API. Оказывается, что не проще. Среда .NET является многоплатформенной и может работать под управлением различных операционных систем, механизмы управления потоками в которых разительно различаются. А с введением собственной модели управления потоками среда .NET унифицирует работу с потоками для всех поддерживаемых платформ.

Сравнение сервисов управления потоками Windows API и .NET

Тем, кто имел опыт работы с потоками при помощи сервисов Windows API, будет небезынтересно ознакомиться со сравнительной характеристикой Windows- и .NET-сервисов управления потоками (табл. 9.1).

Таблица 9.1. *Сравнение сервисов управления потоками в Windows и .NET*

Действие	Windows API	Сервисы среды исполнения .NET
Создание потока	`CreateThread`	Конструктор класса `Thread`. `Thread.Start`
Уничтожение потока	`TerminateThread`	`Thread.Abort`
Приостановление работы потока на неограниченное время	`SuspendThread`	`Thread.Suspend`
Приостановление работы потока на определенное время	`Sleep`	`Thread.Sleep`
Ожидание завершения работы потока	`WaitForSingleObject`	`Thread.Join`
Принудительный выход из потока	`ExitThread`	Не поддерживается
Получение текущего потока	`GetCurrentThread`	`Thread.CurrentThread`
Изменение приоритета потока	`SetThreadPriority`	`Thread.Priority`
Задание имени потока	Не поддерживается	`Thread.Name`
Определение фонового потока	Не поддерживается	`Thread.IsBackground`
Выбор потоковой модели	`CoInitializeEx`	`Thread.ApartmentState`

При первом рассмотрении может показаться, что среда исполнения предоставляет несколько больше сервисов управления потоками, чем Windows. И хотя потоки среды .NET полностью базируются на потоках среды Windows, это действительно так. Некоторые дополнительные возможности среда .NET эмулирует самостоятельно.

Создание собственного потока

Прежде всего, необходимо научиться создавать собственные потоки. Это необходимо для того, чтобы лучше понять их сущность и устройство.

Каждый поток в среде .NET представлен объектом `Thread`, поэтому для создания собственного потока, по крайней мере, придется образовать объект такого типа. Класс `Thread` имеет единственный конструктор.

```
public Thread (
  ThreadStart start
);
```

В качестве параметра он принимает пустой делегат следующего типа.

```
public delegate void ThreadStart();
```

Он должен ссылаться на пользовательскую функцию, код которой будет исполняться в потоке. Следует учесть, что создание объекта не подразумевает автоматического запуска потока. Поток будет создан в спящем режиме, и для того чтобы запустить его, необходимо будет воспользоваться следующим методом.

```
public void Thread.Start();
```

Продемонстрируем создание потока на простейшем примере (листинг 9.1). В нем будет два потока: основной и пользовательский. Оба будут одновременно выводить на консоль строки, каждый в своем бесконечном цикле. Поскольку оба потока будут работать одновременно, то на консоли мы увидим довольно интересный результат.

Листинг 9.1. Простейший пример работы с потоками

```
/*
  Листинг 9.1
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку
// работы с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
```

```
  // Это и есть функция второго потока, которая будет исполняться
  // параллельно с основным потоком, то есть с функцией Main.
  static void ThreadProc()
  {
    // Сообщим пользователю уникальный хеш потока.
    Console.WriteLine("GetHashCode of Second Thread = {0}",
Thread.CurrentThread.GetHashCode());
    for (;;)
    {
      Console.WriteLine("Hello, World, from second thread!");
      Thread.Sleep(500);
    }
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Сообщим пользователю уникальный хеш основного потока
    // приложения.
    Console.WriteLine("GetHashCode of Main Thread = {0}",
Thread.CurrentThread.GetHashCode());
    // Создадим специальный делегат, указывающий на функцию,
    // которую мы хотим поместить в отдельный поток.
    ThreadStart trStart = new ThreadStart(App.ThreadProc);
    // Создадим новый объект потока.
    Thread th = new Thread(trStart);
    // Запустим поток на исполнение.
    th.Start();
    // При надобности можем дождаться завершения
    // созданного потока.
    // th.Join();
    for (;;)
    {
      Console.WriteLine("Hello, World, from first thread!");
      Thread.Sleep(500);
    }
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
GetHashCode of Main Thread = 2
GetHashCode of Second Thread = 3
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
Hello, World from second thread!
Hello, World from first thread!
^C
```

В приложении было два потока, которые одновременно выводили на консоль строки. Символ ^C в конце лога говорит о том, что работа приложения была прервана нажатием комбинации клавиш **Ctrl-C**.

9.2. Класс *Thread*

Рассмотрим более детально класс `Thread` (см. табл. 9.2).

Таблица 9.2. *Описание членов класса* `System.Threading.Thread`

Член класса	Описание
Свойства	
ApartmentState	Позволяет изменить тип COM апартамента для данного потока

Таблица 9.2 *(продолжение)*

Член класса	Описание
Свойства	
 CurrentContext	Позволяет получить текущий контекст взаимодействия для данного потока
CurrentCulture	Возвращает текущие для данного потока региональные установки
CurrentPrincipal	Возвращает специальный объект защиты текущего потока
CurrentThread	Возвращает объект `Thread`, представляющий текущий поток
CurrentUICulture	Возвращает региональный идентификатор пользовательского интерфейса данного потока
IsAlive	Позволяет определить, исполняется ли в данный момент поток
IsBackground	Позволяет определить, является ли поток фоновым
IsThreadPoolThread	Позволяет определить, пришел ли данный поток из управляемого пула потоков
 Name	Возвращает имя потока
 Priority	Возвращает приоритет потока
 ThreadState	Позволяет определить текущее состояние потока
Методы	
 Abort	Приостанавливает исполнение потока
 AllocateDataSlot	Резервирует слот в персональной памяти всех потоков
 AllocateNamedDataSlot	Резервирует именованный слот в персональной памяти всех потоков
 FreeNameDataSlot	Освобождает именованный слот, находящийся в персональной памяти потоков
 GetDomain	Возвращает домен, в котором исполняется данный поток
 GetDomainID	Возвращает уникальный идентификатор домена, в котором исполняется данный поток
 GetNamedDataSlot	Возвращает слот в персональной памяти потока по его имени

Таблица 9.2 *(окончание)*

Член класса	Описание
Методы	
Interrupt	Прерывает исполнение потока
Join	Позволяет дождаться завершения работы потока
ResetAbort	Отменяет запрос на уничтожение потока
Resume	Запускает ранее приостановленный поток
SetData	Записывает данные в указанный слот персональной памяти потока
Sleep	Приостанавливает исполнение потока на заданное время
Start	Запускает поток на исполнение
Suspend	Приостанавливает работу потока

Несмотря на кажущуюся сложность и громоздкость класса `Thread`, использовать его в работе достаточно просто. Далее изучим основные приемы работы с ним, а также более подробно рассмотрим некоторые из его членов.

Уничтожение потоков

Ранее мы уже научились создавать и запускать потоки, теперь необходимо научиться выполнять обратную задачу — уничтожать потоки. Для реализации этих целей предназначен метод `Abort` класса `Thread`.

```
public void Thread.Abort();
public void Thread.Abort( object obj );
```

Весьма интересным является механизм его работы. Метод выбрасывает специальное системное исключение `ThreadAbortException`, которое, развернув весь стек потока, инициирует его изъятие из фильтра необработанных исключений. Основная особенность этого исключения состоит в том, что его невозможно подавить при помощи `catch`-блоков. Отловить удастся, но подавить — нет! Если объявить соответствующий `catch`-блок с фильтром для типа `ThreadAbortException`, то исключение поймано будет. Только вот после того как блок отработает, оно будет повторно выброшено внутренними сервисами среды исполнения.

Для того чтобы действительно подавить уничтожение потока, необходимо воспользоваться следующим методом.

```
public static void Thread.ResetAbort();
```

Только он гарантирует действительную приостановку процесса уничтожения потока и сможет подавить перевыброс исключений после `catch`-блоков.

Такой странный и громоздкий механизм уничтожения потоков создан неспроста, он необходим для того, чтобы гарантировать вызов всех `finally`-блоков, даже в случае уничтожения потока.

Для демонстрации процесса уничтожения и "воскрешения" потока рассмотрим пример (листинг 9.2). Он будет пытаться уничтожить свой главный поток, а затем перехватывать исключение `ThreadAbortException` и восстанавливать его работу.

Листинг 9.2. Подавление процесса уничтожения потока

```
/*
  Листинг 9.2
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Точка входа в приложения.
  // Эта функция выполняется в контексте основного потока
  // приложения.
  public static void Main()
  {
    // Объявим защищенный блок.
    try
    {
      // Попробуем прервать работу нашего приложения.
      Thread.CurrentThread.Abort();
    }
```

```
    // Соответственно, отловим исключение ThreadAbortException, которое
    // сигнализирует о попытке завершения потока.
    catch (ThreadAbortException ex)
    {
      // Сообщим пользователю о выбросе исключения.
      Console.WriteLine("ThreadAbortException was thrown");
      // Отменим завершение потока, подавив исключение
      // ThreadAbortException.
      // ВНИМАНИЕ! Если этот вызов убрать, то поток будет уничтожен,
      // поскольку обработчик catch принципиально не может отловить
      // исключение типа ThreadAbortException.
      Thread.ResetAbort();
    }
    // В случае если поток будет уничтожен, данная строка выведена на
    // консоль не будет, в противном случае мы ее увидим.
    Console.WriteLine("Hello, World!");
  }
};
```

В результате работы программы на консоль будут выведены следующие строки.

```
ThreadAbortException was thrown
Hello, World!
```

Приведем более сложный пример самовосстановления потока (листинг 9.3). Здесь пользовательский поток будет занят эмуляцией бурной деятельности, в то время как основной поток приложения попытается его уничтожить. Однако исследуемый поток, обнаружив это, восстановит себя и сообщит об этом пользователю.

Листинг 9.3. Демонстрация самовосстановления потока

```
/*
  Листинг 9.3
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
```

```
// с потоками.
using System.Threading;
// Основной класс приложения.
public class App
{
  // Эта функция будет эмулировать сложную работу.
  public static void DoSomeInterestingWork(String str)
  {
    for(;;)
    {
      Console.WriteLine(str);
      Thread.Sleep(100);
    }
  }
  // Это функция второго потока.
  public static void ThreadProc()
  {
    String work;
    // Это сообщение наш поток будет непрерывно выводить на консоль
    // до тех пор, пока его не попробуют закрыть. В этом
    // случае сообщение будет изменено.
    work = "Normaly";
    // Объявим цикл, который в случае выхода из защищенного блока
    // запустит поток заново.
    for (;;)
      // Объявим защищенный блок, который позволит отловить
      // попытку закрытия потока.
      try
      {
        // Пусть наш поток что-то делает.
        DoSomeInterestingWork(work);
      }
      // Этот обработчик будет вызываться в случае выброса исключения
      // ThreadAbortException, вызываемого в случае осуществления
      // попытки закрытия потока.
      catch (ThreadAbortException e)
      {
        // Сообщим пользователю о попытке
        // закрытия потока.
```

```
        Console.WriteLine("Exception message: {0}", e.Message);
        Console.WriteLine("He-he somebody try to terminate me");
        // Восстановим работоспособность потока.
        Thread.ResetAbort();
        // Изменим сообщение, которое наш поток непрерывно выводит
        // на консоль.
        work = "Life after end";
      }
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим делегат, указывающий на функцию, которая будет
    // исполнять код нового потока.
    ThreadStart thStart = new ThreadStart(App.ThreadProc);
    // Создадим объект, представляющий второй поток.
    Thread th = new Thread(thStart);
    // Запустим на исполнение созданный поток.
    th.Start();
    Thread.Sleep(1000);
    // Попытаемся прервать работу потока.
    th.Abort();
    // Дождемся завершения работы потока. А если точнее, то не
    // дождемся, поскольку наш поток будет работать вечно.
    th.Join();
  }
};
```

В результате работы программы будут выведены следующие сообщения.

```
...
Normaly
Normaly
Normaly
Exception message: Thread was being aborted.
He-he somebody try to terminate me
Life after end
Life after end
Life after end
...
```

При использовании такого приема по предотвращению уничтожения потоков, необходимо учитывать, что работа потока будет прервана в самый неожиданный момент. А восстановить его работу именно с той точки, на которой он был прерван исключением, в большинстве случаев не представляется возможным. Единственное, что можно сделать автоматически, так это перезапустить поток, а вот восстановление его предыдущего состояния полностью возлагается на разработчика.

На этом особенности работы с функцией `Abort` не заканчиваются. Достаточно интересно поведение функции, если поток, к которому она применяется, находится в нерабочем состоянии.

Самый простой случай возникает, когда метод `Abort` вызван еще до старта потока. Тогда исключение `ThreadAbortException` будет вызвано непосредственно после начала работы потока. Сложнее вариант, когда метод `Abort` будет вызван при нахождении потока в неуправляемой области кода. Скажем, он обращается к некоторой системной функции или внешнему COM-объекту. В этом случае метод `Abort` дождется возвращения потока в управляемую область и тотчас выбросит исключение `ThreadAbortException`. Таким образом, завершение потока может затянуться весьма надолго, особенно, если поток в неуправляемой области занимается чем-то долгосрочным.

Надо отметить, что в обоих случаях после вызова метода `Abort`, состояние потока изменится на `ThreadState.AbortRequested`.

В самом начале подраздела мы упоминали функцию `Abort`, способную принимать параметр типа `obj`. В листинге 9.4 показан пример, который демонстрирует, каким образом при помощи этого параметра можно передавать информацию о причине уничтожения потока.

Листинг 9.4. Использование метода `Abort`, принимающего параметр

```
/*
  Листинг 9.4
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
```

```
{
  // Это функция второго потока приложения.
  public static void ThreadProc()
  {
    // Объявим защищенный блок, который позволит нам отловить
    // закрытие потока.
    try
    {
      // Эмулируем бурную деятельность.
      for(;;);
    }
    catch(ThreadAbortException ex)
    {
      // Выведем на консоль дополнительную информацию,
      // переданную методом Abort.
      Console.WriteLine(ex.ExceptionState.ToString());
    }
  }
  // Точка входа в приложение.
  public static void Main()
  {
  // Создадим новый объект, представляющий поток.
    Thread th = new Thread( new ThreadStart(App.ThreadProc));
  // Запустим поток на исполнение.
    th.Start();
  // Подождем, когда поток действительно начнет исполняться.
  Thread.Sleep(1000);
  // Прервем поток, задав при этом дополнительную информацию.
    th.Abort("Hello, World!");
  // Дождемся завершения потока.
    th.Join();
  }
};
```

В результате работы приложения на консоль будет выведена строка.

```
Hello, World!
```

Информация о состоянии потоков *ThreadState*

За время свой жизни поток проходит через различные состояния. Самые простые из них: рабочее и приостановленное (спящее). Но помимо них существует множество других — каждый их них представлен соответствующим членом перечисления `ThreadState` (табл. 9.3).

Таблица 9.3. *Описание членов перечисления* `ThreadState`

Член перечисления (Флаг)	Состояние потока
`Aborted`	Поток находится в остановленном (нерабочем) состоянии
`AbortRequest`	Был получен запрос на уничтожение потока (`Thread.Abort`), но он все еще исполняется
`Background`	Поток находится в фоновом режиме
`Running`	Поток исполняется
`Stopped`	Поток остановлен
`StopRequested`	Был получен запрос на остановку потока, но он все еще исполняется (флаг предназначен только для внутреннего использования)
`Suspended`	Поток приостановлен
`SuspendRequested`	Был получен запрос на приостановку работы потока, но он все еще исполняется
`Unstarted`	Поток еще не начал выполняться, то есть функция `Thread.Start` еще не была вызвана
`WaitSleepJoin`	Исполнение потока приостановлено в результате работы одной из функций `Wait`, `Sleep`, `Join`

По началу несколько могут запутать термины "остановленный" и "приостановленный". Первый означает, что поток остановлен окончательно и бесповоротно, а его исполнение не может быть возобновлено. Он может лишь быть перезапущен заново. А второй означает, что исполнение потока было приостановлено лишь временно и будет возобновлено позднее, при возникновении определенных условий.

Получить информацию о текущем состоянии потока можно при помощи следующего свойства.

```
public ThreadState Thread.ThreadState { get; }
```

С проверкой флагов, обозначающих состояние потока, связан важный нюанс. Флаг `Running` имеет значение `0`, поэтому провести его проверку при

помощи одной битовой операции не удастся. Microsoft рекомендует осуществлять проверку при помощи следующей конструкции.

```
if ((SomeThread.ThreadState & (ThreadState.Unstarted |
ThreadState.Stopped)) == 0)
{
   // Сюда мы попадем только при условии, что поток SomeThread
   // находился на момент проверки в рабочем состоянии.
}
```

Если условие выполнится, то поток, представленный объектом `SomeThread`, выполнялся на момент проверки условия.

Получение объекта, представляющего текущий поток

Для взаимодействия с любым потоком в среде .NET необходим объект `Thread`. Хорошо, если мы сами создавали поток — в этом случае необходимый объект находится в нашем распоряжении изначально. В противном случае его придется добывать самостоятельно. Для примера можно привести основной поток приложения, создаваемый средой исполнения. Именно в его контексте исполняется функция `Main`. Для того чтобы получить объект, представляющий текущий поток, необходимо воспользоваться следующим статическим свойством.

```
public static Thread CurrentThread();
```

Оно возвращает объект `Thread`, представляющий текущий поток. Необходимо помнить, что, управляя текущим потоком через полученный объект, вы косвенно управляете функцией, в которой находитесь. Работать с объектом нужно очень осторожно, иначе можно заблокировать или повредить поток таким образом, что его дальнейшая работа будет невозможна.

Встроенные механизмы синхронизации потоков

Класс `Thread` предоставляет встроенные механизмы синхронизации потоков, реализованные тремя функциями.

```
// Приостанавливает работу потока на неограниченное время.
public void Thread.Suspend();
// Возобновляет работу потока.
public void Thread.Resume();
// Приостанавливает работу потока на заданное время.
public static void Thread.Sleep(int millisecondsTimeout);
public static void Thread.Sleep(TimeSpan timeout);
```

```
// Позволяет дождаться завершения работы потока.
public void Thread.Join();
public bool Thread.Join(int millisecondsTimeout);
public bool Thread.Join(TimeSpan timeout );
```

Рассмотрим методы и способы работы с ними более подробно.

Впервые, когда я столкнулся с задачей приостановки потока, я поступил, казалось бы, очевидным образом — организовал бесконечный цикл, в котором проверял нужное условие. В случае его выполнения должен был происходить выход из цикла и возобновляться нормальная работа потока. Но в какой-то момент приложение стало настолько сильно загружать процессор, что это приводило к значительному падению производительности на уровне всей системы. Оказывается, для приостановки потоков в Windows необходимо использовать только специальные системные средства. Они замораживают исполнение кода потока и запускают его только в случае необходимости, разгружая тем самым процессор. В моем случае поток беспрестанно загружал процессор исполнением своего бесполезного кода.

Для того чтобы наглядно продемонстрировать сказанное, приведем два примера. Оба они занимаются тем, что приостанавливают работу своего потока. Только первый делает это при помощи бесконечного цикла, а второй — при помощи штатных сервисов среды исполнения.

Первое приложение использует для остановки потока бесконечный цикл (листинг 9.5).

Листинг 9.5. Способ приостановки потоков с использованием бесконечного цикла

```
/*
  Листинг 9.5
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Останавливаем поток на неограниченное время.
```

```
    // ВНИМАНИЕ! Никогда так не делайте.
    for(;;);
  }
};
```

Второй пример использует для остановки потока стандартные средства (листинг 9.6).

Листинг 9.6. Стандартный способ приостановки потоков

```
/*
  Листинг 9.6
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
public class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Приостанавливаем поток на неограниченное время
    // стандартными средствами .NET.
    Thread.CurrentThread.Suspend();
  }
};
```

Результаты работы приложений ничем не будут отличаться, они оба блокируют свою работу и как бы зависают. Но нас будут интересовать не столько результаты их работы, сколько показатели производительности системы во время исполнения. Для обоих приложений показания производительности приведены на рис. 9.2, 9.3.

На рисунках представлено две области 1 и 2. Первая соответствует этапу запуска приложения, вторая показывает его рабочее состояние. Видно, что при использовании бесконечного цикла ресурсы системы неоправданно

перегружаются приложением. А при остановке потока стандартными средствами этого не только не происходит, но даже снижается общая загруженность системы.

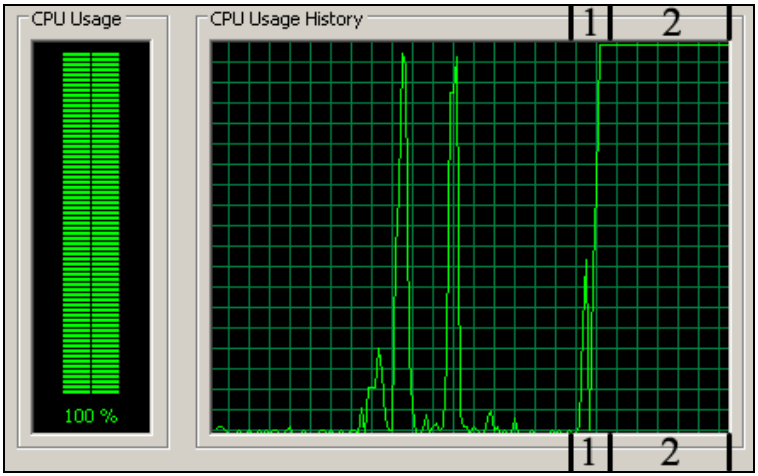


Рис. 9.2. Показатели производительности системы во время приостановки потока с использованием бесконечного цикла

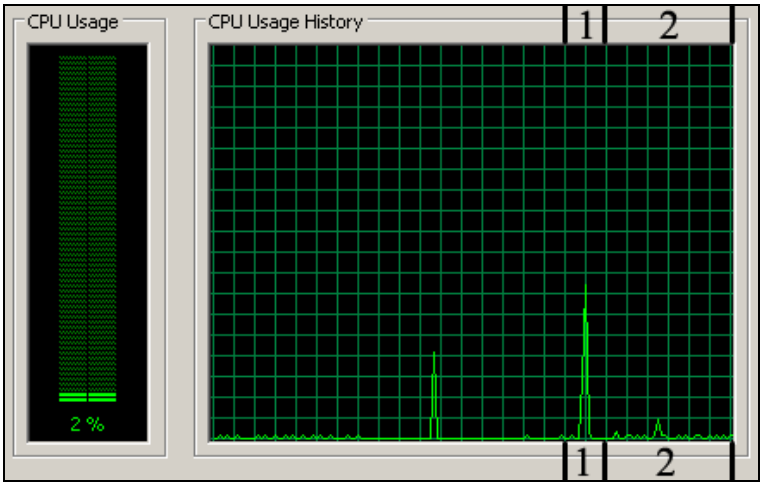


Рис. 9.3. Показатели производительности системы во время приостановки потока стандартными средствами среды исполнения

При помощи функции `Suspend` можно остановить не только текущий поток, но и любой поток, существующий в приложении. Эта функция предназначена не для полного уничтожения потока, а только для его приостановки, следовательно, должен существовать сервис, способный вернуть поток к жизни.

Такой сервис существует, это `Thread.Resume`. Он позволяет запустить ранее приостановленный поток, причем ровно с той точки, на которой он был приостановлен.

Основная и отнюдь не явная особенность работы метода `Suspend` заключается в том, что после его вызова поток приостанавливается не сразу, а только когда дойдет до одной из безопасных точек. Такие точки примечательны тем, что, останавливая в них потоки, можно гарантировать безопасную сборку мусора в приложении. Во время сборки мусора большинство потоков приложения принудительно приостанавливаются в безопасных точках. Но если поток был остановлен заранее вручную, то необходимо и для него гарантировать безопасную сборку мусора. Именно поэтому приостановка работы управляемых потоков происходит только в безопасных точках.

Метод `Suspend` оказывается совершенно непригоден, когда необходимо приостановить работу потока на строго определенное время. Для этого предназначен метод `Sleep`. Он позволяет задать точное время в миллисекундах, на которое следует приостановить работу потока. Единственный недостаток метода заключается в том, что он является статическим и может воздействовать только на текущий поток.

```
public static void Thread.Sleep(int millisecondsTimeout);
public static void Thread.Sleep(TimeSpan timeout);
```

Приостановить работу произвольного потока на заданное время штатными методами не представляется возможным. Можно посоветовать воспользоваться искусственным методом с участием дополнительного потока, который приостановит работу нужного, поспит некоторое время, а затем возобновит его работу. Код функции вспомогательного потока будет выглядеть следующим образом.

```
// Вспомогательный поток, приостанавливающий работу заданного
// на указанное время.
void SubsidiaryThread()
{
  // Приостанавливаем указанный поток.
  SomeThread.Suspend();
  // Ждем заданное время (3 сек).
  Sleep(3000);
  // Возобновляем работу потока.
  SomeThread.Resume();
}
```

Таким образом видно, что найти выход можно всегда, пусть даже схема использования сервисов не будет явной и простой. У метода `Sleep` есть одна

особенность, которая проявляется, если в качестве параметра ему передать константу.

```
public const int Timeout.Infinite;
```

В этом случае поток будет остановлен навсегда. Значение такой константы равно -1. Если его передать вручную, то результат будет таким же.

Иногда требуется дождаться завершения определенного потока. Ситуация довольно жизненная, к примеру, если один из потоков занят некой важной операцией, без завершения которой невозможно дальнейшее функционирование приложения. Для этих целей предназначен метод `Join`.

```
public void Thread.Join();
public bool Thread.Join(int millisecondsTimeout);
public bool Thread.Join(TimeSpan timeout);
```

Он блокирует исполнение текущего потока до тех пор, пока поток, для которого был вызван этот метод, не будет приостановлен. Правда, существуют модификации метода, позволяющие находиться в состоянии ожидания ограниченное время. После чего, даже если поток не был уничтожен, снимается блокировка с ожидающего потока.

При использовании метода необходимо удостовериться в том, что поток находится в рабочем состоянии, иначе будет выброшено исключение `ThreadStateException`. Продемонстрируем такую ситуацию на примере (листинг 9.7). Намеренно попытаемся дождаться завершения не стартовавшего потока.

Листинг 9.7. Попытка дождаться завершения не стартовавшего потока

```
/*
  Листинг 9.7
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу с типами.
using System.Threading;
// Основной класс приложения.
class App
{
  // Это функция нашего потока. Она пустая, поскольку поток мы все равно
  // не будем запускать.
```

```
  public static void ThreadProc()
  {
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим делегат, указывающий на нашу функцию потока.
    ThreadStart thStart = new ThreadStart ( App.ThreadProc );
    // Создадим новый объект, представляющий поток.
    Thread th = new Thread (thStart);
    //Запускать поток намеренно не будем.
    //Thread.Start();
    // Объявим дополнительный защищенный блок для того, чтобы
    // отловить ожидаемый выброс исключения.
    try
    {
      // Попытаемся дождаться завершения не стартовавшего потока.
      th.Join();
    }
    catch (ThreadStateException e)
    {
      // Сообщим пользователю о произошедшей проблеме,
      // присоединив к сообщению описание, возвращаемое средой
      // исполнения.
      Console.WriteLine("Проблема такова - {0}",e.Message);
    }
  }
};
```

В результате работы программы на экран будет выведено сообщение.

```
Проблема такова - Thread has not been started.
```

Что в переводе означает "Поток не запущен".

9.3. Планирование потоков

В сложных приложениях, имеющих множество одновременно работающих потоков, встает задача координирования их совместной работы, называемая планированием потоков. Далее будут описаны приемы и механизмы, позволяющие это осуществить.

Приоритеты потоков

При создании нескольких потоков в приложении обычно каждому из них выделяется собственная задача, которой он занимается. Естественно, что некоторые из задач являются более важными, чем другие. Одни должны быть выполнены в предельно короткие сроки, другие же будут не столько критичны ко времени своего завершения. С целью управления важностью задач, выданных потокам, был введен специальный механизм планирования потоков, основанный на приоритетах. Всего существует пять приоритетов потоков, которые представлены членами перечисления `System.Threading.ThreadPriority` (табл. 9.4).

Таблица 9.4. *Описание членов перечисления* `ThreadPriority`

Значение приоритета	Пояснение
`Lowest`	Поток с самым низким приоритетом, ему будет выделено время только после исполнения всех остальных потоков
`BelowNormal`	Несколько ниже нормального
`Normal`	Нормальный приоритет, с таким приоритетом по умолчанию создаются все управляемые потоки
`AboveNormal`	Немного выше нормального
`Highest`	Самый высокий приоритет

Для установки и получения значения приоритета потока используется следующее свойство.

```
public ThreadPriority Thread.Priority {get; set;}
```

Устанавливая новое значение приоритета потока, мы не просто меняем некоторое значение, а меняем статус важности потока. Потокам с более высокими приоритетами отдается предпочтение, по сравнению с потоками, обладающими низкими приоритетами.

Ранее говорилось, что механизм поддержки потоков в операционной системе Windows является вытесняющим. Каждому потоку выделяется свой квант процессорного времени, в течение которого он монопольно исполняет свой код на процессоре. После чего вытесняется другим потоком и снова ждет своей очереди. Величина кванта процессорного времени и вероятность попадания на процессор всецело зависит от приоритета потока. Чем выше приоритет, тем "длиннее" квант и больше вероятность попасть на процессор.

Наглядно показать работу с приоритетами потоков довольно трудно, тем не менее, попытаемся сделать это. Создадим приложение (листинг 9.8) с двумя потоками, один из которых обладает низким приоритетом, другой — высоким.

Первый будет выводить на консоль единицы, второй двойки, причем оба будут делать это беспрерывно. Запустим и посмотрим, каких цифр на консоли будет больше.

Листинг 9.8. Демонстрация работы двух потоков с различными приоритетами

```
/*
  Листинг 9.8
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Функция первого потока.
  public static void ThreadProc1()
  {
    // Будем непрерывно выводить на консоль число 1.
    for(;;)
      Console.WriteLine("1");
  }
  // Функция второго потока.
  public static void ThreadProc2()
  {
    // Будем непрерывно выводить на консоль число 2.
    for(;;)
      Console.WriteLine("2");
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим делегат для первой функции.
    ThreadStart ths1 = new ThreadStart(App.ThreadProc1);
    // Создадим делегат для второй функции.
```

```
    ThreadStart ths2 = new ThreadStart(App.ThreadProc2);
    // Создадим объект, представляющий первый поток.
    Thread th1 = new Thread(ths1);
    // Создадим объект, представляющий второй поток.
    Thread th2 = new Thread(ths2);
    // Установим приоритет первого потока наименьшим.
    th1.Priority = ThreadPriority.BelowNormal;
    // А приоритет второго потока сделаем нормальным.
    th2.Priority = ThreadPriority.Normal;
    // А это вторая версия распределения приоритетов.
    //th1.Priority = ThreadPriority.Normal;
    //th2.Priority = ThreadPriority.Normal;
    // Запустим оба потока на исполнение.
    th1.Start();
    th2.Start();
    // Приостановим работу нашего потока.
    Thread.CurrentThread.Suspend();
  }
};
```

После запуска приложения на консоли можно будет различить лишь одни двойки. Что естественно, поскольку второй поток имеет более высокий приоритет. Может показаться, что второй поток полностью забивает первый. Однако это не так, и для того чтобы убедиться в этом, перенаправим в файл информацию, выводимую приложением на консоль. Сделать это можно при помощи следующей команды.

```
Some.exe > Some.txt
```

Прервать работу приложения можно при помощи комбинации **Ctrl-Break**. Не позволяйте приложению работать слишком долго, оно достаточно быстро может исчерпать свободное место на вашем диске, несмотря на свою примитивность. В получившемся файле вы сможете отыскать несколько единиц, хотя их число по сравнению с двойками будет ничтожно мало.

Для того чтобы наглядно увидеть борьбу потоков за процессорное время, установите в предыдущем примере одинаковые приоритеты потоков.

```
th1.Priority = ThreadPriority.Normal;
th2.Priority = ThreadPriority.Normal;
```

После чего можно увидеть борьбу за процессорное время, выражающуюся в игре двоек и единиц на мониторе компьютера.

Фоновые потоки

По умолчанию подразумевается, что работа приложения будет завершена по окончании работы всех его потоков. Нет потоков, нет и приложения. Однако разработчики платформы .NET решили нарушить эту аксиому и создали механизм, который позволяет автоматически завершать работу приложения, даже если не все из его потоков уничтожены. Было решено разбить все потоки на две группы: основные и фоновые. Основными считаются потоки, выполняющие критичные для приложения задачи, без которых его работа невозможна. Если все такие потоки завершают свою работу, то среда исполнения автоматически прекращает его работу. Приложение может содержать несколько фоновых потоков, которые механизмами завершения работы управляемого приложения не учитываются. Они их попросту не будут замечать, независимо от того, работают фоновые потоки или нет. В случае, когда все основные потоки будут уничтожены, приложение автоматически завершит свою работу. Ответственность за определение типа потока полностью лежит на программисте. Именно он обязан при помощи свойства

```
public bool Thread.IsBackground { get; set; }
```

указать, какого типа будет тот или иной поток.

Продемонстрируем работу с фоновыми потоками на простом примере (листинг 9.9). По умолчанию главный поток приложения входит в основную группу потоков. В дополнение к нему создадим фоновый поток, который постоянно, с небольшой задержкой будет выводить на консоль приветствие. Таким образом, у нас будет два потока: один основной, другой фоновый. После окончания работы основного приложения, выход из функции `Main`, работа приложения будет автоматически завершена, несмотря на то, что один из потоков будет еще работать.

Листинг 9.9. Демонстрация создания фоновых потоков

```
/*
  Листинг 9.9
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
```

```
class App
{
  // Специально объявим объект, представляющий поток за пределами
  // функции Main, чтобы он не был уничтожен при выходе основного потока.
  private static Thread th;
  // Функция второго потока приложения.
  public static void ThreadProc()
  {
    // Опишем специальный защищенный блок для того, чтобы
    // отследить завершение потока.
    try
    {
      // Сообщим пользователю о начале работы второго потока.
      Console.WriteLine("Enter to second thread");
      for (;;)
      {
        Console.WriteLine("Hello, World!");
        Thread.Sleep(150);
      }
    }
    // Установим обработчик, отслеживающий завершение потока.
    catch(ThreadAbortException)
    {
      // Сообщим пользователю о завершении работы второго потока.
      Console.WriteLine("Quit from second thread");
    }
  }
  // Точка входа в приложение, являющаяся также функцией второго потока.
  public static void Main()
  {
    // Опишем делегат, указывающий на функцию, представляющую
    // второй поток.
    ThreadStart thStart = new ThreadStart(ThreadProc);
    // Создадим объект, представляющий второй поток.
    th = new Thread(thStart);
    // Запустим второй поток на исполнение.
    th.Start();
    // Сделаем его фоновым.
    th.IsBackground = true;
```

```
    // Подождем, пока второй поток гарантированно приостановит свою
    // работу, а также зайдет в защищенный блок.
    Thread.Sleep(500);
    // Сообщим пользователю о закрытии основного потока
    // приложения.
    Console.WriteLine("Quit from Main Thread");
  }
};
```

В результате работы приложения на консоль были выведены следующие строки.

```
Enter to second thread
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Quit from Main Thread
Hello, World!
```

После чего приложение завершило свою работу.

Если второй поток приложения не делать фоновым, то приложение самостоятельно никогда не завершит своей работы. Оно будет бесконечно выводить на консоль приветствие `"Hello, World!"`.

Такая возможность может оказаться полезной в сложных многопоточных приложениях, в которых ранее приходилось самостоятельно уничтожать ненужные потоки.

9.4. Локальная память потока

Подводные камни в многопоточных приложениях

При переходе от простых однопоточных приложений к многопоточным становится ясно, что прежние методики разработки перестают действовать. Перестают работать, казалось бы, такие очевидные и безотказные механизмы, как статические и глобальные переменные. При их использовании в многопоточных приложениях вы можете натолкнуться на весьма неожиданные неприятности.

Обычно считается, что статические и глобальные переменные самопроизвольно не изменяют своего значения во время работы функций. Рассмотрим следующий пример использования статической переменной.

```
class Some
{
```

```
  static int x;
  void SomeFunction()
  {
    // Здесь подразумевается работа с переменной x.
    x = x + 1;
    for (int i = 0; i < 100; i++)
      x += (i % 2) – i;
    x = 33 % x;
  }
};
```

Подразумевается, что значение переменной `x` будет всегда оставаться постоянным, независимо от того, исполняется в данный момент метод `SomeFunction`, завершил ли он свою работу или приступает к новому этапу своего исполнения. Только он может изменить значение переменной и более никто. Но представим себе, что приложение станет многопоточным и `SomeFunction` будет одновременно вызвано из нескольких потоков. В этом случае поведение работы функции будет непредсказуемо. Пока один поток будет выполнять код где-нибудь в середине функции, другой изменит ее значение в самом начале, а третий обнулит где-нибудь в конце.

Проблема возникает оттого, что переменная `x` определена одна на все потоки. Вот если бы мы имели персональную проекцию переменной для каждого потока, то проблема была бы решена. Оказывается, такая возможность существует. Она предоставлена разработчиками среды .NET в виде атрибута.

```
public class ThreadStaticAttribute : Attribute
```

Если пометить этим атрибутом переменную, то для каждого потока будет создан собственный экземпляр переменной.

Применение атрибута несложно и производится следующим образом.

```
[ThreadStatic]
static int TlsValue;
```

После этого переменная `TlsValue` будет уникальна для любого потока, обратившегося к ней. Изменения, внесенные в переменную из одного потока, никоим образом не будут отражаться на других потоках приложения. Продемонстрируем работу с атрибутом на примере использования общей статической переменной одновременно несколькими потоками (листинг 9.10).

Листинг 9.10. Использование потоконезависимой переменной несколькими потоками

```
/*
  Листинг 9.10
  File:   Some.cs
```

```
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Объявим переменную в качестве локальной для потока.
  [ThreadStatic]
  public static int TlsValue = -1;
  // Функция первого потока.
  public static void ThreadProc1()
  {
    // Выведем на консоль значение переменной TlsValue.
    Console.WriteLine("Thread 1: TlsValue is {0}",TlsValue);
    // Изменим ее значение.
    TlsValue = 1;
  }
  // Функция второго потока.
  public static void ThreadProc2()
  {
    // Выведем на консоль значение переменной TlsValue.
    Console.WriteLine("Thread 2: TlsValue is {0}",TlsValue);
    // Изменим ее значение.
    TlsValue = 2;
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Сразу же изменим значение переменной.
    TlsValue = 6;
    // Создадим делегаты, представляющие функции наших потоков.
    ThreadStart ths1 = new ThreadStart(ThreadProc1);
    ThreadStart ths2 = new ThreadStart(ThreadProc2);
    // Создадим объекты, представляющие потоки.
    Thread th1 = new Thread(ths1);
```

```
    Thread th2 = new Thread(ths2);
    // А теперь по очереди запускаем потоки.
    // Запустим первый поток на исполнение.
    th1.Start();
    // Дождемся завершения первого потока.
    th1.Join();
    // Запустим второй поток на исполнение.
    th2.Start();
    // Дождемся его завершения.
    th2.Join();
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
TlsValue is 0
TlsValue is 0
```

Из лога видно, что переменная была уникальна для каждого из потоков и равнялась нулю, несмотря на то, что каждый из потоков, включая основной, так или иначе пытался изменить ее значение. Правда, из листинга не становится очевидным, что за такое поведение ответственен атрибут `ThreadStatic`. Чтобы продемонстрировать это, избавимся от него.

```
// Намеренно отменим применение атрибута.
// [ThreadStatic]
public static int TlsValue = -1;
```

После чего снова запустим приложение. В результате чего на консоли появятся следующие строки.

```
Thread 1: TlsValue is 6
Thread 2: TlsValue is 1
```

Видно, что значение переменной хаотично меняется различными потоками приложения.

Альтернативный способ работы с *TLS*

Помимо декларативного подхода к работе с `TLS` существует возможность прямого доступа к нему при помощи группы представленных ниже методов.

```
// Данная функция позволяет зарезервировать слот локальной
// памяти потока/
public static LocalDataStoreSlot Thread.AllocateDataSlot();
```

```
// Позволяет получить объект из TLS-памяти
// по ключу, представляющему ячейку.
public static object Thread.GetData (LocalDataStoteSlot slot );
// Позволяет записать объект в заданную ячейку TLS-памяти.
public static void Thread.SetData (LocalDataStoreSlot slot, object data)
// Позволяет зарезервировать именованный TLS слот памяти.
public static LocalDataStoreSlot Thread.AllocateNamedDataSlot( string name );
// Позволяет освободить именованный слот TLS-памяти.
public static void Thread.FreeNamedDataSlot ( string name );
```

Локальная память потока построена по принципу ассоциативной коллекции.

Примечание

Ассоциативная коллекция в отличие от массива предполагает доступ не по целочисленным ключам, а по произвольным. В качестве "индексов" можно использовать объекты произвольного типа, лишь бы они были уникальны. Указав соответствующий объект, вы получите необходимый элемент ассоциативной коллекции. Такие объекты принято называть ключами.

Где ключами являются слоты, резервируемые при помощи функций `AllocateDataSlot` или `AllocateNamedDataSlot`. Сам слот ничего хранить не может, он лишь является ключом для доступа к массиву локальной памяти потока. Указав ключ, вы получите доступ к определенной ячейке памяти потока, причем уникальной для каждого потока (рис. 9.4).

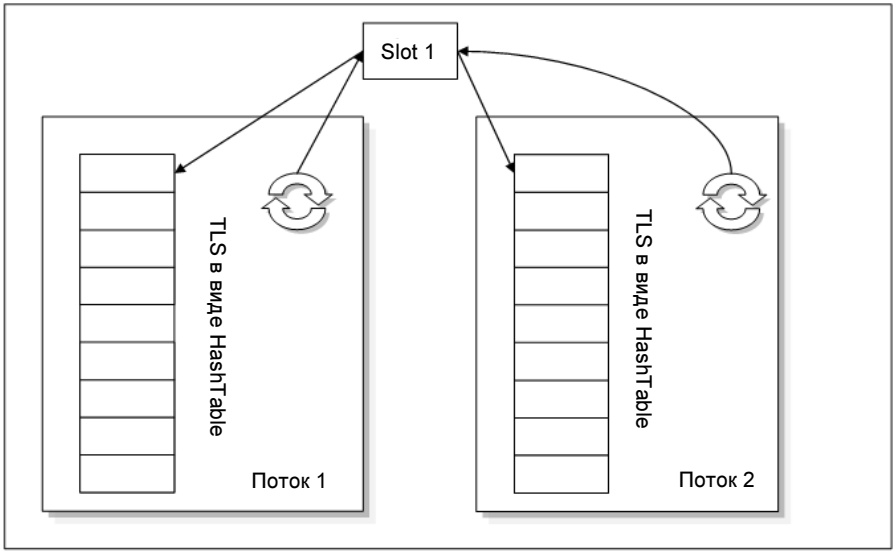


Рис. 9.4. Ассоциативное устройство локальной памяти потока

Обратите внимание на функции.

```
public static object Thread.GetData (LocalDataStoteSlot slot );
public static void Thread.SetData (LocalDataStoreSlot slot, object data)
```

Именно они позволяют получить доступ к локальной памяти потока по ключу, которым в данном случае является слот. Прошу обратить внимание на то, что данные функции определены как статические, то есть возможности чтения локального хранилища произвольных потоков нам не предоставлено. Мы можем обращаться лишь к хранилищу собственного потока, которое в свою очередь будет являться полностью закрытым для всех остальных потоков. Хорошо это или плохо, судить не мне. Но, по крайней мере, могу заявить одно: такой подход несколько увеличивает безопасность приложений. Правда, в какой мере, сказать сложно.

Приведем пример, демонстрирующий работу с локальной памятью потока (листинг 9.11). В главном потоке приложения будет резервировать слот, в который затем будет помещена строка. Затем будет создан вспомогательный поток, который проверит слот и попытается получить доступ к строке. Что, естественно, ему не удастся.

Листинг 9.11. Демонстрация работы с локальной памятью потока

```
/*
  Листинг 9.11
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Это наш слот, который мы зарезервируем в TLS, объявим
  // его глобальным для всего класса, чтобы им можно было
  // воспользоваться из второго потока.
  public static LocalDataStoreSlot slot;
  // Функция второго потока.
  public static void ThreadProc()
```

```
  {
    // Объявим ссылку на строку.
    String str;
    // Установим значение ссылки, хранящейся в слоте.
    // Обратите внимание, что слот у нас один, но значение ссылок,
    // хранящихся в разных потоках, отличаются для одного и того же
    // слота. В данном случае слот выступает в роли ключа для
    // получения ссылки, хранящейся в TLS.
    str = (String)Thread.GetData(slot);
    // Сообщим пользователю значение строки, переданной нам по ссылке,
    // хранящейся в TLS. Поскольку в данном потоке запрашиваемый
    // слот не изменялся, то строка будет пустой.
    if (str == null)
      Console.WriteLine("Thread 2: String is empty");
    else
      Console.WriteLine("Thread 2: String value in second Thread is:↵
{0}", str);
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Объявим строку, ссылку на которую будем хранить в TLS.
    String str = "MainFunction";
    // Создадим делегат, указывающий на функцию нашего потока.
    ThreadStart ths = new ThreadStart(ThreadProc);
    // Создадим объект, представляющий новый поток.
    Thread th = new Thread(ths);
    // Создадим TLS-слот.
    slot = Thread.AllocateDataSlot();
    // Поместим в слот в этом потоке ссылку на ранее
    // объявленную строку.
    Thread.SetData(slot,str);
    // Опишем еще одну ссылку.
    String str2;
    // Получим ссылку, хранящуюся в слоте, зарезервированном нами.
    str2 = (String)Thread.GetData(slot);
    // Выведем на консоль строку, на которую указывает
    // полученная нами ссылка.
```

```
    Console.WriteLine("Str2 from Main thread = {0}",str2);
    // Запустим на исполнение второй поток.
    th.Start();
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
Str2 from Main thread = MainFunction
Thread 2: String is empty
```

9.5. Синхронизация потоков

Ранее мы рассматривали простейшие ситуации, когда несколько потоков в приложении работали каждый независимо друг от друга. Но в реальной жизни дела обстоят несколько сложнее, чаще всего программа представляет собой единый организм, одни потоки которого логически связаны с другими. И для того чтобы скоординировать их работу, были введены специальные механизмы синхронизации потоков. Именно они позволяют согласовать поведение нескольких потоков, для выполнения единых целей.

Среда исполнения предоставляет нам несколько подходов к синхронизации, каждый из которых будет удобен в своем случае. Перечислим предоставляемые сервисы:

- критические секции;
- мьютексы;
- события;
- синхронизированные счетчики;
- синхронизацию записи чтения.

Далее каждый из пунктов будет рассмотрен более подробно.

Критические секции

Критические секции являются простейшим сервисом синхронизации кода. Они позволяют предотвратить одновременное исполнение защищенных участков кода, из различных потоков.

Для начала необходимо показать, для чего это может понадобиться. Приведем простейший пример. Функция `IncrementAndWrite` будет увеличивать значение переменной на единицу, а затем выводить ее значение на консоль.

```
// Тестовый идеализированный класс.
```

```
class Some
{
  // Значение некоторого счетчика.
  private int m_Value;
  // Это конструктор нашего класса.
  public Some()
  {
    // Обнулим счетчик.
    m_Value = 0;
  }
  // Метод увеличивает значение счетчика и сообщает
  // его значение пользователю на консоль.
  public void IncrementAndWrite()
  {
    // Увеличиваем значение счетчика.
    m_Value++;
    // Выводим на консоль его значение.
    Console.WriteLine("Value = {0}",m_Value);
  }
}
```

Казалось бы, все просто и ошибке взяться неоткуда. Однако при работе в многопоточном приложении эта функция может дать сбой. Представьте себе, что первый поток начинает исполнять функцию `IncrementAndWrite`, увеличивает значение переменной `m_Value` и уже собирается выводить ее значение на консоль. Но оказывается, что второй поток начал параллельно с первым исполнять тот же метод, только на несколько квантов позднее. Он также увеличит значение переменной `m_Value`. И что получается? Первый поток выведет на консоль неверное значение переменной.

Программистам, не имеющим опыта отладки многопоточных приложений, этот пример может показаться надуманным, однако заверяю: это совсем не так. В реальных приложениях все гораздо сложнее, поскольку функций там обычно больше и логика гораздо запутаннее. Возвращаясь к нашему примеру, попытаемся сделать его код потокобезопасным. Для того чтобы осуществить это, воспользуемся критической секцией. Защитив при помощи нее код метода `IncrementAndWrite`. Далее представлена новая версия метода.

```
// Тестовый идеализированный класс теперь станет потокобезопасным.
class Some
{
  // Некая переменная-счетчик.
```

```
  private int m_Value;
  // Конструктор нашего класса.
  public Some()
  {
    // Обнулим значение переменной-счетчика перед началом работы.
    m_Value = 0;
  }
  // А это наш метод, увеличивающий значение счетчика и выводящий
  // его значение на консоль. Теперь он будет потокобезопасным.
  public void IncrementAndWrite()
  {
    // Это объявление критической секции, в нее одновременно
    // не смогут войти два потока. Если один из потоков будет
    // находиться внутри критической секции, то другие
    // будут ждать, пока он ее не освободит.
    // Для критических секций обязателен объект ожидания, поэтому
    // я выбрал сам объект, которому принадлежит метод, то есть
    // this
    lock(this)
    {
      // Увеличиваем значение переменной-счетчика.
      m_Value++;
      // Выводим его значение на консоль.
      Console.WriteLine("Value = {0}",m_Value);
    }
  }
}
```

Обратите внимание, что критической секции нужен объект, идентифицирующий ее. Две критические секции, представленные одним объектом, будут идентичны. То есть мы можем описать несколько критических секций в разных частях кода для одного объекта и они будут расцениваться средой исполнения как одна критическая секция. Приведем пример, введя в предыдущий код функцию `DecrementAndWrite`, которая будет защищена той же критической секцией.

```
using System;
// Тестовый идеализированный класс, теперь станет потокобезопасным
class Some
{
  // Некая переменная-счетчик.
```

```
  private int m_Value;
  // Конструктор нашего класса.
  public Some()
  {
    // Обнулим значение переменной-счетчика перед началом работы.
    m_Value = 0;
  }
  // А это метод, увеличивающий значение счетчика и выводящий
  // его значение на консоль. Теперь он будет потокобезопасным.
  public void IncrementAndWrite()
  {
    // Объявление критической секции. В нее одновременно
    // не смогут войти два потока. Если один из потоков будет
    // находиться внутри критической секции, то другие
    // будут ждать, пока он ее освободит.
    // Для критических секций обязателен объект ожидания, поэтому
    // я выбрал сам объект, которому принадлежит этот метод (this).
    lock(this)
    {
      // Увеличиваем значение переменной-счетчика.
      m_Value++;
      // Выводим его значение на консоль.
      Console.WriteLine("Value = {0}",m_Value);
    }
  }
  // Дополнительный метод, защищенный той же критической секцией.
  public void DecrementAndWrite()
  {
    // По смыслу, это та же самая критическая секция,
    // что и предыдущая.
    lock(this)
    {
      // Уменьшаем значение переменной-счетчика.
      m_Value--;
      // Выводим его значение на консоль.
      Console.WriteLine("Value = {0}",m_Value);
    }
  }
}
```

Таким образом, если любой поток захватит одну из критических секций, то другие не смогут попасть ни в первую, ни во вторую секцию. Несмотря на то, что критическая секция разбита на две, она все равно будет считаться средой исполнения за одну.

Для того чтобы объявить критическую секцию, нам пришлось воспользоваться лишь ключевым словом `lock` и не более того. Но за эту простоту мы всецело должны благодарить компилятор C#. Оказывается, что на уровне среды исполнения критические секции организуются при помощи методов

```
public static void Monitor.Enter (object obj);
public static void Monitor.Exit (object obj);
```

Первая открывает критическую секцию, а вторая закрывает ее. А компиляторы высокоуровневых языков попросту транслируют свои ключевые слова в обращения к этим методам.

Для наглядности приведем результат трансляции функции `Monitor` компилятором C#:

```
// Так будет выглядеть код метода после трансляции.
public void IncrementAndWrite()
{
  // Входим в критическую секцию.
  Monitor.Enter( this );
  // Объявляем защищенный блок для того, чтобы гарантировать
  // вызов блока завершения, при помощи которого будем выходить из
  // критической секции.
  try
  {
    // Увеличиваем значение переменной-счетчика.
    m_Value++;
    // Выводим ее значение на консоль.
    Console.WriteLine("Value = {0}",m_Value);
  }
  // Это блок завершения, который будет вызван в любом случае,
  // независимо от того, произошло ранее исключение или нет.
  finally
  {
    // Покидаем критическую секцию, открываем путь в нее для других потоков.
    Monitor.Exit(this);
  }
}
```

Компилятор оборачивает код критической секции в защищенный `try/finally` блок для того, чтобы гарантировать вызов `Monitor.Exit` и обеспечить выход из критической секции. Если бы такой гарантии не предоставлялось, то использование критических секций было бы опасным занятием. Любое исключение, произошедшее в критической секции, тут же приводило бы к возникновению взаимной блокировки.

В критической секции может находиться лишь один поток. Другие потоки, которые попытаются войти в нее, будут блокированы и выставлены в очередь ожидания. Только после выхода потока из критической секции (`Monitor.Exit`) будет запущен следующий в очереди поток (рис. 9.5).

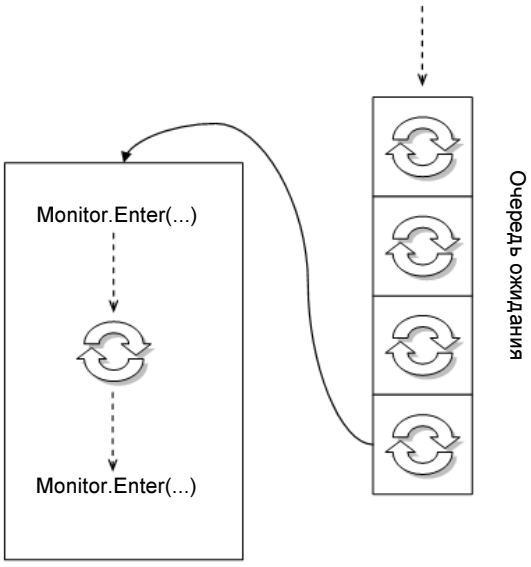


Рис. 9.5. Ожидание потоков на пороге критической секции

Управление блокировками на критических секциях

Итак, если один из потоков попытается войти в критическую секцию, занятую другим потоком, то он будет блокирован. И будет находиться в бездействии, пока не освободится "место под солнцем". Подобное поведение зачастую не только не желательно, а в некоторых, особо критичных к производительности приложениях, вообще не допустимо. Вместо того чтобы простаивать, поток может заняться какой-нибудь полезной работой. Но для этого, прежде всего, необходимо предотвратить его блокирование потока на

входе в критическую секцию. Это может быть осуществлено при помощи следующих методов.

```
// Метод осуществляет попытку входа в критическую секцию без
// дополнительного ожидания. В случае неудачи входа функция
// завершает свою работу и возвращает значение false.
public static bool Monitor.TryEnter(object obj);
// Эта функция позволяет ожидать некоторое время входа
// в критическую секцию. Если за это время вход в секцию не удастся
// осуществить, то функция завершит свою работу, возвратив значение
// false.
public static bool Monitor.TryEnter(object obj, int millisecondsTimeout);
public static bool Monitor.TryEnter(object obj, TimeSpan timeout);
```

Первая функция осуществляет только попытку входа в критическую секцию, и если она не удастся, то она сразу же вернет управление потоку. Другие две функции будут пытаться войти в критическую секцию в течение некоторого времени, если это им не удастся, они также вернут управление.

Сначала приведем пример прямого использования функции `TryEnter`.

```
// Этот метод потокобезопасен, он увеличивает значение
// счетчика переменной и выводит его на консоль, но теперь
// он также возвращает булево значение, определяющее успех его работы.
public bool IncrementAndWrite()
{
  // Попытаемся войти в критическую секцию.
  if (!Monitor.TryEnter(this))
    // Если это не удастся, то сообщим пользователю
    // о неудаче в работе функции.
    return false;
    ...
```

А теперь попробуем получить выгоду от работы функции.

```
// Этот метод потокобезопасен, он увеличивает значение
// счетчика переменной и выводит его на консоль, но теперь
// он также возвращает булево значение, определяющее успех его работы.
public bool IncrementAndWrite()
{
  // Попытаемся войти в критическую секцию.
  while (!Monitor.TryEnter(this))
  {
    // Недолго делаем что-нибудь полезное,
```

```
    // к примеру, посчитаем фракталы.
    DoSomeThingUseful();
  }
  ...
```

Таким образом, во время ожидания метод будет заниматься полезной работой. Но делать это можно только в том случае, если время ожидания входа в критическую секцию не особо важно. Если критическая секция пользуется особой популярностью у потоков приложения, то подобный подход существенно снизит вероятность входа в секцию, даже если сама по себе она достаточно мала.

Для повышения вероятности можно прибегнуть к помощи второго метода, который позволяет задать период ожидания доступа к критической секции.

```
// Этот метод потокобезопасен, он увеличивает значение
// счетчика переменной и выводит его на консоль, но теперь
// он возвращает булево значение, определяющее успех его работы.
public bool IncrementAndWrite()
{
  // Ждем четверть секунды на входе в критическую секцию.
  if (!Monitor.TryEnter(this,250))
  {
    // Недолго делаем что-нибудь полезное,
    // к примеру, посчитаем фракталы.
    DoSomeThingUseful();
  }
  ...
```

Хотя такие подходы могут быть весьма полезны, использовать их надо с крайней осмотрительностью.

Более сложные приемы работы с критическими секциями

Если в ходе работы потока внутри критической секции вдруг выяснится, что имеет смысл пустить в нее следующий поток, но при этом не выходить из нее, это можно осуществить при помощи следующей функции.

```
// Блокирует текущий поток до возникновения заданного события,
// представленного объектом, переданного в качестве первого параметра
// функции.
public static bool Monitor.Wait (object obj);
// Она блокирует поток внутри критической секции до тех пор, пока его
```

```
исполнение не будет возобновлено при помощи метода.
// Освобождает первый поток, ожидающий объект событие.
public static void Monitor.Pulse (object obj);
```

Таким образом, мы имеем возможность повторно блокировать потоки непосредственно внутри критической секции, открывая ее другим потокам.

Для демонстрации работы подхода приведем пример (листинг 9.12). Первый поток, вошедший в критическую секцию, будет блокировать себя, чтобы разрешить второму потоку войти в эту же критическую секцию и изменить данные, с которыми он будет работать.

Листинг 9.12. Самоблокирование потоков внутри критической секции

```
/*
  Листинг 9.12
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Опишем объект, для которого
// будем проводить синхронизацию.
class SyncObject
{
  // Это закрытое значение переменной счетчика.
  private int m_iValue;
  // Конструктор необходим для того, чтобы
  // инициализировать значение внутренней переменной.
  public SyncObject()
  {
    m_iValue = 0;
  }
  // Свойство, позволяющее изменять и получать значение переменной
  // счетчика.
  public int Value
  {
    // Модификатор получения значения свойства.
    get
```

```
    {
      // Возвращаем значение свойства.
      return m_iValue;
    }
    // Модификатор установки значения свойства.
    set
    {
      // Устанавливаем значение свойства.
      m_iValue = value;
    }
  }
};
// Основной класс приложения.
class App
{
  // Объект, по которому будем проводить синхронизацию.
  private static SyncObject m_oSync;
  // Функция второго потока.
  public static void ThreadProc()
  {
    //Создаем критическую секцию для нашего объекта.
    lock(m_oSync)
    {
      // Ожидаем изменений, которые сделает другой поток над
      // объектом. Этой строкой кода
      // мы открываем вход во все критические секции,
      // синхронизирующие код для объекта m_oSync.
      Monitor.Wait(m_oSync);
      // Выводим на консоль значение синхронизируемого объекта.
      Console.WriteLine("m_oSync.Value = {0}",m_oSync.Value);
    }
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Опишем делегат, указывающий на функцию второго потока.
    ThreadStart ths = new ThreadStart (ThreadProc);
    // Создадим объект, представляющий второй поток.
    Thread th = new Thread(ths);
    //Функция создает объект, по которому будет проводиться
```

```
    //синхронизация.
    m_oSync = new SyncObject();
    // Запускаем второй поток на исполнение.
    th.Start();
    //Подождем, пока он дойдет до метода Monitor.Wait.
    Thread.Sleep(100);
    // Несмотря на то, что второй поток находится в идентичной
    // критической секции, мы свободно войдем в эту секцию,
    // так как он нам это разрешил при помощи метода Monitor.Wait.
    lock (m_oSync)
    {
      // Изменяем значение объекта синхронизации.
      m_oSync.Value = 10;
      // Разрешаем выполнение второго потока,
      // сбросив объект синхронизации.
      Monitor.Pulse(m_oSync);
    }
  }
};
```

Результат работы примера будет следующий.

```
m_oSync.Value = 10
```

Понимание механизмов работы примера с непривычки может быть затруднительным. Разберем работу примера по шагам. Сначала он создает дополнительный поток, который входит в критическую секцию, синхронизируемую по объекту `m_oSync`. В это время главный поток приложения блокируется на критической секции, синхронизируемой по этому же объекту. Он будет покорно ожидать, когда критическая секция будет освобождена вторым потоком. Но этого не произойдет, поскольку второй поток блокирует себя в критической секции (`Monitor.Wait`) и позволяет основному потоку приложения войти в нее. Основной поток изменяет значение общей переменной, а затем снимает блокировку со второго потока (`Monitor.Pulse`), сбрасывая объект синхронизации.

9.6. Объекты синхронизации

Для более тонкой синхронизации многопоточных приложений одних критических секций явно недостаточно. Посему в среду исполнения были введены дополнительные объекты синхронизации.

Программисты, имеющие опыт создания многопоточных приложений для Windows, узнают в .NET-объектах простые обертки, реализующие функцио-

нальность классических объектов синхронизаций. В табл. 9.5 приведено сравнение объектов синхронизации Windows и .NET.

Таблица 9.5. *Соответствие объектов синхронизации Windows и .NET*

Объект среды .NET	Объект среды Windows	Группа функций Windows для работы с объектом
`WaitHandle`	Событие	`CreateEvent`, `OpenEvent`, `SetEvent`, `ResetEvent`
`Mutex`	Мьютекс	`CreateMutex`, `OpenMutex`
`ManualResetEvent`	Событие с ручным сбросом	`CreateEvent`, `OpenEvent`, `SetEvent`, `ResetEvent`
`AutoResetEvent`	Событие с автоматическим сбросом	`CreateEvent`, `OpenEvent`, `SetEvent`, `ResetEvent`

Может показаться, что такие объекты являются всего лишь обертками для системных объектов синхронизации Windows. Однако это не совсем так. Нельзя забывать об операционных системах, не поддерживающих данных объектов, под управлением которых может работать платформа .NET. Даже в этом случае Microsoft гарантирует, что предоставляемые объекты будут работать в соответствии с документацией. Фактически, мы получаем собственную независимую систему объектов синхронизации платформы .NET, хотя и довольно похожую на встроенные сервисы синхронизации Windows.

Рассмотрим каждый из объектов синхронизации более подробно.

События *AutoResetEvent* и *ManualResetEvent*

События предназначены для обеспечения взаимодействия между различными потоками. С их помощью один поток может информировать другой о некотором произошедшем событии. Схема работы событий такая: один из потоков создает объект-событие, совершенно неважно какой, затем поток, который хочет получить уведомление о некотором событии, начинает ожидать этого события. Он блокирует себя на этом событии при помощи метода `WaitOne`, предоставляемым объектом события. Любой другой поток приложения может послать ожидающему потоку уведомление о том, что интересующее его событие случилось. Для этого он обращается к объекту-событию, на котором ожидает первый поток, и вызывает его функцию `Set`. Эта функция переводит событие в установленное состояние, вследствие чего блокированный на данном событии поток мгновенно освобождается и продолжает свою работу. Как только поток будет разблокирован, он может справедливо заключить, что произошло событие, которого он ожидал.

При первом знакомстве с событиями, обычно возникает чувство, что это не совсем то, чего ждал от событий — как-то они не так устроены. Тем не менее, они позволяют осуществить возложенные на них задачи, какими бы странными они не казались.

Приведем простейший пример работы события с автоматическим сбросом (листинг 9.13). Главный поток приложения будет создавать событие в занятом состоянии, затем запускать второй поток, который в свою очередь будет ожидать сброса события. После запуска второго потока главный поток приложения будет просить пользователя нажать **Enter** для продолжения, после чего сбросит событие, чем освободит от блокировки второй поток.

Листинг 9.13. Простейший пример использования событий

```
/*
  Листинг 9.13
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  //Это объект событие.
  public static AutoResetEvent are;
  //Функция второго потока.
  public static void ThreadProc()
  {
    //Ожидаем возникновения события.
    are.WaitOne();
    //В этот момент событие уже находится в сброшенном состоянии,
    //так как мы использовали событие с автоматическим сбросом.
    Console.WriteLine("Hello, World, from second Thread!");
  }
  // Точка входа в приложение.
  public static void Main()
  {
```

```
    //Создаем второй поток.
    Thread th = new Thread (new ThreadStart(ThreadProc));
    //Создаем событие в сброшенном состоянии.
    are = new AutoResetEvent(false);
    //Запускаем второй поток на исполнение.
    th.Start();
    // Для продолжения дальнейших действий ожидаем от пользователя
    // нажатия кнопки Enter.
    Console.WriteLine("Press Enter to continue");
    Console.ReadLine();
    // Переводим событие в установленное состояние,
    // освобождая второй поток от блокировки.
    are.Set();
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
Press Enter to continue
Hello, World, from second Thread!
```

После вывода первой строки придется нажать клавишу **Enter**.

Любой объект-событие может находиться в двух состояниях: сброшенном и установленном. Если поток запросит ожидание на сброшенном событии, то он будет блокирован до тех пор, пока событие не будет переведено в установленное состояние. Если поток запросит ожидание на установленном событии, тогда он будет производить свою работу дальше, как будто ничего не случилось.

Начальное состояние события задается при помощи единственного булевого параметра конструктора. Значение `False` предписывает создать событие в сброшенном состоянии, значение `True` — в установленном. Приведем пример, который будет демонстрировать попытку ожидания на установленном событии (листинг 9.14).

Листинг 9.14. Попытка ожидания на сброшенном событии

```
/*
  Листинг 9.14
  File:   Some.cs
  Author: Дубовцев Алексей
*/
```

```
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим объект-событие в установленном состоянии.
    AutoResetEvent are = new AutoResetEvent(true);
    Console.WriteLine("Before waitOne");
    are.WaitOne();
    Console.WriteLine("After waitOne");
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
Before waitOne
After waitOne
```

Вроде бы ничего сверхъестественного, но если попробовать создать событие не в занятом, а в сброшенном состоянии, то приложение навсегда "уйдет в себя".

```
AutoResetEvent are = new AutoResetEvent(false);
```

Для работы с состояниями событий предназначены два метода.

```
// Функция устанавливает событие в сброшенное состояние.
public bool WaitHandle.Set();
// Эта функция устанавливает событие в занятое состояние.
public bool WaitHandle.Reset();
```

Первый переводит событие в сброшенное состояние, второй переводит в занятое состояние. Эти методы описаны в классе `WaitHandle`, который является базовым для классов `AutoResetEvent`, `ManualResetEvent`, `Mutex`. Таким образом, возможно переводить все вышеперечисленные объекты в любое из указанных состояний.

В среде .NET представлено два вида событий: с ручным сбросом и с автоматическим сбросом. События с автоматическим сбросом переводятся в свободное

состояние сразу же после того, как освобождается один из потоков, ожидающих его. Это позволяет блокировать остальные потоки, ожидающие на событии. Таким образом, если события ожидают несколько потоков, то при сбросе события будет освобожден только один из них, после чего событие опять будет переведено в занятое состояние, а остальные потоки будут ждать дальше.

Приведем простейший пример (листинг 9.15). В нем три потока будут блокированы при ожидании на одном и том же событии с автоматическим сбросом.

Листинг 9.15. Работа событий с автоматическим сбросом

```
/*
  Листинг 9.15
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Событие синхронизации.
  private static AutoResetEvent arEvent;
  // Переменная статическая понадобится для подсчета
  // номера потока, и при создании любого
  // потока ее значение будет увеличиваться на единицу.
  private static int ThreadNumber = 1;
  // Функция потока.
  public static void ThreadProc()
  {
    // Эта локальная переменная будет равняться номеру потока
    // значением от 1 и выше.
    int l_TN = ThreadNumber++;
    // Сообщаем пользователю о запуске еще одного потока.
    Console.WriteLine("Starting another thread");
    // Ожидаем события синхронизации.
```

```
    arEvent.WaitOne();
    // Выводим номер нашего потока.
    Console.WriteLine("Thread number is = {0}",l_TN);
    // Показывает, что мы выполняем код после блокирующей функции.
    Console.WriteLine("This line wrote after arEvent.WaitOne()");
    // Переводим событие в свободное состояние, поскольку
    // при выходе из функции WaitOne событие автоматически
    // было сброшено и
    // поэтому его придется установить заново.
    // arEvent.Set();
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Создаем делегат для функции потока.
    ThreadStart ths = new ThreadStart(ThreadProc);
    // Создаем первый поток.
    Thread th = new Thread(ths);
    // Создаем второй поток.
    Thread th2 = new Thread(ths);
    // Создаем третий поток.
    Thread th3 = new Thread(ths);
    // Создаем событие синхронизации в занятом состоянии.
    arEvent = new AutoResetEvent(false);
    // Запускаем первый поток на исполнение.
    th.Start();
    // Запускаем второй поток на исполнение.
    th2.Start();
    // Запускаем третий поток на исполнение.
    th3.Start();
    // Немного приостанавливаем программу для того, чтобы
    // убедиться, что все потоки успели вызвать функцию
    // ожидания события WaitOne.
    Thread.Sleep(500);
    // Показывает, что сейчас будет установлено событие.
    Console.WriteLine("Write Line after Sleep");
    // Устанавливаем событие.
    arEvent.Set();
  }
};
```

В результате работы примера на консоль будут выведены следующие строки.

```
Starting another thread
Starting another thread
Starting another thread
Write Line after Sleep
Thread number is = 2
This line wrote after arEvent.WaitOne()
```

После чего приложение было намертво блокировано из-за ожидания двух оставшихся потоков на событии `arEvent`.

Теперь откомментируем одну важную строку в конце метода `ThreadProc`.

```
Было.
// arEvent.Set();
Стало.
arEvent.Set();
```

Здесь вручную устанавливается событие, на котором ожидают остальные потоки. Таким образом, они все должны вытянуть друг друга по цепочке. И действительно, после модификации листинга результат работы программы будет следующим.

```
Starting another thread
Starting another thread
Starting another thread
Write Line after Sleep
Thread number is = 1
This line wrote after arEvent.WaitOne()
Thread number is = 2
This line wrote after arEvent.WaitOne()
Thread number is = 3
This line wrote after arEvent.WaitOne()
```

В итоге, для того чтобы достигнуть желаемого результата, пришлось самостоятельно устанавливать события каждый раз по выходу из функции `WaitOne`.

События с ручным сбросом устроены несколько иначе: состояние такого события может быть изменено только самим программистом. Если перевести подобное событие в установленное состояние, когда его ожидают несколько потоков, то все они будут освобождены без изменения состояния события. Модифицируем предыдущий пример, используя событие с ручным сбросом (листинг 9.16). Теперь больше не нужно повторно устанавливать событие, все будет работать и так.

Листинг 9.16. Работа событий с ручным сбросом

```
/*
  Листинг 9.16
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  //Событие синхронизации.
  private static ManualResetEvent arEvent;
  // Переменная статическая, которая понадобится для подсчета
  // номера потока, и при создании любого
  //потока ее значение будет увеличиваться на единицу.
  private static int ThreadNumber = 1;
  //Функция потока.
  public static void ThreadProc()
  {
    //Эта локальная переменная будет равняться номеру потока
    //от 1 и до упора.
    int l_TN = ThreadNumber++;
    //Показываем запуск еще одного потока.
    Console.WriteLine("Starting another thread");
    //Ожидаем события синхронизации.
    arEvent.WaitOne();
    //Выводим номер нашего потока.
    Console.WriteLine("Thread number is = {0}",l_TN);
    //Показывает, что выполняется код после блокирующей функции.
    Console.WriteLine("This line wrote after arEvent.WaitOne()");
    // Обратите внимание, что никакой повторной установки события
    // здесь нет.
  }
  //Главная функция примера.
```

```
  public static void Main()
  {
    //Создаем делегат для функции потока.
    ThreadStart ths = new ThreadStart(ThreadProc);
    //Создаем первый поток.
    Thread th = new Thread(ths);
    //Создаем второй поток.
    Thread th2 = new Thread(ths);
    //Создаем третий поток.
    Thread th3 = new Thread(ths);
    //Создаем событие синхронизации в занятом состоянии.
    arEvent = new ManualResetEvent(false);
    //Запускаем первый поток.
    th.Start();
    //Запускаем второй поток.
    th2.Start();
    //Запускаем третий поток.
    th3.Start();
    //Немного приостанавливаем программу для того, чтобы
    //убедиться в том, что оба потока успели вызвать функцию
    //ожидания события WaitOne.
    Thread.Sleep(500);
    //Показывает, что сейчас будет установлено событие.
    Console.WriteLine("Write Line after Sleep");
    //Устанавливаем событие.
    arEvent.Set();
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
Starting another thread
Starting another thread
Starting another thread
Write Line after Sleep
Thread number is = 1
This line wrote after arEvent.WaitOne()
Thread number is = 2
This line wrote after arEvent.WaitOne()
```

```
Thread number is = 3
This line wrote after arEvent.WaitOne()
```

При использовании событий с ручным сбросом нельзя забывать, что вся ответственность за переключение их состояний полностью лежит на вас. Перевод таких событий в нужное состояние необходимо делать самостоятельно в ручном режиме.

Мьютекс

Мьютексы предназначены для ограничения доступа к ресурсу со стороны нескольких потоков. Только поток, захвативший мьютекс, будет работать с ресурсом, другие же будут ожидать, пока тот не освободит мьютекс.

Примечание

Когда впервые сталкиваешься с мьютексами, очень удивляет странное имя. Даже в словаре не находится упоминания о них. И действительно, на уровне ядра такие объекты называются Мутантами (Mutant). А в пользовательском режиме они вдруг переименовываются в мьютексы. Впрочем, по официальной версии имя Mutex происходит от Mutual Exclusion (Взаимное Исключение).

Фактически, мьютекс является кровным родственником критической секции. За тем исключением, что он производит контроль над потоками, которые захватывают его, а также позволяет проводить синхронизацию между потоками даже в различных приложениях.

Прежде чем работать с мьютексом, его необходимо создать. Для этих целей служат четыре его конструктора.

```
// Позволяет создать мьютекс, находящийся в свободном состоянии.
public Mutex();
// Позволяет создать мьютекс, находящийся в указанном состоянии.
public Mutex(bool initallyOwned);
// Позволяют создать именованные мьютексы в указанном состоянии.
public Mutex(bool initallyOwned, string name );
public Mutex(bool initallyOwned, string name, out bool createdNew );
```

Особо интересными являются два последних метода, позволяющих создать именованные мьютексы, при помощи которых можно проводить синхронизацию на уровне всей системы. Обсудим процессы захвата и освобождения мьютексов. Захват происходит точно так же, как и для обычных событий при помощи метода `WaitOne`. С тем отличием, что если потоку удалось захватить мьютекс, то его исполнение не блокируется, а продолжается, как при использовании критических секций. А вот с освобождением мьютексов есть нюанс, и заключается он в том, что освободить мьютекс может только тот поток, который его захватил. Таким образом, налицо монопольное

владение мьютексом со стороны потока, его захватившего. Если вдруг какой-либо другой поток попытается освободить не принадлежащий ему мьютекс, будет выброшено исключение `System.ApplicationException` с сообщением "Attempt to release mutex not owned by caller", гласящим: "попытка освобождения мьютекса не его владельцем". Также у мьютексов есть еще одна интересная особенность — внутренний счетчик попыток захвата. После того как поток захватил мьютекс, все повторные попытки будут регистрироваться внутри мьютекса и впоследствии он потребует, чтобы его столько же раз освободили, прежде чем он перейдет в свободное состояние.

Приведем наглядный пример работы с мьютексами (листинг 9.17). Будем считать, что мьютекс в нашем примере охраняет сервисы вывода на консоль. Главный поток приложения создаст дополнительный поток, который попытается получить разрешение у мьютекса, то есть захватить его. Но к этому моменту он уже будет занят главным потоком приложения, который освободит его только после того, как сам выведет свое сообщение на консоль.

Листинг 9.17. Простейший пример работы с мьютексами

```
/*
  Листинг 9.17
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Это мьютекс, синхронизирующий наши потоки.
  private static Mutex mut;
  // Функция второго потока.
  public static void ThreadProc()
  {
    // Пытаемся захватить мьютекс.
    mut.WaitOne();
    // Информируем о том, что второй поток запущен
    // и мьютекс успешно захвачен.
    Console.WriteLine("Hello, from Second thread!");
```

```
    // Освобождаем мьютекс.
    mut.ReleaseMutex();
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Создаем мьютекс в свободном состоянии.
    mut = new Mutex();
    // Создаем объект-поток.
    Thread th = new Thread( new ThreadStart(ThreadProc));
    // Захватываем мьютекс.
      mut.WaitOne();
    //Запускаем второй поток на исполнение.
    th.Start();
    // Ждем, чтобы убедиться в том, что второй поток будет
    // ожидать мьютекс, пока мы его не освободим.
    Thread.Sleep(500);
    // Информируем пользователя о захвате мьютекса
    // основным потоком приложения.
    Console.WriteLine("Hello, from Main thread!");
    // Освобождаем мьютекс.
    mut.ReleaseMutex();
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
Hello, from Main thread!
Hello, from Second thread!
```

Приведем пример, демонстрирующий множественный захват мьютекса одним из потоков приложения (листинг 9.18). Он дважды обратится к функции `WaitOne` для того, чтобы захватить мьютекс. Затем один раз освободит его и будет ожидать завершения работы второго потока, блокированного на этом мьютексе.

Листинг 9.18. Множественный захват мьютекса одним потоком

```
/*
  Листинг 9.18
  File:   Some.cs
```

```
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  //Это мьютекс, синхронизирующий наши потоки.
  private static Mutex mut;
  //Функция второго потока.
  public static void ThreadProc()
  {
    // Пытаемся захватить мьютекс.
    mut.WaitOne();
    // Информируем о том, что второй поток запущен
    // и мьютекс успешно захвачен.
    Console.WriteLine("Hello, from Second thread!");
    // Освобождаем мьютекс.
    mut.ReleaseMutex();
  }
  // Точка входа в приложение.
  public static void Main()
  {
    //Создаем мьютекс в свободном состоянии.
    mut = new Mutex();
    //Создаем объект-поток.
    Thread th = new Thread(new ThreadStart(ThreadProc));
    // Захватываем мьютекс.
    mut.WaitOne();
    // Повторно захватываем мьютекс.
      mut.WaitOne();
    // Запускаем второй поток.
      th.Start();
    // Информируем пользователя о захвате мьютекса
    // основным потоком приложения.
    Console.WriteLine("Hello, from Main thread!");
```

```
    //Освобождаем мьютекс.
    mut.ReleaseMutex();
    // Ожидаем завершения второго потока.
    th.Join();
  }
};
```

В результате запуска приложения на консоль будет выведена строка

```
Hello, from Second thread!
```

После чего возникнет взаимная блокировка на мьютексе `mut`, в результате которой приложение "свиснет".

Синхронизация нескольких приложений с помощью мьютексов

Для того, чтобы синхронизировать потоки, находящиеся в различных приложениях, придется прибегнуть к помощи именованных мьютексов. Их можно создать при помощи следующих конструкторов.

```
public Mutex(bool initallyOwned, string name);
public Mutex(bool initallyOwned, string name, out bool createdNew);
```

Здесь в качестве второго параметра необходимо указать уникальное имя мьютекса. Уникальные имена мьютексов необходимы для того, чтобы их было легче передать другому приложению. Так, создав в одном приложении мьютекс с некоторым именем, мы просто откроем его в другом по этому имени и будем использовать. Иначе пришлось бы обратиться к сервисам .NET Remoting, отвечающим за удаленную передачу данных между приложениями, которые обеспечили бы передачу мьютекса искомому приложению.

При создании мьютекса особое внимание следует обратить на его имя — оно обязательно должно быть уникальным в рамках всей системы, иначе не миновать возникновения проблем. Если имя вашего мьютекса совпадет с любым другим существующим в системе, то произойдет коллизия и поведение приложения будет непредсказуемым. Причем при выборе имени мьютекса необходимо гарантировать, чтобы оно было уникальным не только на стадии разработки проекта, но также и во время всей жизни вашего приложения. Фактически, вы обязаны гарантировать, чтобы имя мьютекса было всегда и везде уникально, вне зависимости от системы и установленного на ней дополнительного программного обеспечения. Осуществить эту, казалось бы, нереальную задачу не так уж и сложно. Для этого всего лишь следует формировать имя мьютекса по следующему правилу

```
ИмяКомпании.ИмяПрограммы.UUID
```

Где UUID — это универсальный уникальный 128-битный идентификатор (Universally Unique Identifier). Вы наверняка встречались с ними, обычно они записываются в следующем виде {17eb85ef-0d43-490c-af9a-4a6ef4a4b42b}.

Эти идентификаторы уникальными называют неспроста. Вся их природа заключается в том, что вероятность случайно создать два одинаковых идентификатора практически равна нулю.

Примечание

При создании уникального идентификатора учитывается время с точностью до миллисекунд, серийный номер сетевой карты, а также некоторое случайное число. UUID разработан в рамках программы по распределенным компьютерным вычислениям для идентификации объектов в едином сетевом пространстве.

Если вам стало страшно при мысли о том, как вы будете создавать свой собственный UUID, не бойтесь, вам для этого не придется использовать ни одной страшной формулы или хитроумного алгоритма, нужно будет лишь запустить утилиту

`uuidgen.exe`,

которая входит в стандартную поставку Visual Studio.

Приведем комплексный пример, состоящий из двух приложений, который будет демонстрировать их синхронизацию при помощи мьютексов (листинг 9.19). Ниже представлен код первого приложения. Оно будет создавать именованный мьютекс, а затем ждать от пользователя нажатия **Enter**, после чего изменит состояние созданного ранее мьютекса.

Листинг 9.19. Серверная часть приложения, демонстрирующего межпроцессную синхронизацию

```
/*
  Листинг 9.19
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
```

```
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим свободный именованный мьютекс.
    Mutex mut = new Mutex(false, "Lexa.MutexSample.{f8592579-26b5-4a31-
866e-a4e9202ae682}" );
    // Захватим мьютекс, переведя его тем самым в занятое состояние.
    mut.WaitOne();
    // Сообщим пользователю о том, что ему предлагается нажать
    // любую кнопку.
    Console.WriteLine("Press any key to release mutex close second appli-
cation");
    // Ожидаем ввода пользователя.
    Console.Read();
    // Освобождаем мьютекс.
    mut.ReleaseMutex();
    // Сообщаем пользователю о том, что мьютекс освобожден.
    Console.WriteLine("Mutex released");
    // Предскажем закрытие второго приложения.
    Console.WriteLine("Second application closed");
    // А это приложение будет ожидать бесконечно долго.
    Thread.Sleep(Timeout.Infinite);
  }
};
```

В листинге 9.20 приведен исходный код второй части примера — приложения, которое будет откликаться на изменение состояния мьютекса, созданного первым приложением. После освобождения мьютекса оно будет выводить приветственное сообщение на консоль, а затем завершать свою работу. Обратите внимание на конструктор, при помощи которого создается мьютекс.

```
public Mutex(bool initallyOwned, string name, out bool createdNew );
```

Это специальный конструктор, принимающий дополнительный булевый параметр, который позволяет узнать, был ли создан новый мьютекс или открыт уже существующий. Значимость данного параметра ни в коем случае нельзя недооценивать, за ним необходимо пристально следить. Потому как если мы вдруг создадим собственный мьютекс, а не откроем уже существующий — это будет означать, что основное приложение-сервер еще не запущено. А работа данного возможна только под управлением серверного.

Листинг 9.20. Клиентская часть приложения, демонстрирующего межпроцессную синхронизацию

```
/*
  Листинг 9.20
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Эта переменная позволит нам узнать, был ли ранее
    // создан необходимый мьютекс.
    bool bIsRun;
    // Создадим или точнее попытаемся открыть ранее созданный
    // мьютекс.
    Mutex mut = new Mutex( false,
 "Lexa.MutexSample.{f8592579-26b5-4a31-866e-a4e9202ae682}", out bIsRun );
    // Проверим, не создали ли мы новый мьютекс.
    if(bIsRun)
    {
      // Да, создали. Значит, первое приложение еще не
      // запущено, сообщим об этом пользователю и завершим
      // работу.
      Console.WriteLine("First application does not run");
      return;
    }
    // Сообщим пользователю о том, что сейчас будем
    // захватывать мьютекс.
    Console.WriteLine("Before mut.WaitOne()");
    // Попытаемся захватить мьютекс.
```

```
    mut.WaitOne();
    // Сообщим пользователю о том, что мы успешно захватили
    // мьютекс.
    Console.WriteLine("After mut.WaitOne()");
    // Ждем одну секунду и выходим.
    Thread.Sleep(1000);
  }
};
```

Работать с приложениями нужно следующим образом: первым делом необходимо запустить первое приложение, в отдельном консольном окне, затем запустить второе приложение. После чего снова переключиться в первое приложение и нажать в нем клавишу **Enter**, в результате чего второе приложение выведет на консоль фразу "After mut.WaitOne()", подождет одну секунду, а затем завершит свою работу.

Зачем, спрашивается, нужно было выставлять секундное ожидание в самом конце работы клиентского приложения? Сначала может показаться, что это сделано для того, чтобы пользователь успел увидеть сообщение, выводимое на консоль. Однако цель куда интереснее, чем может показаться с первого взгляда. Если взглянуть на код клиентского приложения более пристально, то можно увидеть, что оно монопольно захватывает мьютекс и возвращает его только по закрытию главного потока приложения. Для того чтобы наглядно убедиться в этом, запустите приложение-сервер и несколько копий клиентской программы. После чего освободите мьютекс из сервера нажатием клавиши **Enter**, и вы увидите, как копии второго приложения закрываются одна за другой с интервалом в одну секунду. Каждая копия второго приложения попытается захватить мьютекс, но поскольку их много, а мьютекс только один, среда исполнения выстроит их в очередь. И когда сервер первый раз освободит мьютекс, он достанется первому приложению, пытавшемуся захватить мьютекс. Затем оно отработает и автоматически освободит мьютекс, который перейдет во владение следующей программе. И так далее, пока все приложения не буду закрыты.

Серверное приложение при запуске создает мьютекс в свободном состоянии, затем захватывает его и приглашает пользователя нажать любую клавишу для того, чтобы освободить мьютекс и закрыть второе приложение. Клиентское же приложение первым делом создает мьютекс, используя конструктор с тремя параметрами, для того чтобы проверить, был ли создан новый мьютекс или открыт уже существующий. Если оно (приложение) делает вывод о том, что был создан новый мьютекс, а не открыт уже существующий, созданный серверным приложением, то он извещает пользователя о том, что для нормальной работы необходимо запустить первое приложение. Если же третий параметр конструктора

```
public Mutex(bool initallyOwned, string name, out bool createdNew );
```

равен false, то это означает, что был открыт уже существующий мьютекс, а не создан новый. В этом случае второе приложение начинает ожидание освобождения мьютекса при помощи функции `WaitOne`, расположенной в базовом классе `WaitHandle`. Как только в серверном приложении пользователь нажмет клавишу **Enter**, мьютекс будет освобожден и клиентское приложение тут же захватит его, о чем уведомит пользователя и завершит свою работу, снова освободив мьютекс.

Напоследок расскажу вам еще об одном применении мьютексов. Их очень удобно использовать, когда хочется ограничить количество открытых копий вашего приложения. Для этого при создании мьютекса необходимо проверять, был ли он создан или же открыл существующий. Из второго случая автоматически вытекает, что приложение было запущено повторно. Приведем пример такой программы (листинг 9.21).

Листинг 9.21. Ограничение количества запущенных копий приложения на основе мьютексов

```
/*
  Листинг 9.21
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Эта переменная будет сигнализировать о второй запущенной
    // копии приложения.
    bool bIsSecondCopy;
    // Попытаемся создать новый мьютекс.
    Mutex mt = new Mutex(false, "OneCopy.{cf7052f5-b1c3-4ddc-97c3-
99ac1a40c7de}", out bIsSecondCopy);
    // Проверим, создали ли мы мьютекс или открыли.
    if (!bIsSecondCopy)
```

```
    {
      // Мьютекс уже существует, значит придется
      // выходить.
      Console.WriteLine("Sorry only one copy of application can working↵
at time");
      return;
    }
    // А здесь приложение должно что-либо делать.
    Console.WriteLine("Emulation of doing some work");
    Thread.Sleep(Timeout.Infinite);
  }
};
```

При попытке запустить более одной копии приложения, вы неминуемо потерпите неудачу.

Синхронизация счетчиков

При написании сверхнадежных многопоточных приложений, многие из потоков интенсивно проходят через одни и те же участки кода, поэтому в них рекомендуется использовать специальные сервисы работы с переменными-счетчиками.

Может показаться невероятным, что на конструкциях подобной данной могут возникнуть какие-либо проблемы:

```
i++
```

Казалось бы, это атомарная операция, которой не страшны многопоточные коллизии. Однако, как оказывается, данная операция является далеко не атомарной, она транслируется компилятором примерно в следующую группу инструкций.

```
// Загрузить значение локальной переменной.
ldloc.0
// Загрузить единицу.
ldc.i4.1
// Сложить.
add
// Установить значение локальной переменной.
stloc.0
```

Которые, в свою очередь, транслируются JIT в еще большее количество родных для платформы инструкций. Таким образом, при одновременном

исполнении этой конструкции в нескольких потоках могут возникнуть непредвиденные коллизии.

Для того чтобы избежать этого, следует использовать следующие функции при работе с переменными-счетчиками.

```
// Сравнивает два целочисленных значения.
public static int Interlocked.CompareExchange (
  ref int location1,
  int value,
  int comparand
);
// Уменьшает целочисленное значение на единицу.
public static int Interlocked.Decrement (
  ref int location
);
// Увеличивает целочисленное значение на единицу
public static int Interlocked.Increment (
  ref int location
);
// Меняет местами два целочисленных значения.
public static int Exchange (
  ref int location1,
  int value
);
```

Здесь представлены все необходимые операции, на основе которых можно создать собственные сколь угодно сложные:

- ❒ обмен со сравнением;
- ❒ увеличение на единицу;
- ❒ уменьшение на единицу;
- ❒ обмен значениями.

Отдельного пояснения здесь требует только метод `CompareExchange` — он производит обмен значениями только в том случае, если значение первого параметра (`location1`) будет совпадать со значением последнего (`comparand`). Использование остальных методов тривиально.

Так же хотелось бы отметить, что у методов `CompareExchange` и `Exchange` существуют модификации, позволяющие оперировать с произвольными объектами, а не только с целочисленными значениями.

```
public static object Interlocked.Exchange (
  ref object location1,
```

```
  object value
);
public static object Interlocked.CompareExchange (
  ref object location1,
  object value,
  object comparand
);
```

При работе с произвольными объектами метод `CompareExchange` для определения равенства объектов будет использовать метод `Equals`, предоставляемый классом `Object`.

Внимательные читатели могут заметить, что синхронизация переменных-счетчиков также возможна при помощи критических секций. Приведем пример, наглядно демонстрирующий разницу данных подходов.

```
// Это первый вариант, он использует критические секции.
class SomeClass
{
  // Это закрытая переменная-счетчик.
  private int m_iVar;
  // Функция потока
  public void ThreadInc()
  {
    // Объявим критическую секцию, которая защитит нашу переменную
    // от одновременного посягательства нескольких потоков.
    lock(this)
      m_iVar++;
   }
...
// А это второй вариант, использующий специализированные сервисы
// среды .NET
class SomeClass
{
  // Это закрытая переменная-счетчик.
  private int m_iVar;
  // Функция потока.
  public void ThreadInc()
  {
    // Увеличим значение счетчика-переменной. При этом
    // гарантируется, что это действие будет защищено в смысле
```

```
    // многопоточности.
Interlocked.Increment(ref m_iVar);
  }
...
```

Какой из подходов использовать, решать вам. Но в заключение я хотел бы отметить, что даже если вы не будете синхронизировать переменные-счетчики в многопоточных приложениях, то, возможно, беды и не произойдет. Но вот если коллизия случиться, то обнаружить ее будет ой как не просто.

9.7. Синхронизация операций ввода/вывода

Ни для кого не является секретом, что взаимодействие программ с внешним миром производится посредством операций ввода/вывода. Будь то операции вывода на консоль, записи в файл или же осуществление запроса по сети. В современных многопоточных приложениях необходимо позаботиться о том, чтобы не дай бог не произошло коллизий во время совершения этих операций. Особенно опасен случай работы нескольких потоков с одним источником данных. Если приложение будет бездумно, да к тому же интенсивно записывать и считывать данные из источника, то беды не избежать. Поэтому в современных приложениях рекомендуется использовать блокировку по принципу одиночная запись/множественное чтение. Такой подход предполагает, что в один момент может производиться лишь одна операция записи или несколько операций чтения, но перекрываться во времени они никоим образом не могут (рис. 9.6).

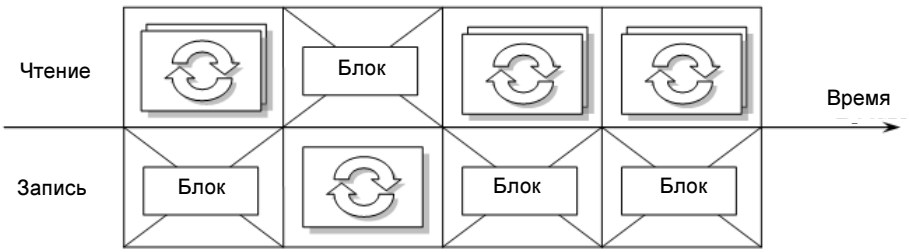


Рис. 9.6. Синхронизация операций ввода/вывода

С первого взгляда можно усмотреть несправедливость, почему одновременно разрешена только одна операция записи, против нескольких операций чтения. Но если обратиться к сути вопроса, то все становится предельно ясно. Если в источник одновременно начнут записывать несколько клиентов, то возникнет сумбур и в итоге конечный вид данных предсказать невозможно. А вот читать из источника одновременно могут сколько угодно клиентов, учитывая,

что все операции чтения на это время запрещены, а следовательно, на время чтения все данные будут абсолютно статичны.

Примечание

В высокопроизводительных системах иногда используются механизмы, позволяющие нескольким клиентам одновременно записывать данные в источник. При работе с подобной системой клиент должен изначально указать, какая область источника ему требуется. А система контроля коллизий либо предоставит ему доступ к этому участку, либо заблокирует, если он уже занят другим клиентом. Здесь подобные системы рассматриваться не будут.

Для реализации такой схемы работы с источниками данных можно воспользоваться описанными выше средствами синхронизации потоков, что наверняка будет не так уж и затруднительно. Но Microsoft заранее позаботилась о нас и включила в стандартную поставку специальный сервис, позволяющий осуществить задуманное. Он представлен классом.

```
public sealed class ReaderWriterLock
```

Он осуществляет синхронизацию чтения/записи для произвольного источника любого типа и при этом удовлетворяет следующим требованиям.

- Быстрая работа с большими потоками данных.
- Оптимизированный механизм балансировки операций чтения/записи. Класс старается усреднить время ожидания операции, избегая долгих блокировок отдельных операций.
- Поддержка таймаутов, что может быть полезно для предотвращения взаимоисключающих блокировок.
- Возможность кеширования объектов синхронизации. Это позволяет уменьшить использование системных ресурсов, путем использования одних и тех же объектов синхронизации для разных потоков.
- Поддержка вложенных блокировок операций чтения/записи.
- Поддержка спин-счетчиков и блокировок для предотвращения переключений контекста на мультипроцессорных системах.
- Поддержка восстановления после непредвиденных сбоев, вроде проблем создания событий синхронизации и т. п.

Сколь бы страшным не казался данный класс, для работы с ним нам придется использовать всего лишь четыре его метода. Все остальное он сделает за нас полностью автоматически. Нам понадобятся два метода, осуществляющие запрос на операции чтения и записи.

```
// Запрашивает блокировку чтения.
public void ReaderWriterLock.AcquireReaderLock (
  // Время ожидания блокировки.
```

```
  int millisecondsTimeout
);
// Запрашивает блокировку записи.
public void ReaderWriterLock.AcquireWriterLock (
  // Время ожидания блокировки.
  int millisecondsTimeout
);
```

Данные методы запрашивают блокировки чтения и записи, которые будут работать в соответствии с алгоритмом, описанным ранее.

Помимо этого нам понадобятся методы, снимающие захваченные блокировки:

```
// Освобождает блокировку чтения.
public void ReaderWriterLock.ReleaseReaderLock();
// Освобождает блокировку записи.
public void ReaderWriterLock.ReleaseWriterLock();
```

Для наглядности приведем пример синхронизированного доступа к ресурсу при помощи класса `ReaderWriteLock`. Первым делом создадим класс `Resource`, представляющий некий гипотетический ресурс. Он будет предоставлять две классические операции — чтения и записи. Обе они будут устроены приблизительно одинаково, перечислим производимые операции по пунктам.

1. Производится захват соответствующей блокировки.
2. Выводится сообщение, уведомляющее о начале операции.
3. После этого создается полусекундная задержка для эмуляции работы операции.
4. Выводится сообщение об окончании операции.
5. Освобождается захваченная блокировка.

Прошу обратить внимание, что первый пункт может быть выполнен удачно далеко не всегда, в некоторых случаях потоки будут ожидать, пока блокировка не освободится.

Для того чтобы продемонстрировать полноценную работу с ресурсом, создадим двадцать потоков (листинг 9.22). Где четные потоки будут записывать в ресурс, а нечетные читать из него. Для того чтобы создать такое внушительное количество потоков, воспользуемся хитрым сервисом — пулом потоков. Более подробно о нем будет рассказано далее, а сейчас вам лишь достаточно знать, что он позволяет зарегистрировать в себе большое количество функций потоков. После чего он будет последовательно выбирать их из внутренней очереди, создавая для каждого из них собственный поток, причем все время, поддерживая постоянное количество потоков. По умолчанию пул может выдержать 25 потоков.

Листинг 9.22. Синхронизация операций чтения/записи при помощи встроенных сервисов

```
/*
  Листинг 9.22
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Это некоторый класс, который является источником
// данных и предоставляет операции чтения/записи.
class Resource
{
  // Встроенный объект синхронизации операций чтения/записи.
  private ReaderWriterLock m_rwl = new ReaderWriterLock();
  // Операция чтения является несколько абстрактной
  // и не позволяет получить данные, а лишь имитирует их чтение.
  // В качестве параметра принимает номер потока, который инициирует
  // эту операцию.
  public void Read(int threadNumber)
  {
    // Запрашиваем блокировку для чтения, при этом собираемся
    // ждать столько, сколько нужно.
    // (Infinite)
    m_rwl.AcquireReaderLock(Timeout.Infinite);
    // Объявим защищенный блок, для того чтобы гарантировать
    // освобождение блокировки чтения.
    try
    {
      // Сообщаем о начале операции чтения данных, при этом
      // сообщаем номер потока, запросившего операцию чтения
      Console.WriteLine("Start -  resource |reading| for thread {0}",
threadNumber);
      //  Эмулируем чтение данных при помощи задержки.
      Thread.Sleep(500);
```

```
      // Сообщаем о завершении операции чтения данных.
      Console.WriteLine("Stop  -  resource |reading| for thread {0}",
threadNumber);
    }
    // Блок завершения будет выполнен всегда.
    finally
    {
      // Освобождаем блокировку чтения.
      m_rwl.ReleaseReaderLock();
    }
  }
  // Операция записи, так же как и операция чтения, является
  // абстрактной и не позволяет осуществить какую бы то ни было
  // запись, а лишь имитирует ее при помощи ожидания.
  public void Write(int threadNumber)
  {
    // Запрашиваем блокировку для записи.
    m_rwl.AcquireWriterLock(Timeout.Infinite);
    // Обязательно объявим защищенный блок для того, чтобы
    // гарантировать освобождение блокировки записи.
    try
    {
      // Сообщаем о начале операции записи
      // и о номере потока, запросившего операцию записи.
      Console.WriteLine("Start -  resource |writing| for thread
{0}",threadNumber);
      // Имитируем запись данных при помощи
      // кратковременной задержки.
      Thread.Sleep(500);
      // Сообщаем об окончании операции записи данных.
      Console.WriteLine("Stop  - resource |writing| for thread
{0}",threadNumber);
    }
    // Защищенный блок будет выполнен всегда.
    finally
    {
      // Освобождаем блокировку чтения.
      m_rwl.ReleaseWriterLock();
    }
```

```
  }
};
// Основной класс нашего приложения.
class App
{
  // Количество потоков, которые будут тестировать
  // операции чтения/записи класса, описанного ранее.
  private static int m_ThreadsCount = 20;

  // Событие, информирующее об окончании
  // работы всех потоков.
  private static AutoResetEvent  m_EndEvent = new AutoResetEvent(false);
  // Экземпляр класса, предоставляющий синхронизированные
  // операции ввода/вывода.
  private static Resource m_Res = new Resource();
  // Точка входа в наше приложение.
  public static void Main()
  {
    // Создаем в пуле двадцать потоков, передавая
    // каждой функции потока ее порядковый номер
    // в интервале от нуля до 20.
    for (int threadNumber = 0; threadNumber < 20; threadNumber++)
      ThreadPool.QueueUserWorkItem(
new WaitCallback(WorkWithResource), threadNumber);
    // Ожидаем завершения всех потоков.
      m_EndEvent.WaitOne();
      Console.Write("All operation completed successfull");
  }
  // Функция наших потоков, которая в качестве параметра принимает
  // свой номер.
  private static void WorkWithResource( object state )
  {
    // Номер нашего потока.
    int MyThreadNumber = (int)state;
    // Выбираем для потока операцию ввода/вывода
    // при помощи операции взятия остатка от деления.
    // То есть, по-русски, если номер потока нечетный, то
    // читаем, иначе пишем.
    if ((MyThreadNumber % 2) == 1)
```

```
      // Если номер потока нечетный, то осуществляем чтение.
      m_Res.Read(MyThreadNumber);
    else
      // Если номер потока четный — осуществляем запись.
      m_Res.Write(MyThreadNumber);
    // Уменьшаем счетчик количества работающих потоков,
    // если это последний, поток сигнализирует о завершении
    // всех операций ввода/вывода.
    if (Interlocked.Decrement(ref m_ThreadsCount) == 0)
      m_EndEvent.Set();
  }
};
```

В результате работы примера на консоль были выведены следующие строки.

```
Start -  resource |writing| for thread 0
Stop  -  resource |writing| for thread 0
Start -  resource |reading| for thread 1
Stop  -  resource |reading| for thread 1
Start -  resource |writing| for thread 4
Stop  -  resource |writing| for thread 4
Start -  resource |reading| for thread 3
Start -  resource |reading| for thread 5
Stop  -  resource |reading| for thread 3
Stop  -  resource |reading| for thread 5
Start -  resource |writing| for thread 6
Stop  -  resource |writing| for thread 6
Start -  resource |reading| for thread 11
Start -  resource |reading| for thread 7
Start -  resource |reading| for thread 9
Stop  -  resource |reading| for thread 11
Stop  -  resource |reading| for thread 7
Stop  -  resource |reading| for thread 9
Start -  resource |writing| for thread 8
Stop  -  resource |writing| for thread 8
Start -  resource |reading| for thread 17
Start -  resource |reading| for thread 13
Start -  resource |reading| for thread 15
```

```
Stop  -  resource |reading| for thread 17
Stop  -  resource |reading| for thread 13
Stop  -  resource |reading| for thread 15
Start -  resource |writing| for thread 10
Stop  -  resource |writing| for thread 10
Start -  resource |reading| for thread 19
Stop  -  resource |reading| for thread 19
Start -  resource |writing| for thread 12
Stop  -  resource |writing| for thread 12
Start -  resource |writing| for thread 14
Stop  -  resource |writing| for thread 14
Start -  resource |writing| for thread 16
Stop  -  resource |writing| for thread 16
Start -  resource |writing| for thread 18
Stop  -  resource |writing| for thread 18
Start -  resource |writing| for thread 2
Stop  -  resource |writing| for thread 2
All operation completed successfull
```

Из лога четко видно, что операции чтения происходили по несколько штук одновременно, в отличие от операций записи, которые всегда идут поодиночке.

Хотелось бы ненадолго вернуться к листингу и обратить ваше внимание на способ освобождения блокировок.

```
...
finally
{
  m_rwl.ReleaseReaderLock();
}
...
finally
{
  m_rwl.ReleaseReaderLock();
}
...
```

Данные методы помещены в `finally`-блоки, естественно, неспроста, они позволяют гарантировать стопроцентный вызов методов, освобождающих блокировки. Это крайне важно, поскольку если хоть одна блокировка не будет снята, то дальнейший ход работы приложения будет непредсказуем.

Либо будут блокированы вообще все операции (если сбой произойдет при записи), либо навсегда будут блокированы операции записи (если сбой произойдет при чтении).

Расширенные возможности

Иногда во время операции чтения вдруг может понадобиться совершить запись в источник данных. Если действовать в соответствии со всеми правилами, то есть необходимо запросить соответствующую блокировку. Но интуитивно понятно, что при этом возникнет взаимная блокировка, которая приведет к зависанию приложения. Дабы мои слова не казались пустыми, модифицируем предыдущий пример, вставив операцию записи внутрь операции чтения.

```
...
  public void Read(int threadNumber)
  {
    // Запрашиваем блокировку для чтения.
    m_rwl.AcquireReaderLock(Timeout.Infinite);
    // Обязательно объявим защищенный блок для того, чтобы
    // гарантировать освобождение блокировки чтения.
    try
    {
      // Сообщаем о начале операции чтения данных.
      Console.WriteLine("Start -  resource |reading| for thread {0}",
threadNumber);
      // Эмулируем чтение данных при помощи задержки.
      Thread.Sleep(500);
      // Сообщаем о завершении операции чтения данных.
      Console.WriteLine("Stop  -  resource |reading| for thread {0}",
threadNumber);
      // ВНИМАНИЕ! Вот здесь-то мы и зависнем при попытке записи.
      Write(threadNumber);
    }
    finally
    {
      // Освобождаем блокировку чтения.
      m_rwl.ReleaseReaderLock();
    }
  }
...
```

В результате запуска приложение будет намертво заморожено по описанным ранее причинам.

Специально для этого случая предусмотрены сервисы преобразования блокировок из чтения в запись и обратно. Для реализации этих целей предназначены следующие методы.

```
// Позволяет сконвертировать блокировку записи в блокировку чтения,
// то есть наша операция чтения под действием этой функции
// превратится в операцию записи.
public LockCookie ReaderWriterLock.UpgradeToWriterLock(
  // Время ожидания.
  int millisecondsTimeout
);
// Позволяет операцию записи обратно превратить в операцию чтения.
public void ReaderWriterLock.DowngradeFromWriterLock(
  // Идентификатор блокировки.
  ref LockCookie lockCookie
);
```

Первый метод производит попытку преобразования уже запрошенной блокировки чтения в блокировку записи. Если это удается, то он возвращает специальный объект `LockCookie`, который после завершения операции записи позволит вернуть блокировку в прежнее состояние. Делается это при помощи второго метода.

Продемонстрируем работу с данными методами, модифицировав предыдущий пример.

```
...
  public void Read(int threadNumber)
  {
    // Идентификатор преобразования блокировки.
    LockCookie l_lk;
    // Запрашиваем блокировку для чтения.
    m_rwl.AcquireReaderLock(Timeout.Infinite);
    // Обязательно объявим защищенный блок для того, чтобы
    // гарантировать освобождение блокировки чтения.
    try
    {
      // Сообщаем о начале операции чтения данных.
      Console.WriteLine("Start -  resource |reading| for thread {0}",
threadNumber);
      // Эмулируем чтение данных при помощи задержки.
```

```
      Thread.Sleep(500);
      // Сообщаем о завершении операции чтения данных.
      Console.WriteLine("Stop  -  resource |reading| for thread {0}",
threadNumber);
      // ВНИМАНИЕ! Вот так надо проводить операцию записи
      // из операции чтения.
      // Конвертируем блокировку в операцию записи.
      l_lk = m_rwl.UpgradeToWriterLock(Timeout.Infinite);
      // Выполняем запись.
      Write(threadNumber);
      // Конвертируем блокировку обратно.
      m_rwl.DowngradeFromWriterLock(ref l_lk);
      // А вот здесь то мы и зависнем при попытке записи,
      // поскольку не произвели преобразование блокировок.
      // Write(threadNumber);
    }
    // Защищенный блок будет вызван в любом случае.
    finally
    {
      // Освобождаем блокировку чтения.
      m_rwl.ReleaseReaderLock();
    }
  }
...
```

Во время выполнения этот пример, в отличие от своего предыдущего собрата, не зависнет, а будет работать, как швейцарские часы.

Также хотелось бы предупредить об одном нюансе захвата блокировок. Для того чтобы освободить блокировку, необходимо столько раз сделать это, сколько раз она была захвачена. Класс `ReadWriterLock` ведет внутренний подсчет количества попыток захвата и освобождения блокировок.

И в заключение раздела рассмотрим два интересных метода класса `ReaderWriteLock`.

```
// Переводит блокировку в свободное состояние.
public LockCookie ReaderWriterLock.ReleaseLock();
// Восстанавливает состояние блокировки.
public void ReaderWriterLock.RestoreLock(
  // Идентификатор блокировки
ref LockCookie lockcookie
};
```

Первый освобождает блокировку независимо от того, сколько до этого раз она была захвачена. Второй метод восстанавливает состояние блокировки, основываясь на информации, предоставленной ему через объект `LockCookie`.

Вас удивляет, для чего может понадобиться временно снимать блокировку? Чтобы развеять сомнения в надобности данных методов, приведем простой пример. Предположим, что операции чтения и записи проходят через сеть с низкой пропускной способностью, а для увеличения производительности работы программы используется локальный кэш. Во время проведения операций обмена данными по сети, вы сможете освобождать блокировку, на время ожидания подтверждения операции от удаленного сервера. Тем самым вы позволите на время передачи запроса по сети использовать другим потокам локальный кэш вашего источника. Правда, в этом случае необходимо предупредить возникновение коллизий при попытках одновременного обращения к сетевому источнику.

9.8. Пул потоков

При создании приложений, интенсивно использующих потоки, вы рано или поздно столкнетесь с неудобством ручного создания и управления потоками. Ярким тому примером может служить программа, демонстрирующая синхронизацию операций ввода/вывода, написанная нами в предыдущем разделе. Для автоматизации процесса управления потоками в стандартную библиотеку .NET был введен класс, организующий доступ к пулу потоков.

```
public sealed class ThreadPool
```

Принцип работы пула потоков предельно прост. Вы лишь регистрируете в нем функции потоков. А он уже автоматически создает для них потоки и контролирует их. В очереди пула потоков на ожидании может находиться до 100 функций. Одновременно пул исполняет не более 25 потоков, остальные функции находятся в очереди ожидания. Как только один из потоков завершит свою работу, менеджер пула автоматически выберет из очереди ожидания ближайшую функцию и создаст на ее основе поток. Таким образом, на совести программиста остается лишь добавление необходимых функций в очередь пула потоков, а все остальное будет сделано автоматически.

Для регистрации собственных функций в пуле потоков предназначены следующие две функции:

```
// Регистрирует функцию в пуле потоков.
public static bool ThreadPool.QueueUserWorkItem(
  // Делегат функции потока.
  WaitCallback callBack
};
// Регистрирует функцию в пуле потоков, позволяя передать ей параметр.
```

```
public static bool ThreadPool.QueueUserWorkItem(
  // Делегат функции потока.
  WaitCallback callback
  // Параметр для функции потока.
  object state
  };
```

В качестве основного параметра должен быть передан экземпляр делегата `WaitCallback`, который должен указывать на пользовательскую функцию.

```
public delegate void WaitCallback(
 // Параметр для функции потока, который позволяет передать
 // дополнительную информацию.
 object state
);
```

Как видите, использование пула дает нам еще одно преимущество по сравнению с обыкновенным способом создания потоков, мы можем передать параметр нашей функции потока.

Приведем простейший пример работы с пулом потоков (листинг 9.23). Он будет добавлять в пул два делегата, указывающие на одну и ту же функцию, при этом в каждом случае он будет передавать ей свой персональный параметр. Функция потока будет выводить на консоль значение переданной ей переменной, что позволит зарегистрировать количество созданных потоков.

Листинг 9.23. Простейший пример использования пула потоков

```
/*
  Листинг 9.23
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу с потоками.
using System.Threading;
// Основной класс нашего приложения.
class App
{
  // Функция потока.
  public static void ThreadProc(object state)
  {
```

```
    // Производим явное преобразование типов.
    String str = (String)state;
    // Выводим на консоль переданное сообщение.
    Console.WriteLine("state = {0}",state);
  }
  // Точка входа в наше приложение.
  public static void Main()
  {
    // Добавляем в пул нашу функцию.
    ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc), "1");
    // Добавляем в пул еще одну функцию.
    ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc), "2");
    // Приостававливаем главный поток.
    Thread.Sleep(-1);
  }
};
```

В результате работы приложения на консоль будут выведены следующие строки.

```
1
2
```

Необходимо отметить одну интересную особенность работы метода `QueueUserWorkItem`. В качестве первого параметра он принимает делегат, который, по идее, должен указывать на пользовательскую функцию потока. Но делегаты обладают одним интересным свойством: они могут указывать на несколько функций. Это позволит нам разом добавлять в пул потоков по несколько функций. Продемонстрируем это, немного подправив предыдущий листинг.

```
ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc) + new
WaitCallback(ThreadProc), "1");
```

В результате работы измененного приложения на консоль будут выведены следующие строки.

```
1
1
2
```

Все потоки, создаваемые пулом, имеют нормальный приоритет. Но вы в любой момент можете изменить приоритет прямо из функции потока.

С потоками, создаваемыми через пул, связан еще один неприятный момент: все они являются фоновыми. Именно поэтому в конце предыдущего приложения

пришлось добавить вызов метода Sleep, блокирующий основной поток приложения, иначе бы его работа завершалась досрочно.

Если потоки, выполняемые в пуле, являются критичными для приложения, стоит самостоятельно изменить их статус с фонового на основной. Сделать это можно следующим образом.

```
Thread.CurrentThread.IsBackground = false;
```

Естественно, из самой функции потока.

Расширенные возможности пула потоков

Помимо основных своих функций пул потоков предоставляет дополнительные бонусные возможности.

К их числу можно смело отнести регистрацию функций, реагирующих на изменение состояния объектов синхронизаций — событий или мьютексов.

Для регистрации подобных функций предназначен следующий метод.

```
// Регистрирует функцию в пуле потоков, которая вызывается при
// изменении состояния указанного события.
public static RegisterWaitHandle ThreadPool.RegisterWaitForSingleObject(
  // Объект для ожидания.
  WaitHandle waitObject
  // Делегат функции уведомления о происхождении события.
  WaitOrTimerCallback,
  // Параметр для функции уведомления.
  object state,
  // Задержка в миллисекундах.
  int millisecondsTimeout,
  // Уведомлять о событии только один раз.
  bool executeOnlyOnce
);
```

Теперь вам больше не надо создавать дополнительные потоки, ожидающие изменения состояния события. Вам необходимо лишь зарегистрировать функцию в пуле потоков, которая будет автоматически вызвана при изменении состояния указанного события.

Обратите внимание на объект `RegisterWaitHandle`, возвращаемый методом. Он позволит вам в случае необходимости отключить функцию ожидания от объекта. Для этого вам придется воспользоваться методом

```
public bool RegisteredWaitHandle.Unregister(
  // Событие, уведомляющее об успехе отмены.
  WaitHandle waitObject
);
```

Приведем простейший пример использования функций уведомления (листинг 9.24). Приложение создаст простое событие с автоматическим сбросом, затем зарегистрирует функцию, ожидающую изменение данного события, после чего с чистой совестью собственноручно изменит состояние события.

Листинг 9.24. Слежение за изменением состояния события при помощи пула потоков

```
/*
  Листинг 9.24
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за работу с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Событие, за которым будет следить пул потоков.
  static AutoResetEvent are = new AutoResetEvent(false);
  // Функция уведомления об изменении события.
  public static void Event (object state, bool timeOut)
  {
    // В реальной программе здесь может происходить
    // что-нибудь и более значимое, но мы лишь
    // сообщим пользователю об изменении состояния события.
    String str = (String)state;
    Console.WriteLine(str);
  }
  // Точка входа в приложение.
  public static void Main()
  {
    // Регистрируем функцию уведомления для нашего события.
    ThreadPool.RegisterWaitForSingleObject(
      // Это событие, которе будет стеречь пул потоков
      are,
      // Создадим делегат, ссылающийся на нашу функцию
```

```
      // уведомления.
      new WaitOrTimerCallback(Event),
      // Это параметр, который мы хотим передать функции
      // уведомления
      "Hello, World!",
      // Ждем бесконечно.
      Timeout.Infinite,
      // Уведомлять о событии только один раз.
      true);
    // Изменяем состояние события.
    are.Set();
    // Ждем бесконечно.
    Thread.Sleep(Timeout.Infinite);
  }
};
```

В результате работы данного приложения на консоль будет выведена следующая строка.

```
Hello, World!
```

Данная функция может существенно облегчить разработку ваших многопоточных приложений, главное не забывать, что функция уведомления будет вызываться в собственном персональном потоке.

Максимальное количество потоков в пуле

Ранее говорилось, что пул потоков по умолчанию одновременно исполняет только 25 потоков. Отчасти это верно. Но я обязан вас предупредить, что данное значение может быть изменение сервером среды исполнения. В случае обычных приложений .NET у среды исполнения нет сервера, она работает под прямым управлением операционной системы. Но среда .NET может встраиваться в другие приложения, к примеру, в виде банального скриптового движка (IIS (ASP.NET), Yukon). В этом случае сервер среды .NET может изменить количество одновременно исполняемых пулом потоков при помощи следующего метода, входящего в состав интерфейса `ICorThreadpoolVtbl`,

```
HRESULT ( STDMETHODCALLTYPE *CorSetMaxThreads )(
  // Указатель на пул потоков.
  ICorThreadpool * This,
  // Количество потоков.
  /* [in] */ DWORD MaxWorkerThreads,
```

```
  // Количество функций уведомления.
  /* [in] */ DWORD MaxIOCompletionThreads
);
```

Для того чтобы определить максимальное количество потоков, которые могут одновременно исполняться в пуле.

```
// Данная функция сообщает о максимальных параметрах пула.
public static void ThreadPool.GetMaxThreads(
  // Количество потоков.
  out int workerThreads,
  // Количество функций уведомлений.
  out int completionPortThreads
);
```

В заключение раздела хотелось бы отметить, что программисты из Microsoft на славу потрудились, создавая пул потоков. Теперь его использование стало простым и интуитивно очевидным, чего никак не скажешь о родном пуле потоков самой операционной системы.

9.9. Таймеры

Практически ни одно современное приложение не обходится без использования таймеров. Использование таймеров в среде .NET стало как никогда ранее просто и примитивно. Вам лишь нужно создать объект-таймер, передав его конструктору делегат, указывающий на вашу функцию, время начала срабатывания и также интервал срабатывания.

```
public Timer(
  // Делегат для функции таймера.
  TimerCallback callback,
  // Параметр для функции.
  object state,
  // Время запуска.
  int dueTime,
  // Интервал запуска.
  int period
);
```

У этого конструктора существует несколько версий, позволяющих задать время в самых различных форматах, но принцип работы один и тот же.

Для закрытия таймера предполагается использовать метод `Dispose`, уничтожающий объект таймера. После чего никакие уведомления и вызовы пользовательских функций происходить не будут.

Необходимо упомянуть о возможности динамического изменения параметров таймера. По ходу его работы можно изменять время запуска и интервал срабатывания. Для этого предназначен следующий метод.

```
// Позволяет изменить параметры таймера.
public bool Timer.Change(
  // Время запуска.
  int dueTime,
  // Интервал запуска.
  int period
);
```

Этот метод, точно так же как и конструктор класса, имеет несколько модификаций, позволяющих задать время в различных форматах.

Приведем пример, демонстрирующий работу с таймерами (листинг 9.25). Он будет создавать таймер с периодом срабатывания полсекунды, к которому присоединит функцию, выводящую сообщение на консоль. Затем приложение будет ожидать ввода от пользователя, после которого увеличит интервал срабатывания таймера до двух секунд. После следующего ввода пользователя таймер будет уничтожен.

Листинг 9.25. Пример простейшего использования таймеров

```
/*
  Листинг 9.25
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы
// с потоками.
using System.Threading;
// Основной класс приложения.
class App
{
  // Функция таймера, которая будет вызываться через определенные
  // промежутки времени, определенные при создании таймера.
  private static void TimerProc(object state)
  {
    // Выводим на экран информацию, используя параметр.
    Console.WriteLine("state = {0}",(String)state);
  }
  // Точка входа в наше приложение.
```

```
  public static void Main()
  {
    // Создаем таймер.
    Timer timer = new Timer(new TimerCallback(TimerProc),"Param" ,0 , 500);
    // Попросим пользователя нажать клавишу Enter для того, чтобы
    // несколько замедлить работу таймера.
    Console.WriteLine("Press Enter to slow timer");
    Console.ReadLine();
    Console.WriteLine("Timer slowed");
    timer.Change(0,2000);
    // Приглашем пользователя нажать Enter, для того
    // чтобы остановить таймер.
    Console.WriteLine("To Stop timer press Enter");
    Console.ReadLine();
    // Уничтожаем таймер.
    timer.Dispose();
    // Ждем нажатия клавиши Enter, а затем завершаем работу.
    Console.WriteLine("Press Enter to close programm");
    Console.ReadLine();
  }
};
```

Когда я только начал использовать таймеры, делал я это с большой осторожностью, выбирая лишь малые интервалы времени их срабатывания. Сейчас для меня не совсем ясно, почему я так поступал, но с точностью могу вам сказать, что таймеры Windows — это достаточно надежный инструмент. И вы можете не опасаясь задавать в качестве интервала срабатывания не только секунды, но даже месяцы и годы. Если, конечно, ваша программа будет непрерывно работать в течение такого большого промежутка времени.

9.10. Датчики производительности

Вводя дополнительные потоки в приложения, мы надеемся выиграть в производительности. Но иногда все происходит совершенно наоборот. В большинстве случаев это происходит по неопытности со всеми, кто впервые сталкивается с многопоточным программированием. Уж больно неординарна эта тема.

В отладке многопоточных приложений вам очень помогут показания датчиков производительности, которые предоставят исчерпывающие сведения о статистике работы вашей программы.

Нам предстоит работать с датчиками из группы. Сделать это можно, используя диалоговое окно, изображенное на рис. 9.7.

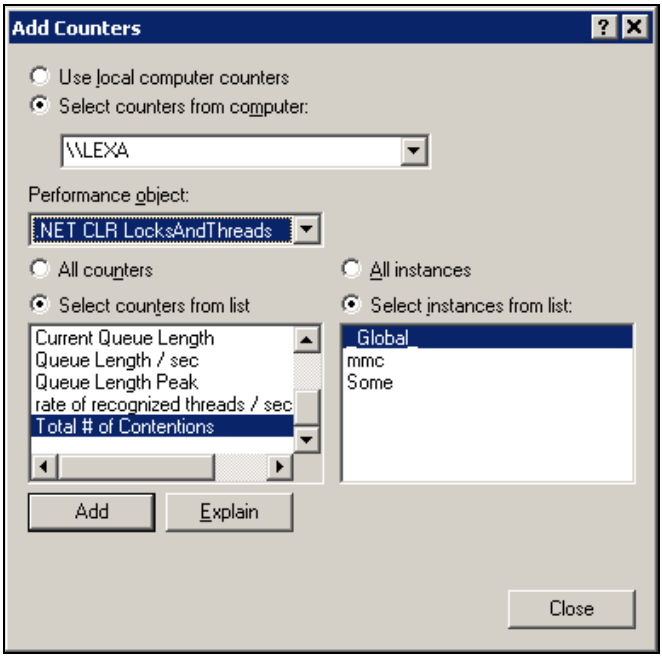
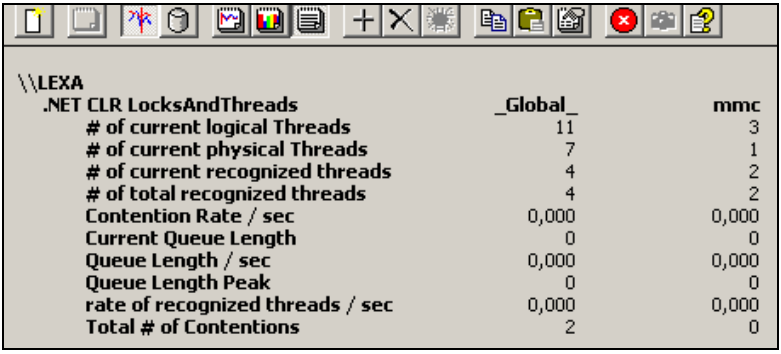


Рис. 9.7. Добавление показаний датчиков производительности в отчет



\\LEXA

.NET CLR LocksAndThreads	_Global_	mmc
# of current logical Threads	11	3
# of current physical Threads	7	1
# of current recognized threads	4	2
# of total recognized threads	4	2
Contention Rate / sec	0,000	0,000
Current Queue Length	0	0
Queue Length / sec	0,000	0,000
Queue Length Peak	0	0
rate of recognized threads / sec	0,000	0,000
Total # of Contentions	2	0

Рис. 9.8. Отображение показаний датчиков в режиме отчета

Для ознакомления с информацией, предоставляемой датчиками, я бы рекомендовал выбрать все возможные датчики из группы .NET CLR `LocksAndThreads`, а также их модификации. После чего переключиться в режим отображения отчета (рис. 9.8).

Далее приведено краткое описание каждого из интересующих нас датчиков (табл. 9.6).

Таблица 9.6. *Описание датчиков производительности*

Датчик	Описание
`# of current logical Threads`	Отображает общее количество управляемых потоков, включая как работающие, так и приостановленные потоки. Показатель не усредняется по времени, а отображает ситуацию на момент снятия показаний
`# of current physical Threads`	Количество системных потоков, которые используются для поддержания функционирования управляемых потоков. В число этих потоков не входят потоки, используемые средой исполнения во внутренних целях
`# of current recognized Threads`	Количество потоков, запущенных без помощи среды исполнения внешними модулями. Для них среда исполнения автоматически создает обертки в виде экземпляров класса `Thread`
`# of total recognized Threads`	Суммарное количество внешних потоков, запущенных из внешних модулей. Сюда включаются все потоки за время работы программы
`Contention Rate / Sec`	Количество неудачных запросов управляемых блокировок в секунду
`Current Queue Length`	Количество потоков, находящихся в очереди ожидания блокировок. Показатель отображает текущую ситуацию, не усредняя значение по времени
`Queue Length / Sec`	Длина очереди ожидания в секунду. Особенность счетчика заключается в том, что он не усредняет значение на всем протяжении снятия показаний, а вычисляет средние по двум соседним значениям
`Queue Length Peak`	Общее количество потоков, которым пришлось находиться в очереди ожидания во время работы программы
`rate of recognized threads / sec`	Количество потоков в секунду, которые создаются внешними модулями, и обертываются средой исполнения при помощи объектов `Thread`
`Total # of Contentions`	Общее количество неудачных попыток запроса блокировок

Анализируя показания этих датчиков, иногда можно узнать очень много нового о своих программах. Как будто вовсе и не ты писал это приложение.

Заключение

Работа с потоками является одной из наиболее сложных задач, встающих перед программистами при написании прикладных программ. Здесь большую роль играют чутье и опыт и менее — здравый смысл. Поэтому не расстраивайтесь, если что-то не будет получаться с первого раза, практика — лучшая школа программирования.

Глава 10

Архитектура доменов

Платформа .NET изначально проектировалась как сверхнадежная и высокопроизводительная система. Для достижения этой цели архитекторам .NET пришлось создать ряд новых технологий, которые обеспечили необходимый уровень стабильности и надежности. К одной из таких технологий можно смело отнести поддержку доменной структуры в приложениях .NET. Ранее подобная технология нигде широко не использовалась, это фирменная разработка архитекторов Microsoft .NET. В этой главе кратко рассмотрено описание технологии, а также несколько примеров, демонстрирующих работу с ней.

10.1. Развитие технологий изолирования кода

Во времена развития ДОС, приложениям было дозволено все, никаких ограничений на производимые операции со стороны операционной системы не было. Программы могли обращаться к любым участкам памяти и даже напрямую к коду ядра операционной системы, минуя стандартные шлюзы. Доступ к оборудованию был также абсолютно свободным. Подобная анархия отчасти оправдывалась однозадачной природой системы. Поскольку одновременно на такой системе могло исполняться лишь одно приложение, то ему разрешались любые действия, лишь бы оно обеспечивало стабильную работу.

С появлением многозадачных систем положение дел резко изменилось. Теперь необходимо было обеспечить персональную изоляцию каждого приложения от действий других приложений, а также защитить саму операционную систему от несанкционированных посягательств. Идеология многозадачных систем подразумевает, что даже нестабильно работающее приложение не должно никоим образом влиять на работу других, а также на работу системы в целом. Добиться этого удалось при помощи введения процессов — специальных оболочек, ограничивающих полномочия отдельных программ. Для каждой программы создается свой персональный процесс, в который она

и помещается. Процесс заставляет программу поверить, что она монопольно владеет системой: ей предоставляется собственное виртуальное адресное пространство размером 4 гигабайта для 32-разрядных систем и 16 Гбайт оперативной памяти для 64-разрядных систем, дополнительно программе предоставляются ссылки на определенные системные сервисы. Однако память, предоставляемая приложению через процесс, является виртуальной. Приложение лишь "думает", что работает с настоящей памятью, на самом деле все его действия контролируются со стороны операционной системы совместно с процессором. Для обычных приложений существенно понижен уровень привилегий доступа к системным сервисам, а также полностью закрыт доступ к оборудованию. С одной стороны, процессы позволяют поддерживать стабильность в системе, несмотря на некорректные операции отдельных приложений, а с другой стороны, предоставляют отдельным приложениям удобную и комфортную среду обитания. Единственным недостатком процессов является необходимость организации взаимодействия между ними. Специально для этого операционная система предоставляет набор механизмов, позволяющих процессам взаимодействовать между собой. Они носят название IPC (Inter Process Communication — межпроцессное взаимодействие). Ниже приведен список механизмов, предназначенных для обеспечения взаимодействия между процессами:

- буфер обмена;
- COM-технология (локально работает через LPC);
- копирование данных, через сообщение WM_COPYDATA;
- DDE (Dynamic Data Exchange, динамический обмен данными);
- файлы, проецируемые в память;
- почтовые ящики;
- каналы (трубы — pipes);
- удаленный вызов процедур RPC;
- сокеты;
- локальный вызов процедур LPC (Local Procedure Call) (является внутренним недокументированным механизмом).

Таким образом, для обеспечения взаимодействия или обмена данными между процессами A и B, необходимо воспользоваться одним из этих сервисов (рис. 10.1).

У всех этих сервисов, пожалуй, за исключением файлов, проецируемых в память, есть один существенный недостаток — они слишком медленно работают. Связано это с тем, что при обращении к другому процессу приходится копировать данные, которые необходимо ему сообщить. К тому же, механизмы IPC реализованы в режиме ядра и при обращении к ним проис-

ходят неоднократные переключения контекста защиты, что также не способствует увеличению скорости их работы.

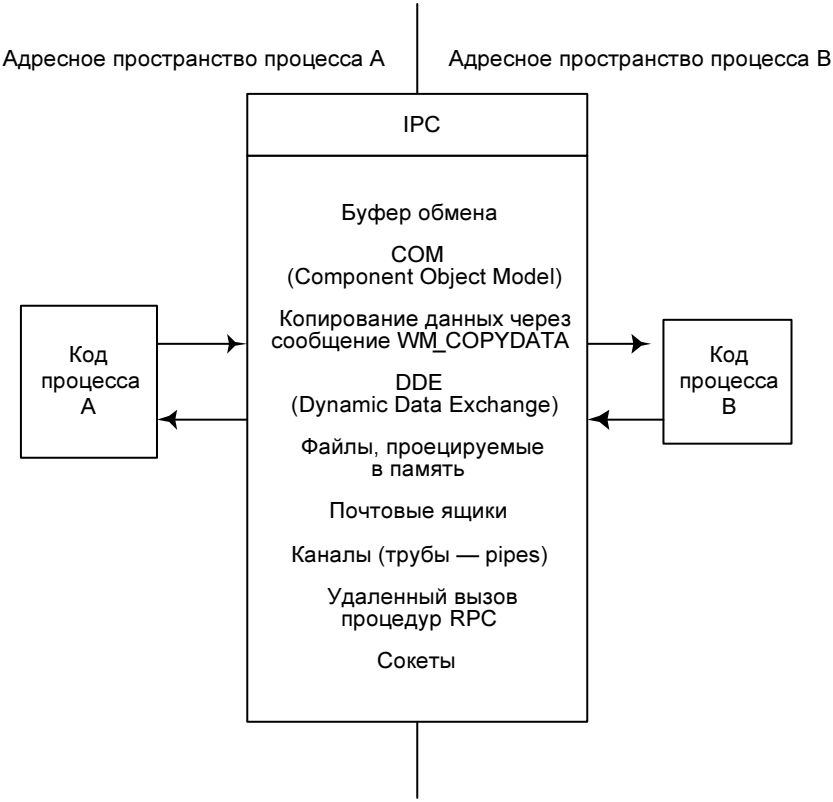


Рис. 10.1. Средства IPC, предназначенные для взаимодействия с процессами

Необходимо отметить еще одну важную роль процессов: они управляют уровнем привилегий кода, расположенного в них. Каждому процессу выдается определенный набор привилегий — разрешений на доступ к определенным сервисам, что позволяет еще более четко контролировать код, исполняющийся в них. Таким образом, процессы отлично справляются с задачей изоляции кода, но обладают недостаточно производительными механизмами взаимодействия между ними.

При разработке платформы .NET было решено разработать технологию изоляции кода, которая была бы лишена этих недостатков. Такой технологией стали домены.

10.2. Домены в приложениях .NET

Домены очень похожи на процессы за тем лишь исключением, что они существуют не на уровне операционной системы, а внутри управляемых приложений. Любое приложение по умолчанию содержит хотя бы один домен — основной. Приложение может создать дополнительные домены, которые будут изолированы друг от друга, а также от основного домена приложения. Каждый экземпляр любого типа обязательно приписан к одному из доменов, помимо этого каждая сборка, загружаемая в приложение, будет приписана к одному из доменов. Таким образом, объекты и код сборки, приписанные к одному домену, напрямую не будут доступны из других доменов. Для того чтобы обратиться к объекту, приписанному к одному домену из другого, придется воспользоваться технологией .NET Remoting, которая является аналогом IPC (рис. 10.2).

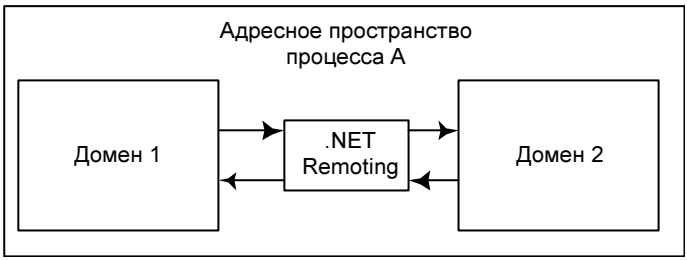


Рис. 10.2. Поддержка связи между доменами на основе технологии .NET Remoting

Механизмы .NET Remoting, по сравнению с обычными сервисами IPC, на порядок производительней. Объясняется это тем, что механизмы .NET Remoting, при взаимодействии между доменами, работают в контексте одного процесса, поэтому удается избежать лишних переключений общего контекста защиты, а также многих других неприятных эффектов. К тому же, внутренние механизмы .NET Remoting достаточно грамотно спроектированы и оптимизированы. В особо критичных случаях удается даже избежать бессмысленного копирования данных в одном процессе, а наладить взаимодействие с объектом практически напрямую, через, так называемые, прозрачные переходники (Transparent Proxy), которые контролируют лишь вопросы безопасности и защиты.

Домены являются независимыми частями приложений .NET, гарантирующие полную независимость от остальных его частей (доменов). Даже если в одном из доменов произойдет критическая ошибка или прорыв в системе защиты, это никоим образом не скажется на остальных доменах. Они будут работать, как и раньше. Такое уникальное свойство доменов поможет разработчикам высокопроизводительных масштабируемых серверных приложений. Теперь можно не прибегать к использованию механизма имперсонации,

нужно лишь выставить необходимые атрибуты защиты для домена, в котором предполагается обрабатывать пользовательские запросы. Также в доменах можно располагать код сторонних производителей, в надежности которого приходится сомневаться. В этом случае, даже если внешний модуль и вызовет какие-либо проблемы, "упадет" лишь тот домен, к которому он приписан, не повлияв на работу приложения в целом.

Для наглядности приведем пример, демонстрирующий работу с доменами (листинг 10.1). Он будет порождать второй домен (помимо основного), затем создавать в нем объект. После этого будет произведено обращение из основного домена приложения к объекту, находящемуся во втором домене, через механизм .NET Remoting. Для того чтобы было известно, в каком домене исполняется в текущий момент код, в критических точках приложение будет выводить информацию о домене приложения.

Листинг 10.1. Пример использования доменов

```
/*
  Листинг 10.1
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку технологии
// .NET Remoting.
using System.Runtime.Remoting;
// Введем объект, с которым будем взаимодействовать из другого домена.
class OtherType : MarshalByRefObject
{
  // Опишем обычную функцию, код которой будет автоматически
  // приписан к домену, в котором создан объект.
  public void Hello()
  {
    Console.WriteLine("Hello, World, from Domain whose name is {0}",
AppDomain.CurrentDomain.FriendlyName);
  }
};
// Основной класс приложения.
class App
{
```

```
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим второй домен приложения.
    AppDomain secondDomain = AppDomain.CreateDomain("Second Domain");
    // Внутри второго домена создадим объект.
    ObjectHandle objHandle =
secondDomain.CreateInstance("Some","OtherType");
    // Создадим прозрачный прокси-переходник
    // для взаимодействия с объектом из другого домена.
    OtherType otherTypeObject = (OtherType)objHandle.Unwrap();
    // Проверим, действительно ли нам предоставили прозрачный переходник.
    if(RemotingServices.IsTransparentProxy(otherTypeObject))
      Console.WriteLine("The unwrapped object is a proxy.");
    else
      Console.WriteLine("The unwrapped object is not a proxy!");
    // Выведем на консоль имя текущего домена приложения.
    Console.WriteLine("Main function of Application Domain name is {0}",
AppDomain.CurrentDomain.FriendlyName);
    // Обратимся к функции нашего объекта, находящегося в другом домене
    // приложения.
    otherTypeObject.Hello();
  }
}
```

В результате работы приложения на консоль будут выведены следующие строки.

```
The unwrapped object is a proxy.
Main function of Appliction Domain name is Some.exe
Hello World from Domain whose name is Second Domain
```

Как видно из листинга, взаимодействие с объектами из других доменов по большому счету тривиально. Не нужно использовать даже специальных сервисов, все обращения идут прямо через тип искомого объекта.

10.3. Сборки и домены

Таким образом, любая сборка обязательно должна быть приписана к одному из доменов приложения. К примеру, когда запускается новое управляемое приложение, среда исполнения создает основной домен и загружает туда

искомую сборку. Если приложение будет иметь много доменов, то встает вопрос о производительности подобного подхода. Ведь загружать для каждого домена свою копию сборки будет весьма и весьма накладно. Однако оказывается, этого не происходит за счет работы механизма разделения кода. Суть механизма заключается в том, что все элементы сборки, которые можно использовать совместно, никогда не копируются. А те элементы, которые должны быть уникальными для каждого домена, копируются только при действительной необходимости — фактически при первом обращении к ним.

Таким образом, код сборки всегда представлен в единственном экземпляре, а ее данные при необходимости копируются персонально для каждого домена. Приведем пример сборки, которая будет содержать и код, и данные (листинг 10.2).

Листинг 10.2. Пример сборки, содержащей данные и код

```
/*
   Листинг 10.2
   File:   Some.cs
   Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку пользовательского
// интерфейса.
using System.Windows.Forms;
// Это единственный класс, представленный в нашей сборке.
class Say
{
  // Это данные сборки.
  public string Message;

  // А это разделяемый код сборки.
  public void SayHello()
  {
    MessageBox.Show(Message);
  }
};
```

Если рассматривать сборку в контексте двухдоменного приложения, то получится следующая картина (рис. 10.3).

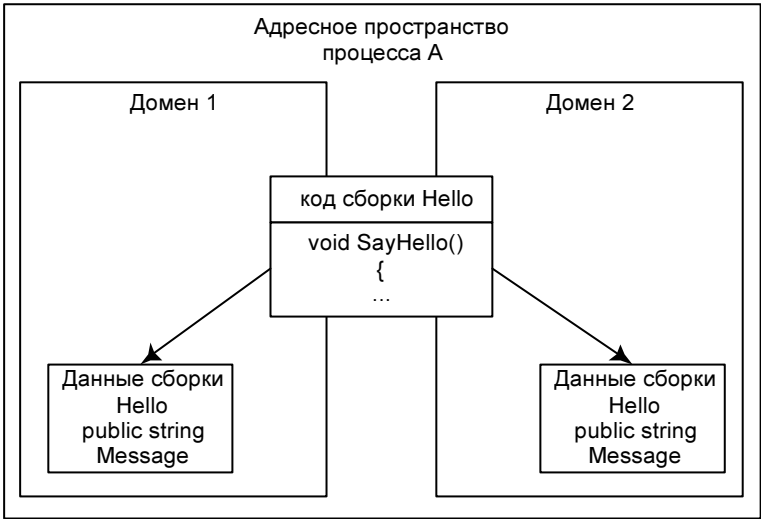


Рис. 10.3. Простая сборка в контексте двухдоменного приложения

На схеме показано, что код сборки используется совместно обоими доменами, а данные были размножены персонально для каждого из них. На деле оказывается, что такая стратегия иногда себя не оправдывает. Поэтому разработчиками были предложены еще несколько видов совместного использования сборок различными доменами. В частности, существует доменно-нейтральный режим использования сборки, в котором сборка вообще не обращает никакого внимания на домены.

Существует три схемы использования сборок по отношению к доменам.

- Загружать как доменно-нейтральную только `Mscorlib` (общую библиотеку классов), а все остальные сборки разделять между доменами классическим образом. Такая схема используется по умолчанию, поскольку она идеально подходит для большинства приложений.
- Загружать все сборки как доменно-нейтральные. Такая схема чаще используется для приложений, в контексте которых выполняется сразу несколько программ.
- Загружать как доменно-нейтральные только строго именованные сборки. Эта схема позволяет несколько ускорить загрузку сборок. При введении такой схемы было сделано предположение: если сборка является строго именованной, то могут использовать различные приложения из различных доменов. Если же сборка обычная, то она, скорее всего, будет востребована лишь одним из приложений и разделять ее между другими доменами смысла не имеет.

Отказ от концепции доменно-нейтральных сборок может служить еще одной, неочевидной с первого взгляда, цели. Таким образом, можно защитить

свои данные от несанкционированного посягательства. Если сборка принадлежит только одному домену, нам гарантировано, что все ее данные не могут быть несанкционированно использованы кодом из других доменов. Здесь, конечно, можно вспомнить о классической системе безопасности, основанной на правах, привилегиях, списках доступа (ACL — Access Control List). Однако предложенная схема гарантирует, что даже пользователь с высшим уровнем привилегий не сможет посягнуть на персональные данные домена.

10.4. Загрузка исполняемых файлов в домен

В целях демонстрации процесса загрузки сборок в домены, приведем пример приложения, запускающего несколько приложений в одной программе.

Уже было упомянуто о возможности запуска нескольких приложений внутри одного .NET-процесса. Для этого необходимо создать дополнительный домен и загрузить в него на исполнение стороннее приложение. Сделать это несложно, при помощи всего лишь двух строк кода (листинг 10.3).

Листинг 10.3. Приложение, запускающее несколько управляемых приложений

```
/*
   Листинг 10.3
   File:   Some.cs
   Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Основной класс нашего приложения.
class App
{
  // Точка входа в приложение.
  static void Main(String[] param)
  {
    foreach (String strApplicationName in param)
    {
     // Создаем второй домен.
     AppDomain SecondDomain = AppDomain.CreateDomain(strApplicationName↵
  + " - Non main domain");
    // Запускаем приложение в контексте второго домена.
```

```
      SecondDomain.ExecuteAssembly(strApplicationName);
    }
  }
};
```

При помощи этого приложения можно заставить несколько управляемых приложений работать в контексте одного процесса. Необходимо лишь указать имена их исполняемых файлов в качестве параметров командной строки.

10.5. Домены и потоки

Взаимоотношения потоков и доменов — это, пожалуй, одна из самых сложных для понимания тем в программировании для платформы .NET. Проблема заключается в том, что между потоками и доменами нет никаких взаимоотношений. Потоки и домены — это различные понятия, никоим образом не связанные между собой. Хотя это звучит довольно запутанно, на деле же оказывается все примитивно просто.

Поток — механизм распараллеливания кода, предназначенный для организации одновременного исполнения нескольких участков кода. Говоря упрощенно, потоки отвечают лишь за то, чтобы в вашей программе одновременно исполнялись две функции.

Домен — средство защиты и изоляции кода и данных. Домен отвечает за изолирование некоторых данных и кода от других доменов.

Поток всегда исполняется в одном из доменов приложения. Фактически, в один конечный момент поток приписан к одному из доменов приложений. В такие моменты потоку доступны только те данные, которые принадлежат его домену. Но ничто не мешает потоку перейти во владения другого домена, используя сервисы .NET Remoting. Схема взаимодействия потоков и доменов представлена на рис. 10.4.

Таким образом, потоки могут беспрепятственно, если не учитывать барьер системы безопасности, переходить из домена в домен.

Потоки и домены — это понятия, лежащие в разных плоскостях. Для лучшего понимания можно представить себе, что поток — это река, протекающая через разные провинции средневекового Китая, где каждая провинция представляет собой обособленный домен.

Для того чтобы продемонстрировать протекание потоков через домены, приведем пример. Модифицируем предыдущий пример так, чтобы при входе в различные домены он сообщал информацию о текущем потоке. Исходный код примера представлен в листинге 10.4.

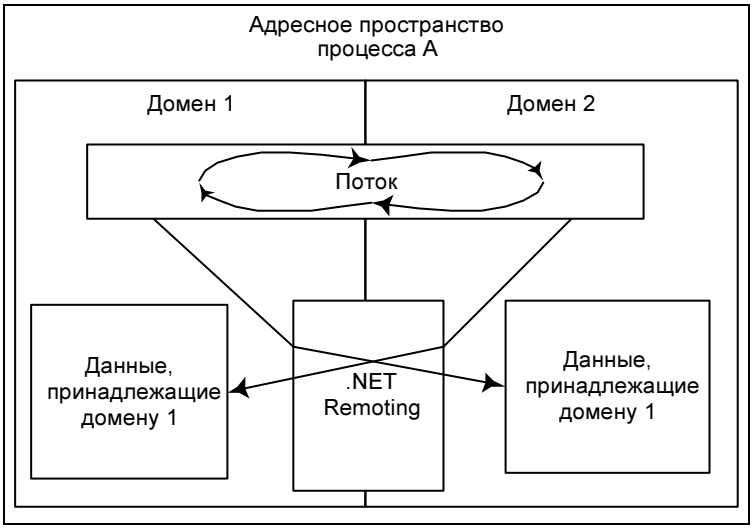


Рис. 10.4. Взаимодействие доменов и потоков

Листинг 10.4. Пример протекания потоков через домены

```
/*
   Листинг 10.4
   File:   Some.cs
   Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку технологии
// .NET Remoting.
using System.Runtime.Remoting;
// Подключим пространство имен, отвечающее за поддержку потоков.
using System.Threading;
// Введем класс, объекты которого будем приписывать к доменам.
class OtherType : MarshalByRefObject
{
  // Функция объекта, которую будем вызывать из других доменов.
  public void Hello()
  {
    // Сообщим пользователю имя домена, к которому приписан данный код.
    Console.WriteLine("Hello, World, from Domain whose name is {0}!",
```

```
AppDomain.CurrentDomain.FriendlyName);
    // Сообщим пользователю данные о текущем потоке.
    Console.WriteLine("Thread Hash code is {0}",
Thread.CurrentThread.GetHashCode());
  }
};
// Основной класс приложения.
class App
{
  // Точка входа в приложение.
  public static void Main()
  {
    // Создадим второй домен приложения.
    AppDomain secondDomain = AppDomain.CreateDomain("Second Domain");
    // Во втором домене приложения создадим объект.
    ObjectHandle objHandle = secondDomain.CreateInstance("Some","OtherType");
    // Создадим специальный прозрачный переходник для междоменного
    // взаимодействия с объектом.
    OtherType otherTypeObject = (OtherType)objHandle.Unwrap();
    // Проверим, действительно ли переходник является прозрачным.
    if(RemotingServices.IsTransparentProxy(otherTypeObject))
      Console.WriteLine("The unwrapped object is a proxy.");
    else
      Console.WriteLine("The unwrapped object is not a proxy!");
        // Сообщим пользователю имя текущего домена.
    Console.WriteLine("Main function of Appliction Domain name is {0}",
AppDomain.CurrentDomain.FriendlyName);
    // Сообщим пользователю уникальные данные текущего потока.
    Console.WriteLine("Thread Hash code is {0}",
Thread.CurrentThread.GetHashCode()c);
    // Обратимся к объекту, находящемуся в другом домене.
    otherTypeObject.Hello();
  }
}
```

В результате работы приложения на консоль будут выведены следующие строки.

```
The unwrapped object is a proxy.
Main function of Appliction Domain name is Some.exe
```

```
Thread Hash code is 8
Hello World from Domain whose name is Second Domain
Thread Hash code is 8
```

Содержание лога показывает, что, на протяжении жизни приложения, поток находился в двух различных доменах. При этом не использовалось никаких дополнительных сервисов для координирования их взаимодействия. Следовательно, можно заключить, что между ними связи нет.

10.6. Имена доменов

Все домены в среде исполнения имеют собственные удобочитаемые имена. Сделано это для комфортной работы программистов, чтобы они ориентировались не в безликих объектах доменов, а могли узнавать их по именам.

В большинстве случаев имя основного домена приложения совпадает с именем его сборки. Правда, из этого правила есть исключение: если приложение будет загружено в один из доменов другого, в этом случае за имя домена будет отвечать родительское приложение.

Приведем пример, который выводит на консоль имя основного домена приложения (листинг 10.5.).

Листинг 10.5. Пример, сообщающий пользователю имя основного домена приложения

```
/*
   Листинг 10.5
   File:   Some.cs
   Author: Дубовцев Алексей
*/
// Подключим основное пространство имен общей библиотеки классов.
using System;
// Подключим пространство имен, отвечающее за поддержку работы с потоками.
using System.Threading;
// Основной класс нашего приложения.
class App
{
  // Точка входа в наше приложение.
  public static void Main()
  {
    // Выведем на консоль имя основного домена нашего
    // приложения.
```

```
    // Метод GetDomain класса Thread возвращает объект,
    // представляющий текущий домен, а поскольку этот
    // код всегда  будет выполняться в основном домене
    //(если, конечно, приложение было загружено
    // нормально)
    // и потоке приложения, свойство FriendlyName
    // позволяет получить удобочитаемое имя домена
    Console.WriteLine(Thread.GetDomain().FriendlyName);
  }
};
```

Если запустить приложение обычным способом, то результат его работы будет следующий:

```
Some.exe
```

А вот если воспользоваться одним из примеров данной главы, позволяющим запускать несколько управляемых приложений в своем теле, тогда результат будет несколько отличаться:

```
Some2.exe — Non main domain
```

При первом знакомстве домены приложений вызывают мало приятных ощущений и ассоциаций. Но на деле их программирование довольно просто, главное — понимать их внутреннюю сущность, а также принципы работы механизмов, их поддерживающих.

Глава 11

Введение во взаимодействие с операционной системой

.NET является полностью независимой программной средой. Она создает для управляемых приложений .NET свой закрытый мир, предоставляя им богатый набор всевозможных сервисов, независимо от конкретной аппаратной и программной платформы, на которых реально будет исполняться приложение. Здесь под аппаратной платформой подразумевается процессор и материнская плата, на базе которых собран компьютер, или, другими словами, компьютерное "железо" (Intel, AMD, Apple). Под программной — используемая операционная система (Windows, FreeBSD, Linux, MacOs).

На момент написания книги были представлены реализации виртуальной машины .NET для всех операционных систем семейства Windows (9x, NT, CE), а также для MacOS, FreeBSD и даже Linux.

Примечание

Реализация виртуальной машины .NET под Linux носит название MONO. Это независимый проект с открытым исходным кодом, ознакомиться с которым вы можете на странице **http://www.go-mono.com**. На момент написания данной главы проект все еще находился в стадии разработки.

Теоретически любая программа, написанная для .NET, обладает феноменальной переносимостью, она без перекомпиляции будет работать на всех перечисленных платформах. Почему теоретически? А потому, что некоторые широко используемые типы из общей библиотеки .NET не предоставлены для всех платформ. К примеру, к ним относятся все классы из пространства имен `System.Windows.Forms`, отвечающие за поддержку оконного пользовательского интерфейса. Данные классы строго привязаны к операционным системам от Microsoft. Среда исполнения .NET предоставляет специальные механизмы для взаимодействя этих классов с сервисами операционной системы.

Примечание

Правда, разработчикам из команды MONO зависимость данных классов от Windows не мешает заниматься их переносом на Linux-платформу. На момент написания книги данная работа была завершена на 51%.

Благодаря мощности и гибкости данных сервисов разработчикам общей библиотеки классов не пришлось создавать собственную оконную подсистему, они лишь разработали набор классов, инкапсулирующих вызовы к подсистемам User и GDI, отвечающих за работу с окнами и графикой соответственно.

Таким образом, очевидна выгода от применения сервисов взаимодействия с операционной системой. Мы можем использовать наработанный годами код в новых .NET-приложениях. Правда, здесь есть одно *но*. При использовании данных сервисов переносимость приложений будет ограничена определенным кругом операционных систем, в нашем случае — семейством Windows: Window 95, Windows 98, Windows Me, Windows NT, Windows 2000, Windows XP Home и Professional, Windows 2003 Server. Если вы готовы к такому ограничению, что в большинстве случаев оказывается оправдано, тогда взаимодействие с операционной системой позволит существенно расширить спектр используемых вами сервисов, по сравнению с одной лишь общей библиотекой классов .NET. Необходимо отметить, что все же лучше использовать стандартные сервисы, предоставляемые общей библиотекой типов.

В этой главе мы подробно обсудим проблемы взаимодействия с сервисами операционной системы. Вы научитесь использовать все типы сервисов, начиная от прямых низкоуровневых вызовов и заканчивая использованием компонентной объектной модели (COM).

11.1. Немного теории

Перед тем как начать обсуждать в деталях работу конкретных механизмов, необходимо определиться, что с чем и как будет взаимодействовать. У нас есть среда .NET, предоставляющая закрытое управляемое пространство, основанное на сборках, и операционная система, предоставляющая свои сервисы посредством DLL- и COM-библиотек (рис. 11.1).

Виртуальная машина .NET изолирует свои приложения от операционной системы и разрешает использовать ее сервисы только по особому допуску. .NET-код, использующий сервисы операционной системы, считается небезопасным, и среда исполнения, по желанию администратора, может запретить исполнение приложений, использующих данную возможность.

Примечание

По умолчанию доступ к сервисам операционной системы для прикладных приложений со стороны виртуальной машины .NET не ограничен.

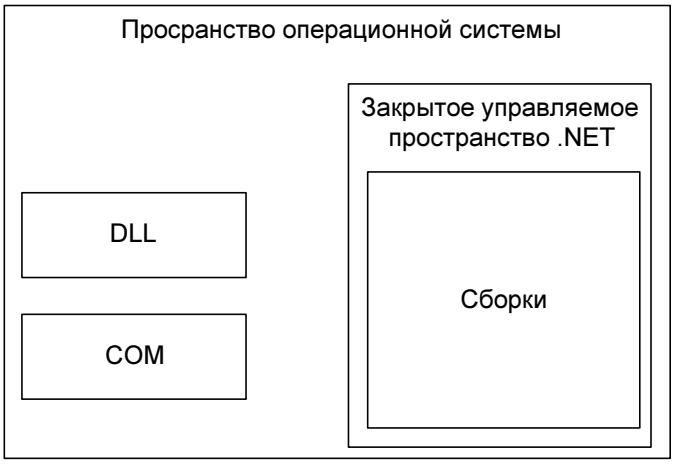


Рис. 11.1. Область взаимодействия между операционной системой и .NET

Таким образом, пространство сервисов операционной системы для .NET-приложений является полностью контролируемым со стороны среды исполнения и может быть закрыто для них в любой момент.

Со стороны операционной системы дело обстоит несколько иначе. Ведь по отношению к ней виртуальная машина .NET является не более чем обычным процессом. Следовательно, доступны любые действия, в рамках сервисов, предоставляемых самой операционной системой: сканирование ее адресного пространства, вызов функций из динамических библиотек, обращение к COM-сервисам и т. п. Этих возможностей оказывается вполне достаточно для налаживания полнофункционального взаимодействия между двумя средами.

Взаимодействие внутри сред

Итак, на уровне операционной системы взаимодействие между ее компонентами осуществляется при помощи динамических библиотек или COM-компонентов (рис. 11.2).

Технология COM построена на базе DLL, поэтому на данной схеме блоки COM располагаются над DLL-блоками. Наиболее образованные читатели могут упрекнуть меня в том, что здесь не указаны средства IPC (Interprocess Communications, Средства межпроцессного взаимодействия). В свое оправдание скажу, что здесь мы будем рассматривать не средства связи между работающими приложениями, а способы предоставления и использования компонентов, таких как COM-объекты, динамические библиотеки и сборки.

Кстати, о сборках: они отвечают за взаимодействие кода в среде .NET. Так или иначе, все сервисы .NET базируются на сборках, они предназначены для хранения кода управляемых приложений (рис. 11.3).

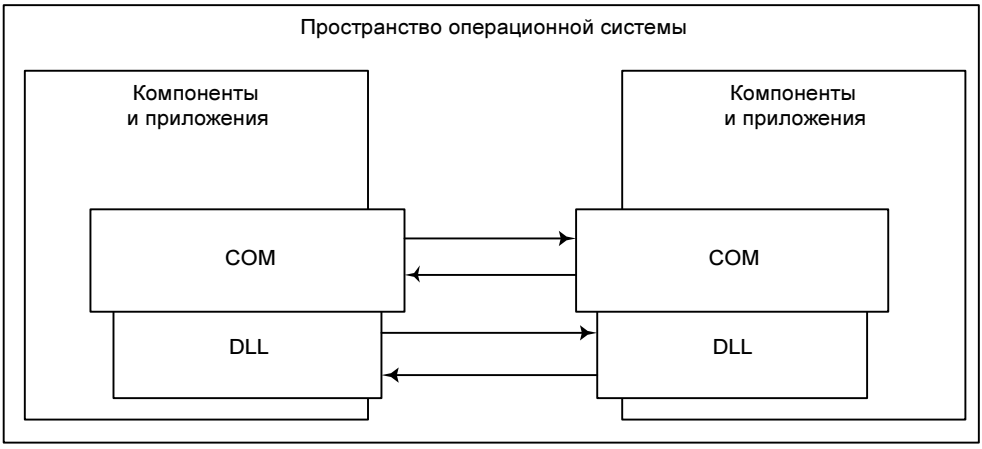


Рис. 11.2. Взаимодействие приложений на уровне операционной системы

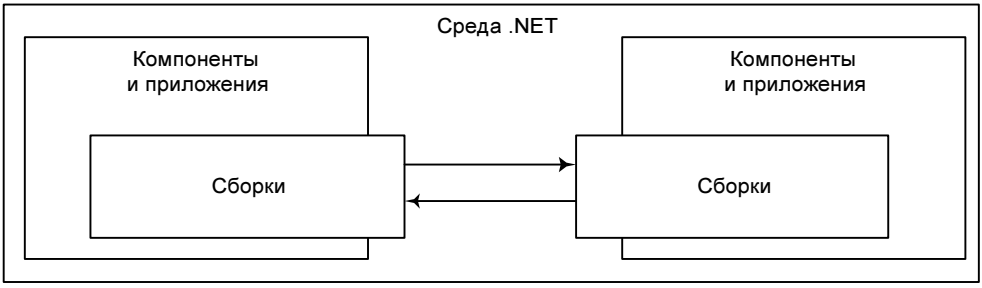


Рис. 11.3. Взаимодействие в среде .NET происходит через сборки

Мы не будем отдельно рассматривать модули, так как они крайне редко используются независимо от сборок.

Взаимодействие .NET-приложений с сервисами ОС

Для обращения к сервисам операционной системы из управляемых приложений используются родные для них механизмы: сборки и сервисы общей библиотеки классов (рис. 11.4).

.NET-программисты могут создавать специальные сборки, которые будут имитировать обычные .NET-приложения, а сами являться "хитрыми" переходниками к сервисам операционной системы.

Примечание

Создание таких сборок полностью автоматизировано при помощи соответствующих утилит.

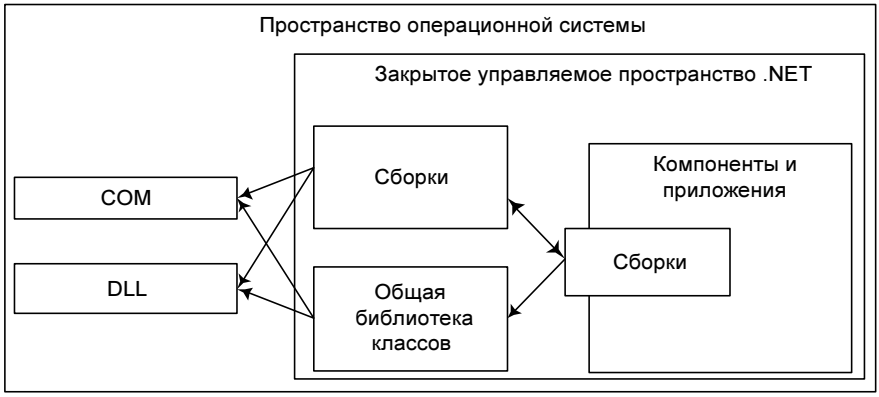


Рис. 11.4. Механизм взаимодействия .NET-кода с сервисами операционной системы

Наряду с этим существует возможность напрямую обращаться к сервисам операционной системы через общую библиотеку классов. Так или иначе, взаимодействие с операционной системой для управляемых приложений выглядит как обращение к обычному .NET-коду, за исключением некоторых нюансов, которые здесь не рассматриваются.

Обращение неуправляемых приложений к средствам .NET

Для операционной системы стандартными средствами разделения кода являются динамические библиотеки и компоненты COM. Соответственно .NET предоставляет шлюзы в свою среду на основе данных технологий (рис. 11.5).

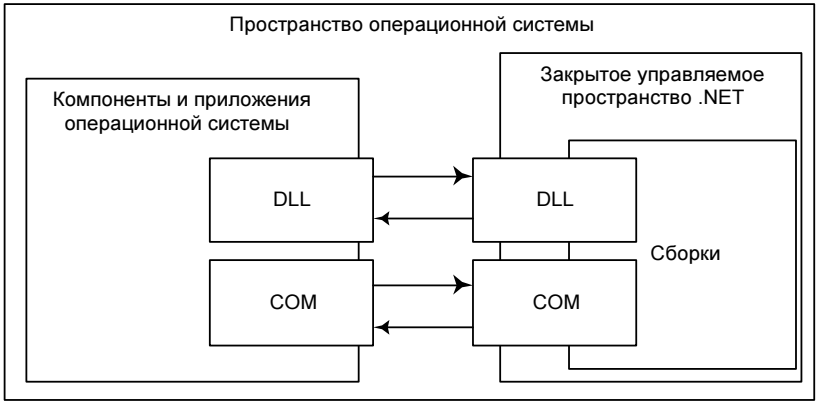


Рис. 11.5. Взаимодействие с .NET-приложениями из среды операционной системы

Вы можете создать специальные DLL или COM компоненты, которые будут обращаться к управляемому коду, но для операционной системы будут неотличимы от обычных. Взаимодействие с ними будет происходить при помощи стандартных API сервисов операционной системы, без учета каких бы то ни было .NET особенностей.

11.2. Приступим к делу

Все сервисы операционной системы, включая COM, так или иначе, базируются на динамических библиотеках. Они являются основой основ операционных систем Windows. Это один из наиболее важных механизмов, на котором построены данные операционные системы. Именно поэтому рассмотрим динамические библиотеки первыми.

Понимание принципов работы и внутреннего устройства динамических библиотек является ключом к использованию технологий более высокого уровня, таких как COM.

11.3. Взаимодействие с DLL из управляемого кода

В данном разделе мы будем использовать функции, импортируемые управляемым приложениями из динамических библиотек. Данная возможность будет продемонстрирована на приложении, которое будет вызывать функцию `MessageBox` из библиотеки `User32.dll`. Напомню, что функция выводит диалоговое окно с указанным пользовательским сообщением, и наверняка знакома большинству читателей.

Для того чтобы осуществить задуманное, придется воспользоваться атрибутом `DllImport`, располагающемся в пространстве имен `System.Runtime.InteropServices`. Его будет необходимо применить к одной из функций нашего класса, а в параметрах атрибута указать имя динамической библиотеки, из которой мы хотим импортировать функцию. Код приложения представлен в листинге 11.1.

Листинг 11.1. Использование динамической библиотеки User32.dll из управляемого кода

```
/*
  Листинг 11.1
  File:   Some.cs
  Author: Дубовцев Алексей
*/
```

```
// Подключим основное пространство имен общей библиотеки классов
using System;
// Подключим пространство имен, отвечающее за поддержку сервисов
// взаимодействия с операционной системой
using System.Runtime.InteropServices;

// Основной класс приложения
class App
{
  // Указываем, что данную функцию необходимо импортировать
  // из динамической библиотеки
  [DllImport("user32.dll")]
  // обратите на внимание на спецификатор extern - он обязателен
  public extern static int MessageBox(Int32 hwnd, string Message,
string Title, Int32 Flags);

  // Точка входа в приложение
  public static void Main()
  {
    // Вызовем функцию, импортированную из динамической библиотеки
    MessageBox(0, "Hello World", "Title of this message",0);
  }
}
```

Скомпилировать данный код можно при помощи команды

```
csc Some.cs
```

После запуска приложения на экране появится окно, представленное на рис. 11.6.

Рис. 11.6. Окно **Title of this message**

Атрибут `DllImport`, который мы применяли к функции, является компиляторным атрибутом. Он не встраивается в конечный файл приложения. Встретив данный атрибут, компилятор считывает его данные и помечает

функцию, к которой он применен специальным модификатором `pinvokeimpl`, а также создает в манифесте запись о связи с динамической библиотекой. Для того чтобы убедиться в этом, посмотрим на IL-код нашего предыдущего примера. Получить его вы можете, воспользовавшись утилитой `ildasm`. Обратите внимание на метод `MessageBox` класса `App`, а также на способ его вызова из метода `Main` (листинг 11.2).

Листинг 11.2. Результат трансляции предыдущего примера в IL-код

```
/*
  Листинг 11.2
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Директива, указывающая, что код данной сборки использует
// динамическую библиотеку user32.dll
.module extern user32.dll
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:3300:0
}
.assembly Some
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Some.exe
// MVID: {3C4E29B9-A4FE-4E89-9564-F9115CFF6D82}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001

.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  // К данному методу мы применяли атрибут DllImport. Как видите,
  // теперь его здесь нет и в помине. Компилятор преобразовал его
```

```
  // в модификатор pinvoke, снабдив его соответствующими параметрами
  .method public hidebysig static pinvokeimpl("user32.dll" winapi)
          int32  MessageBox(int32 hwnd,
                            string Message,
                            string Title,
                            int32 Flags) cil managed preservesig
  {
  }

  .method public hidebysig static void  Main() cil managed
  {
    .entrypoint
    // Code size       19 (0x13)
    .maxstack  4
    // Загрузим в управляемый стек четыре параметра
    ldc.i4.0
    ldstr      "Hello World"
    ldstr      "Title of this message"
    ldc.i4.0
    // Вызов функции, импортированной из DLL, ничем
    // не отличается от обычного.
    call       int32 App::MessageBox(int32,
                                     string,
                                     string,
                                     int32)
    pop
    ret
  }

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  1
    ldarg.0
    call       instance void [mscorlib]System.Object::.ctor()
    ret

}
```

В итоге, оказалось, что атрибут `DllImport` был нужен лишь для того, чтобы передать необходимую информацию компилятору C#, который соответствующим образом модифицировал описание метода, во время его трансляции в IL-байт код.

Внутренний механизм вызова функций

Как мы уже успели убедиться, вызов функций, импортированных из динамических библиотек, ничем не отличается от их управляемых собратьев. Это очевидно из следующего кода:

```
// Загрузим в управляемый стек четыре параметра
// Обратите внимание параметры передаются в прямом порядке
ldc.i4.0
ldstr      "Hello World"
ldstr      "Title of this message"
ldc.i4.0
// Вызов функции, импортированной из DLL, ничем
// не отличается от обычного.
call       int32 App::MessageBox(int32,
                                 string,
                                 string,
                                 int32)
```

При обращении к любой функции среда исполнения определяет, вызывалась ли она ранее или нет. Если нет, будет проведена подготовительная работа. В случае обычной управляемой функции управление передается JIT-компилятору, который преобразует IL-тело функции в машинный код и возвращает указатель на получившийся результат. В нашем случае все будет происходить несколько иначе. Среда исполнения обнаружит спецификатор `pinvokeimpl`, говорящий о двух вещах: функция импортирована из динамической библиотеки и о том, что она не имеет IL-тела. Данная функция, по сути дела, является всего лишь заглушкой, не имеющей реального кода, а предоставляющая лишь необходимую информацию о типах. Следовательно, передавать JIT-компилятору ее бессмысленно — IL-код-то отсутствует! Поэтому среда исполнения создает для нее специальную заглушку межплатформенного вызова. После чего заносит информацию о ней в карту платформенных вызовов. Данная заглушка курируется стандартными сервисами маршалинга (рис. 11.7).

Для управляемого кода функция платформенного вызова будет выглядеть "как родная", но на самом деле это всего лишь заглушка, перенаправляющая вызов к стандартным сервисам маршалинга.

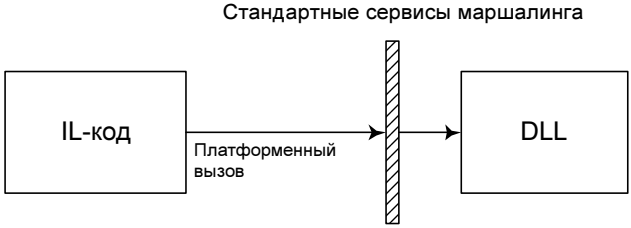


Рис. 11.7. Сервисы маршалинга

Примечание

Основной код, обеспечивающий межплатформенные вызовы, располагается в файлах ndirect.cpp и ndirect.h, в .NET CLI Shared Source. Для более углубленного изучения вопроса вы можете обратиться к этому источнику.

Данные сервисы занимаются передачей управляемых вызовов к реальным функциям на уровень операционной системы. Они отвечают за преобразование и передачу параметров функций, а также контролируют все межплатформенные вызовы. При вызове функции из динамической библиотеки сервисы маршалинга проверяют — использовалась ли данная библиотека ранее, если нет, то производится ее загрузка, а затем импортирование и вызов ее функций. Таким образом, реальная загрузка динамических библиотек происходит непосредственно перед первым вызовом функций, импортируемых из них. Подобная политика загрузки библиотек позволяет существенно ускорить процесс загрузки приложений, а также потенциально экономит ресурсы, поскольку функции могут быть и не вызваны, а тогда и соответствующие им библиотеки не будут загружены.

Но как говорится: ничему нельзя верить на слово! Проверим поведение среды исполнения на примере. Для этого создадим специальную динамическую библиотеку, которая во время загрузки будет выдавать сообщение в виде диалогового окна. А в библиотеке определим функцию `DllMain`, которая вызывается стандартным загрузчиком операционной системы всякий раз при попытке загрузки библиотеки.

Примечание

Даже если библиотека уже загружена, но ее пытаются подключить при помощи функции `LoadLibrary`, загрузчик все равно вызовет функцию `DllMain`, только передаст ей другой управляющий код.

Кроме того, наша библиотека будет экспортировать функцию `SayHello`, выводящую на консоль приветствие, переданное ей пользователем. Прототип функции представлен ниже.

```
void WINAPI SayHello (char* szMessage)
```

Наконец, создадим управляемое приложение, которое по желанию пользователя будет обращаться или не обращаться к нашей динамической библиотеке.

Итак, приступим, код динамической библиотеки вы найдете ниже (листинг 11.3).

Листинг 11.3. Динамическая библиотека, позволяющая отследить свою загрузку

```
/*
  Листинг 11.3
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Подключим стандартный заголовочный файл Windows API
#include <windows.h>

// Точка входа в динамическую библиотеку —
// это та самая функция, которая будет вызываться загрузчиком
// операционной системы
DWORD WINAPI DllMain ( HINSTANCE hProcess, DWORD fdwReason, LPVOID lpvReserved)
{
  // Параметр fdwReason указывает на причину вызова функции
  // нас будет интересовать случай первой загрузки динамической
  // библиотеки в процесс среды исполнения .NET
  if ( fdwReason == DLL_PROCESS_ATTACH )
  {
    // Сообщим пользователю о подключении библиотеки,
    // предложим ему нажать кнопку OK для того, чтобы продолжить
    // загрузку динамической библиотеки
    MessageBox(0,"Press Ok button to load library","Some.dll library↳
DllMain function",0);
    }

    // Сообщим загрузчику, что инициализация библиотеки прошла успешно
    return 1;
}
```

```
// Функция, экспортируемая из нашей библиотеки, именно ее мы будем вызывать
// из нашего управляемого приложения
void WINAPI SayHello (char* szMessage)
{
  // Получим описатель стандартного потока вывода.
  // По умолчанию это консоль.
  HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);

  DWORD dwWrite;
  // Выведем на консоль сообщение, переданное в качестве первого
  // параметра функции
  WriteConsole(hConsole,szMessage,lstrlen(szMessage),&dwWrite,0);
}
```

Для того чтобы заставить линкер создать в таблице экспорта нашей библиотеки запись для функции `SayHello`, нам придется использовать специальный DEF-файл. Его код приведен ниже (листинг 11.4).

Листинг 11.4. Файл, указывающий, какие функции необходимо экспортировать из динамической библиотеки

```
;
; Листинг 11.4
; File:   Some.def
; Author: Дубовцев Алексей
;
LIBRARY Some.dll
EXPORTS
       SayHello
```

После создания обоих файлов можно приступить к сборке динамической библиотеки, сделать это можно при помощи следующих двух команд, обращающихся сначала к компилятору, а затем к линкеру.

```
cl Some.cpp /c
link Some.obj /def:Some.def /dll User32.lib /nodefaultlib Kernel32.lib
/entry:DllMain
```

В итоге должна получиться динамическая библиотека `Some.dll` размером примерно 2.5 Кбайт. Такой размер достигнут за счет отказа от стандартной библиотеки, о чем свидетельствует ключ линкера `/nodefaultlib`. Следующим шагом будет создание управляемого .NET-приложения, которое

и обратится к нашей динамической библиотеке. Его код представлен ниже (листинг 11.5).

Листинг 11.5. Управляемое приложение, использующее динамическую библиотеку

```
/*
  Листинг 11.5
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен
using System;

// Подключим пространство имен, отвечающее за взаимодействие с операционной
// системой
using System.Runtime.InteropServices;

// Подключим пространство имен, отвечающее за поддержку
// пользовательского интерфейса
using System.Windows.Forms;

// Основной класс нашего приложения
class App
{
  // Укажем, что функцию необходимо импортировать из
  // динамической библиотеки Some.dll
  [DllImport("Some.dll")]
  // не забудем указать спецификатор extern, указывающий
  // на то, что функция не будет иметь явного IL-тела
  public extern static void SayHello(String s);

  // Точка входа в приложение
  public static void Main()
  {

    // Запросим пользователя, хочет ли он загружать
    // динамическую библиотеку и вызывать из нее функцию
```

```
    if (MessageBox.Show(
      "Do you want to load Some.dll\n library and call SayHello↵
function?",
      "Question",
      MessageBoxButtons.YesNo,
      MessageBoxIcon.Question ) == DialogResult.Yes)
    {
      // В случае, если пользователь согласится,
      // произведем вызов, тем самым загрузив библиотеку
      SayHello("Hello World");
    }
  }
}
```

Скомпилировать данный пример можно при помощи следующей команды.

```
csc Some.cs
```

В результате запуска приложения на экране появится диалоговое окно, запрашивающее пользователя о том, хочет ли он воспользоваться динамической библиотекой или нет (рис. 11.8).

Рис. 11.8. Запрос на загрузку динамической библиотеки

Если пользователь нажмет кнопку **No**, тогда приложение благополучно завершит свою работу и ничего примечательного не произойдет. В противном случае программа обратится к функции `SayHello`, причем это будет первое обращение к ней. Соответственно среда исполнения определит, что необходима загрузка динамической библиотеки, после чего обратиться к сервису `LoadLibrary`. О чем мы узнаем по диалоговому окну функции `DllMain` нашей библиотеки (рис. 11.9).

После нажатия кнопки библиотека будет загружена и произойдет обращение к функции `SayHello`. О чем мы узнаем, обратившись к консоли нашего приложения, на которую будет выведена строка "Hello, World". Мы достигли желаемого — убедились в том, что загрузка динамических библиотек происходит непосредственно перед вызовом функции. Именно с этой целью в приложение была введена столь хитрая система диалоговых окон.

Рис. 11.9. Диалоговое окно функции `DllMain` нашей библиотеки

Но мы на этом не остановимся, а еще немного поэкспериментируем. Для начала продублируем вызов функции `SayHello`, для того чтобы убедиться в том, что при повторном вызове не происходит никаких посторонних действий.

```
...
{
  // В случае, если пользователь согласится,
  // произведем вызов, тем самым загрузив библиотеку
  SayHello("Hello World");
  // Повторный вызов функции из динамической библиотеки должен
  // произойти моментально
  SayHello("Hello World");
}
...
```

Действительно, второй вызов `SayHello` происходит моментально, по крайней мере, дополнительных обращений к `DllMain` не происходит.

Теперь попробуем предотвратить загрузку нашей библиотеки, для этого возвратим код ошибки из функции `DllMain`. Это будет выглядеть следующим образом, приведем прототип интересующей нас функции (листинг 11.6).

Листинг 11.6. Функция, предотвращающая загрузку динамической библиотеки

```
// Точка входа в динамическую библиотеку —
// это та самая функция, которая будет вызываться загрузчиком
// операционной системы
DWORD WINAPI DllMain ( HINSTANCE hProcess, DWORD fdwReason, LPVOID lpvReserved)
{
  // Параметр fdwReason указывает на причину вызова функции
  // нас будет интересовать случай первой загрузки динамической
  // библиотеки в процесс среды исполнения .NET
  if ( fdwReason == DLL_PROCESS_ATTACH )
  {
    // Сообщим пользователю о подключении библиотеки
```

```
    // предложим ему нажать кнопку OK для того, чтобы продолжить
    // загрузку динамической библиотеки
    MessageBox(0,"Press Ok button to load library","Some.dll library
DllMain function",0);
  }

  // !!!ОБРАТИТЕ ВНИМАНИЕ!!!
  // Сообщим загрузчику о том, что инициализация библиотеки не удалась
  return FALSE;
}
```

В этом случае при попытке загрузки библиотеки среда исполнения выбросит исключение `DllNotFoundException`. И выведет следующую информацию на консоль приложения:

```
Unhandled Exception: System.DllNotFoundException: Unable to load DLL (Some.dll).
   at App.SayHello(String s)
   at App.Main()
```

Если судить строго, информация, предоставляемая средой исполнения по данному исключению, не является верной. Она может ввести пользователя в заблуждение, поскольку то же самое сообщение он увидит, если библиотека будет физически отсутствовать на диске. А здесь же мы имеем принципиально другой случай — библиотека была найдена, то есть файл библиотеки был обнаружен. Она не была загружена лишь по той причине, что не смогла провести собственную инициализацию.

При работе с динамическими библиотеками может возникнуть еще одна неприятная ситуация. Необходимая библиотека вроде бы будет найдена, но запрашиваемой функции в ней не окажется. В этом случае будет выброшено исключение `EntryPointNotFoundException`. Вас не должно смущать название данного исключения, настоящая точка входа в динамическую библиотеку тут не причем (Entry Point, Точка входа). Данное исключение будет выбрасываться для любых не найденных в библиотеке функций, и при этом они вовсе не обязаны являться точками входа в динамическую библиотеку. В итоге мы имеем три вида явных проблем при работе с динамическими библиотеками.

- Библиотека найдена, но не удалось ее загрузить. Функция инициализации динамической библиотеки (`DllMain`) вернула код ошибки. Исключение `FileNotFoundException`.
- Файл динамической библиотеки не был найден. Исключение `FileNotFoundException`.
- Необходимая функция отсутствует в данной динамической библиотеке. Исключение `EntryPointNotFoundException`.

Все три вида проблем являются наиболее благоприятными, поскольку их легко отловить, установив соответствующие обработчики исключений. Бывают проблемы куда хуже. К примеру, если библиотека будет найдена и в ней окажется как бы нужная функция, но это будет вовсе не та библиотека и функция, которая подразумевалась при создании приложения. В такой ситуации работа приложения абсолютно непредсказуема. Здесь можно только посоветовать пользоваться Versioning API, для того чтобы четко контролировать версию загружаемой библиотеки.

Динамическое подключение к библиотекам

В предыдущих разделах мы научились использовать динамические библиотеки. Делали мы это при помощи атрибута `DllImport`, который прекрасно себя зарекомендовал как простое и удобное средство. Единственное неудобство данного метода заключается в том, что мы обязательно должны знать имя функции, на стадии компиляции программы. Мы не можем действительно динамически подключать библиотеки по ходу исполнения нашей программы. И самое ужасное заключается в том, что среда исполнения не предоставляет способов сделать это. В ней попросту не предусмотрено такой возможности. Но нас с вами это не остановит. Далее будет показано, как динамически загрузить заранее неизвестную библиотеку и вызвать из нее произвольную функцию.

Для того, чтобы осуществить задуманное придется воспользоваться технологией отражения и самостоятельно сгенерировать код, который будет обращаться к динамической библиотеке. Этим будет заниматься определенная нами функция `LoadLibrary`, прототип которой приведен ниже

```
public static Delegate LoadLibrary (
  // Имя библиотеки, в которой расположена функция
  String DllName,
  // Имя функции, которую следует импортировать
  String FunctionName,
  // Прототип функции. Должен быть передан в качестве
  // делегата
  Type functionDelegate)
```

Данной функции необходимо передать имя библиотеки и необходимой функции, а также делегат. Он будет необходим для передачи информации о прототипе функции. То есть о количестве ее параметров и типе. После чего функция вернет делегат, вызвав который, можно будет получить доступ к функции в динамической библиотеке. Полный код приложения приведен далее (листинг 11.7).

Листинг 11.7. Пример действительно динамического обращения к DLL-библиотекам из управляемых приложений

```
/*
  Листинг 11.7
  File:   Some.cs
  Author: Дубовцев Алексей
*/


// Подключим основное пространство имен
using System;
// Подключим пространство имен, отвечающее за технологию отражения
using System.Reflection;
using System.Reflection.Emit;
// Здесь расположены сервисы, отвечающие за взаимодействие с операционной
// системой
using System.Runtime.InteropServices;

// Данный класс предоставляет сервисы действительно динамической загрузки
// DLL-библиотек и подключения экспортируемых ими функций
class DynamicDll
{
  // Функция загружает библиотеку и импортирует из нее функцию
  public static Delegate LoadLibrary (
      // Имя библиотеки, в которой расположена функция
      String DllName,
      // Имя функции, которую следует импортировать
      String FunctionName,
      // Прототип функции. Должен быть передан в качестве
      // делегата
      Type functionDelegate)
  {
    // Получим текущий домен приложения
    AppDomain domain = AppDomain.CurrentDomain;

    // Создадим новый объект сборки
    AssemblyName assemblyName = new AssemblyName();
    // Определим имя нашей новой сборки
```

```
assemblyName.Name = "MyDynamicAssembly";

// Создадим новую динамическую сборку в памяти
AssemblyBuilder assemblyBuilder =
  domain.DefineDynamicAssembly(assemblyName,
  // Только для запуска кода
  AssemblyBuilderAccess.Run);

// Создадим в данной сборке модуль, необходимый для
// хранения кода
ModuleBuilder moduleBuilder =
assemblyBuilder.DefineDynamicModule("MyDynamicModule");

// Получим функцию делегата, которая несет прототип необходимой
// нам функции, то есть содержит нужную для маршалинга информацию
// о типах
// Специальный внутренний недокументированный метод Invoke
// служит для обращения к функциям, ссылающимся на делегат
MethodInfo mi = functionDelegate.GetMethod("Invoke");

// Определимся с возвращаемым значением и параметрами функции
Type RetType = mi.ReturnType;

// Запросим типы параметров делегата
// они нам понадобятся для создания собственной функции
ParameterInfo[] paramsInfo = mi.GetParameters();

Type[] parameterType = new Type[paramsInfo.Length];
for (int i = 0; i < parameterType.Length; i++)
  parameterType[i] = paramsInfo[i].ParameterType;


// Определим метод, который будет осуществлять взаимодействие
// с кодом операционной системы
MethodBuilder mb =
moduleBuilder.DefinePInvokeMethod(
          // Имя функции
          FunctionName,
          // Имя библиотеки
```

```
                DllName,
                // Флаги связывания
                MethodAttributes.Public |
                MethodAttributes.Static |
                // Определяет то, что вызов
                // платформенный
                MethodAttributes.PinvokeImpl,
                // Формат вызова для среды .NET
                // стандартный
                CallingConventions.Standard,
                // Тип функции
                RetType,
                // Параметры функции
                parameterType,
                // Формат функции в динамической
                // библиотеке stdcall
                CallingConvention.StdCall,
                CharSet.Ansi);

    // Создаем ранее определенные функции
    moduleBuilder.CreateGlobalFunctions();

    // Получаем созданный метод
    mi = moduleBuilder.GetMethod(FunctionName);

    // Возвращаем делегат, который может быть использован для
    // вызова импортированной функции
    return Delegate.CreateDelegate(functionDelegate,mi);
  }
};

// Опишем прототип функции, которую мы собираемся импортировать из
// динамической библиотеки в качестве делегата
delegate Int32 typedefMessageBox(Int32 hwnd, String Message,
String Title, Int32 Flags);

// Основной класс нашего приложения
class App
{
```

```
  // Точка входа в приложение
  public static void Main()
  {
    // Создадим экземпляр делегата, через который будем
    // осуществлять вызов функции из динамической библиотеки
    typedefMessageBox MessageBox;
    // Загрузим динамическую библиотеку и запросим необходимую
    // нам функцию
    MessageBox = (typedefMessageBox)DynamicDll.LoadLibrary(
      // Имя библиотеки
      "user32.dll",
      // Имя функции
      "MessageBox",
      // Прототип нашей функции, как тип делегата
      typeof(typedefMessageBox));

    // А это вызов нашей функции
    MessageBox(0,"Hello World","Title",0);
  }
}
```

Скомпилировать данное приложение можно при помощи знакомой команды

```
csc Some.cs
```

В результате работы программы на консоль будет выведено обычное диалоговое окно с сообщением "Hello, World". Ничего экстраординарного в этом нет, если, конечно, не учитывать тот факт, что функция, выводящая данное диалоговое окно, была импортирована нами полностью динамически. Основное достоинство данного подхода заключается в том, что нам не надо заранее знать имена необходимой динамической библиотеки и интересующей нас функции.

Что примечательно конечный вызов функции, импортированной из динамической библиотеки, по сути дела ничем не отличался от обычного

```
// А это вызов нашей функции, через делегат
MessageBox(0,"Hello, World","Title",0);
```

Мы воспользовались полученным делегатом как обычной функцией. Благодарить за это в первую очередь надо компилятор языка C#. Он скрыл от нас всю черновую работу, проведенную им по вызову делегата. Он преобразовал простой вызов к следующему виду.

```
castclass  typedefMessageBox
```

```
stloc.0
ldloc.0
ldc.i4.0
ldstr      "Hello, World"
ldstr      "Title"
ldc.i4.0
callvirt   instance int32 typedefMessageBox::Invoke(int32,
                                                    string,
                                                    string,
                                                    int32)
```

В переводе на C# это выглядит примерно так

```
MessageBox.Invoke(0, "Hello, World","Title",0)
```

Вызов метода `Invoke` приводит к обращению к функции, осуществляющей связь с неуправляемым кодом. Заметьте ссылку не на саму функцию `MessageBox`, а на заглушку, отвечающую за осуществление ее вызова. Но для нас этот механизм останется абсолютно прозрачным, поскольку код, отвечающий за это, скрыт глубоко в недрах среды исполнения .NET.

Еще хотелось бы сделать комментарии по поводу класса `DynamicDll`. Он написан в чисто демонстрационных целях и является абсолютно непригодным для использования в реальных приложениях. При подключении нескольких функций он будет неоправданно расходовать ресурсы приложения из-за введения дополнительных динамических сборок и модулей. По-хорошему, он должен создавать лишь одну сборку и уже туда добавлять все новые и новые методы. Также хотелось бы иметь возможность задавать свойства подключаемой функции (формат вызова и кодировку). При необходимости реализация данных возможностей не составит особого труда, поэтому не будем приводить ее здесь.

Использование атрибута *DllImport*

Настало время рассмотреть атрибут `DllImport` во всех подробностях. Далее (табл. 11.1) вы найдете краткие пояснения по каждому члену данного атрибута.

Таблица 11.1. *Описание атрибута* `System.Runtime.InteropServices.DllImport`

Член класса	Описание
`BestFitMapping`	Включает механизм улучшенного преобразования символов из UNICODE в ANSI-кодировку
`CallingConvention`	Определяет формат вызова функции, то есть соглашение о способе передачи ее параметров

Таблица 11.1 *(окончание)*

Член класса	Описание
CharSet	Определяет вид (ANSI, UNICODE) строк для данной функции
EntryPoint	Определяет имя функции, импортируемой из dll
ExactSpelling	Определяет, будет ли среда исполнения пытаться модифицировать имя функции в соответствии с форматом строк указанным в поле CharSet. Данный параметр крайне удобен при использовании функций из Windows API (Здесь имеются в виду постфиксы A и W.)
PreserveSig	Определяет, будет ли преобразован прототип функции с HRESULT возвращаемого явно, в один из дополнительных параметров [out, retval] (В основном используется при взаимодействии с COM.)
SetLastError	Позволяет определить, необходимо ли считывание значение кода ошибки установленного SetLastError
ThrowOnUnmappableChar	Определяет, нужно ли выбрасывать исключение в случае, если при преобразовании из UNICODE кодировки в ANSI возникла проблема и символ пришлось заменить знаком вопроса '?'
Value	Возвращает имя динамической библиотеки

Далее каждый из членов будет рассмотрен более подробно.

Конструктор

```
public DllImportAttribute(
   string dllName
);
```

Конструктор содержит один обязательный позиционный параметр, принимающий имя динамической библиотеки, из которой должно производиться импортирование указанной функции.

Интересной особенностью является тот факт, что имя файла библиотеки может быть заданно без расширения. Среда исполнения самостоятельно добавит его и попытается загрузить библиотеку по новому имени.

DllImport.BestFitMapping

```
public bool BestFitMapping;
```

При вызове функций из динамических библиотек зачастую требуется передавать строки. На уровне управляемого кода в качестве строкового параметра функции мы передаем объект типа `String`. Очевидно, что неуправляемый код не может работать с объектами данного типа, поскольку не имеет о них ни малейшего представления. Для того чтобы сгладить различия между средами, сервисы маршалинга автоматически преобразовывают объект `String` в обычные строки ANSI- или UNICODE-формата. Сам `String` хранит строки в UNICODE-формате. Но при преобразовании из UNICODE в ANSI, могут возникнуть неоднозначности. Возможное количество символов UNICODE (65536) гораздо больше, чем множество символов ANSI (256). Многие символы, например ©, не представлены в ANSI-кодировке и при попытке преобразования вместо них будут использованы знаки вопроса '?'.

Параметр `BestFitMapping` позволяет задействовать интеллектуальные алгоритмы преобразования символов, которые существенно повышают вероятность удачного преобразования.

DllImport.ThrowOnUnmappableChar

```
public bool ThrowOnUnmappableChar;
```

Но все же преобразовать символ UNICODE в ANSI удается далеко не всегда, даже при использовании улучшенных интеллектуальных алгоритмов. Данный параметр атрибута позволяет контролировать процесс преобразования. В случае если символ не удастся преобразовать и значение данного параметра равно `true`, то тогда будет выброшено исключение.

DllImport.CallingConvention

```
public CallingConvention CallingConvention;
```

Данное поле позволяет определить формат передачи параметров импортируемой из динамической библиотеки функции. Эта информация является обязательной — она задает критически важный набор правил, по которым будут передаваться параметры в данную функцию. В качестве возможных значений данного поля могут выступать члены перечисления `CallingConvention` (табл. 11.2). По поводу данного перечисления есть одно маленькое замечание, у него есть очень похожий собрат — перечисление `CallingConventions` (обратите внимание на дополнительную букву `s`, стоящую в конце слова). Оно в отличие от рассматриваемого здесь перечисления располагается в пространстве имен `System.Reflection` и служит для задания формата вызова функций внутри среды исполнения. Будьте внимательны, не спутайте их ненароком!

Таблица 11.2. *Описание перечисления System.Runtime.InteropServices.CallingConvention*

Член перечисления (Флаг)	Описание
Cdecl	Параметры передаются в прямом порядке. Вызывающая функция очищает стек
FastCall	Параметры по возможности передаются через регистры процессора (Данный формат вызова пока не поддерживается средой исполнения.)
StdCall	Параметры передаются через стек в обратном порядке. Функция сама очищает стек (Является стандартом для Windows API.)
ThisCall	Функции неявно передается указатель на объект, в регистре ECX, остальные параметры проталкиваются в стек. Используется для вызова методов классов
Winapi	Стандартный для данной операционной системы формат вызова

Особо хотелось бы отметить флаг `WinApi`, который по сути дела форматом вызова не является. Он указывается на необходимость использования формата вызова функций, являющегося стандартом для той операционной системы, в которой на данный момент исполняется среда .NET.

Программируя для операционных систем Windows NT/9x в качестве стандартного формата вызова функций мы привыкли видеть `stdcall`, но, например, для Windows CE стандартом является `cdecl`.

DllImport.CharSet

```
public CharSet CharSet;
```

Параметр позволяет указать тип строк, используемых функцией. При передаче строковых параметров сервисы стандартного маршалинга опираются на информацию, указанную в данном параметре и производят соответствующее преобразование, между ANSI- и UNICODE-кодировками.

Далее представлено описание членов перечисления `CharSet` (табл. 11.3).

Таблица 11.3. *Описание членов перечисления System.Runtime.InteropServices.CharSet*

Член перечисления (Флаг)	Описание
Ansi	Однобайтовые ANSI-строки
Auto	Тип строк система выбирает автоматически

Таблица 11.3 *(окончание)*

Член перечисления (Флаг)	Описание
None	Если верить документации, то данный флаг давно уже устарел и трактуется как ANSI
Unicode	Двухбайтовые UNICODE-строки

С первого взгляда флаг `Auto` может вызвать довольно сильное удивление. Как это так? Каким же образом среда исполнения сможет определить — какую кодировку необходимо использовать в каждом конкретном случае? Оказывается, она и не будет этого делать для каждого случая. Она выберет кодировку в зависимости от текущей операционной системы. К примеру, для систем класса 9x используется тип строк `Ansi`, и аргументируется это тем, что для данных систем встроенная поддержка `Unicode` отсутствует. А вот для систем класса NT будет сделан безоговорочный выбор в пользу `Unicode`, так как ими она поддерживается на уровне ядра.

DllImport.EntryPoint

```
public string EntryPoint;
```

Данный параметр позволяет в явном виде задать имя функции, импортируемой из динамической библиотеки. При этом появляется одна приятная возможность — можно изменить имя управляемой функции, связанной с импортируемой из динамической библиотеки. К примеру, функцию `MessageBox` можно подключить следующим образом:

```
[DllImport("user32.dll",EntryPoint="MessageBox")]
// обратите внимание на спецификатор extern - он обязателен
public extern static int KukurukuFunction(Int32 hwnd, string Message,
string Title, Int32 Flags);
```

После такого определения можно будет обращаться к функции `MessageBox` через метод `KukurukuFunction`.

Помимо этого, параметр `EntryPoint` позволяет импортировать функции по их порядковым номерам (ординалам). Для этого необходимо задать имя в специальном формате — "#порядковый_номер_функции". В этом случае имя управляемой функции также может быть произвольным. Продемонстрируем на примере, импортировав функцию `MessageBox`, из библиотеки `User32.dll` по ее порядковому номеру (листинг 11.8).

Листинг 11.8. Импортирование функции по ее порядковому номеру, под другим именем

```
/*
  Листинг 11.8
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключаем основное пространство имен общей библиотеки классов
using System;
using System.Runtime.InteropServices;

// Основной класс приложения
class App
{
  // Импортируем функцию из библиотеки User32.dll по ее порядковому
  // номеру
  [DllImport("user32.dll", EntryPoint="#477")]
  // Обратите внимание — имя функции не совпадает с библиотечным
  public extern static Int32 Hehe(Int32 hwnd, string Message,
string Title, Int32 Flags);

  // Точка входа в приложение
  public static void Main()
  {
    // Вызываем импортированную функцию
    Hehe(0,"Hello World","Title of this message",0);
  }
}
```

В результате работы данной программы на экране появится знакомое диалоговое окно с сообщением "Hello, World" и предложением нажать кнопку **OK**.

Если параметр `EntryPoint` не используется, то среда исполнения получает информацию об имени функции из прототипа ее управляемого метода. Если данный параметр указан в явном виде, то имя метода не имеет никакого значения.

DllImport.ExactSpelling

```
public bool ExactSpelling;
```

Данный параметр определяет, будет ли средой исполнения модифицироваться имя функции в зависимости от выбранного типа строк (параметр `CharSet`). Людям, не знакомым с тонкостями API-программирования, это может показаться бредом — зачем и для чего нужно изменять имена функций, да еще в соответствии с кодировкой, используемой для передачи ее строковых параметров. На самом деле все предельно просто: все API-функции от Microsoft используют следующее соглашение о постфиксах.

Примечание

Постфиксом называется одна или несколько букв, стоящих в конце слова, в нашем случае, в конце имени функции.

- Функции, работающие с ANSI-строками, используют постфикс "A".
- Функции, работающие с UNICODE-строками, используют постфикс "W".

Примечание

Постфикс **W** для UNICODE-строк является сокращением от Wide. В переводе на русский Wide означает широкий. В UNICODE-кодировке используется широкий двухбайтовый набор символов

Все API-функции, работающие со строками, представлены в двух вариантах: UNICODE и ANSI, имена которых заканчиваются постфиксом W и A соответственно. При этом необходимо отметить, что имени без постфикса не предоставляется. Для большей ясности разберем данный момент на примере функции `MessageBox`. Большинство программистов привыкло использовать данную функцию в своем коде, обращаясь к ней по имени `MessageBox`. Но мало кто знает, что такой функции в Windows API на самом деле нет. В этом можно легко убедиться при помощи утилиты `dumpbin`, исследовав таблицу экспортируемых функций библиотеки `User32.dll`. Получить необходимую информацию можно при помощи команды:

```
dumpbin User32.dll /exports
```

Ниже приведена интересующая нас часть отчета, выведенного утилитой `dumpbin`.

```
Dump of file user32.dll


File Type: DLL

  Section contains the following exports for USER32.dll
```

```
    ordinal hint RVA      name
              ...
          475  1DA 0004E6CC MenuWindowProcW
          476  1DB 0001EAE6 MessageBeep
          477  1DC 0002ADD7 MessageBoxA
          478  1DD 0002ADFE MessageBoxExA
          479  1DE 00038860 MessageBoxExW
          480  1DF 0002B87C MessageBoxIndirectA
          481  1E0 0002FCE7 MessageBoxIndirectW
          482  1E1 0002AE1B MessageBoxTimeoutA
          483  1E2 0002AD73 MessageBoxTimeoutW
          484  1E3 00038839 MessageBoxW
          485  1E4 0001F570 ModifyMenuA
          486  1E5 0003001C ModifyMenuW
              ...
```

Как нетрудно заметить, библиотека предоставляет две функции `MessageBoxA` и `MessageBoxW`, а функции `MessgaBox` в ней нет. Но как же тогда из кода мы обращаемся к функции по ее неполному имени? Каким образом компилятор выбирает, какую функцию необходимо вызвать?

Ответ достаточно прост. При использовании в своих программах имени `MessageBox`, мы обращаемся не к функции, а к макросу, содержание которого будет зависеть от типа используемой кодировки. Для каждой API- функции, имеющей две версии A и W, в стандартных заголовочных файлах введены специальные макросы, которые позволяют обращаться к ним по укороченному имени.

```
// Реальное описание функций MessageBox, взятое из стандартных
// заголовочных файлов Platform SDK
WINUSERAPI
int
WINAPI
MessageBoxA(
    IN HWND hWnd,
    IN LPCSTR lpText,
    IN LPCSTR lpCaption,
    IN UINT uType);
WINUSERAPI
int
WINAPI
MessageBoxW(
```

```
    IN HWND hWnd,
    IN LPCWSTR lpText,
    IN LPCWSTR lpCaption,
    IN UINT uType);

// Специальный макрос, позволяющий обращаться к функции по неполному имени
#ifdef UNICODE
// Если в программе в качестве основной кодировки выбрана Unicode,
// то используем функцию с постфиксом W
#define MessageBox  MessageBoxW
#else
// Иначе используем Ansi-версию функции
#define MessageBox  MessageBoxA
#endif // !UNICODE
```

Вернемся к параметру `ExactSpelling` атрибута `DllImport`, он позволяет запретить среде исполнения модифицировать имя функции путем добавления постфиксов. Для этого необходимо установить значение данного параметра в `true`. По умолчанию, значение данного параметра равно `false`. Если бы это было не так, то все наши примеры, вызывающие функцию `MessageBox`, попросту бы не работали. Для того чтобы продемонстрировать это, напишем пример, в котором явно запретим среде исполнения экспериментировать с именем нашей функции. Код примера представлен ниже (листинг 11.9).

Листинг 11.9. Пример использования параметра ExactSpelling атрибута DllImport

```
/*
  Листинг 11.9
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки классов
using System;
// Подключим пространство имен, отвечающее за поддержку сервисов
// взаимодействия с операционной системой
using System.Runtime.InteropServices;

// Основной класс приложения
class App
{
```

```
  // Указываем на то, что данная функция импортирована из DLL,
  // запретив модифицировать имя функции, параметр ExactSpelling
  [DllImport("user32.dll", ExactSpelling = true)]
  public extern static int MessageBox(Int32 hwnd, string Message,↳
 string Title, Int32 Flags);

  // Точка входа в приложение
  public static void Main()
  {
    // Вызываем описанную ранее функцию
    MessageBox(0,"Hello World","Title of this message",0);
  }
}
```

При попытке запуска данного приложения среда исполнения сообщит нам о произошедшем исключении `EntryPointNotFoundException`. При этом на консоль будет выдано следующее пояснение:

```
Unhandled Exception: System.EntryPointNotFoundException: Unable to find
an entry point named MessageBox in DLL user32.dll.
   at App.MessageBox(Int32 hwnd, String Message, String Title, Int32
Flags)
   at App.Main()
```

Обратите внимание: среда исполнения не может найти функцию, которую мы, казалось бы, столько раз успешно использовали. Теперь для того чтобы обратиться к данной функции, придется указать ее полное имя — `MessageBoxW`.

DllImport.SetLastError

```
public bool SetLastError;
```

Установка данного параметра равным `true` предписывает сервису стандартного маршалера обновлять значение последнего кода ошибки, установленного неуправляемой функцией при помощи `SetLastError()`. А получить данное значение можно будет при помощи метода `GetLastWin32Error`, класса `Marshal`. Его прототип представлен ниже:

```
public static int Marshal.GetLastWin32Error();
```

Если значение параметра `SetLastError` атрибута `DllImport` будет равно `false`, то при использовании функции `GetLastWin32Error` не рассчитывайте получить правильный результат. Значение последнего кода ошибки обновлено не будет. Продемонстрируем на примере. Для этого создадим динамическую библиотеку, которая будет устанавливать последний код ошибки равным 5. Затем напишем управляемое приложение, которое будет пытаться

получить данный код ошибки, установив параметр `SetLastError` должным образом (`true`). Посмотрим, что из этого выйдет.

Исходный код динамической библиотеки представлен ниже (листинг 11.10).

Листинг 11.10. Динамическая библиотека, изменяющая последний код ошибки после обращения к ней

```
/*
  Листинг 11.10
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Подключим стандартный заголовочный файл Windows API
#include <windows.h>

// Данная функция устанавливает код ошибки. Она будет экспортирована
// из динамической библиотеки
void Error()
{
   // Установим код ошибки
   SetLastError(5);
}
```

Нам также понадобится специальный DEF-файл, в котором будут указаны имена функций, которые необходимо экспортировать (листинг 11.11).

Листинг 11.11. Файл, указывающий, какие из функций должны быть экспортированы из динамической библиотеки

```
;
; Листинг 11.11
; File:   Some.def
; Author: Дубовцев Алексей
;

LIBRARY Some.dll
EXPORTS
  Error
```

Скомпилировать библиотеку можно при помощи двух следующих команд:

```
cl Some.cpp /c
link Some.obj /dll /def:Some.def /nodefaultlib Kernel32.lib /noentry
```

Размер динамической библиотеки должен получиться около двух с половиной Кбайт.

Теперь нам необходимо создать управляемое приложение, которое будет демонстрировать работу с данной динамической библиотекой, а также использовать параметр `SetLastError`. Код .NET-приложения представлен ниже (листинг 11.12).

Листинг 11.12. Управляемое приложение, демонстрирующее работу с параметром SetLastError

```
/*
  Листинг 11.12
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки классов
using System;
// Подключим пространство имен, отвечающее за поддержку сервисов
// взаимодействия с операционной системой
using System.Runtime.InteropServices;

// Основной класс нашего приложения
class App
{
  // Импортируем функцию из динамической библиотеки,
  // при этом не забудем задать параметр SetLastError
  // причем таким образом, чтобы среда исполнения не обновляла
  // его значение при вызове функции
  [DllImport("Some.dll", SetLastError = false)]
  public extern static void Error();

  // Точка входа в приложение
  public static void Main()
  {
    // Вызываем функцию из DLL - она установит код последней ошибки
```

```
    // равным 5
    Error();

    // Выводим информацию о последней ошибке
    Console.WriteLine("LastError = {0} ", Marshal.GetLastWin32Error());
  }
}
```

В результате работы данного приложения я получил код ошибки, равный 203, что явно не соответствовало коду, установленному функцией `Error`. У вас код ошибки может быть совершенно другим, заранее предсказать его очень сложно.

Таким образом, можно сделать вывод — среда исполнения без специального приглашения (параметр `SetLastError`) не будет считывать код ошибки, установленной неуправляемой функцией.

Можно попробовать использовать функцию `GetLastError` напрямую, самостоятельно импортируя ее из библиотеки `Kernel32.dll`. Однако таким способом действительный код ошибки узнать не получается. С чем это связано, я точно сказать не могу. Но можно предположить, что, скорее всего, самостоятельное обращение к функции `GetLastError` искажает код последней ошибки на стадии ее вызова. Продемонстрируем это на примере, модифицировав предыдущий листинг, для прямого обращения к функции `GetLastError`. Его код будет выглядеть так (листинг 11.13).

Листинг 11.13. Попытка получить последний код ошибки напрямую при помощи функции GetLastError

```
/*
  Листинг 11.13
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки классов
using System;
// Подключим пространство имен, отвечающее за поддержку сервисов
// взаимодействия с операционной системой
using System.Runtime.InteropServices;
```

```
// Основной класс нашего приложения
class App
{
  // Импортируем функцию из динамической библиотеки,
  // при этом не забудем задать параметр SetLastError
  // причем таким образом, чтобы среда исполнения не обновляла
  // его значение при вызове функции
  [DllImport("Some.dll", SetLastError = false)]
  public extern static void Error();

  // Напрямую импортируем функцию GetLastError
  [DllImport("Kernel32.dll")]
  public static extern int GetLastError();

  // Точка входа в приложение
  public static void Main()
  {
    // Вызываем функцию из DLL, она установит код последней ошибки
    // равным 5
    Error();

    // Попытаемся вывести информацию о последней ошибке
    Console.WriteLine(GetLastError());
    // Выводим информацию о последней ошибке
    Console.WriteLine("LastError = {0} ",Marshal.GetLastWin32Error());
  }
}
```

В результате работы данного приложения, я получал какие угодно коды ошибок, но только не правильные. Нужного результата мне добиться не удалось, даже установив параметр `SetLastError` равным `false` для самой функции `GetLastError()`.

```
[DllImport("Kernel32.dll", SetLastError = false)]
```

Таким образом, для получения сведений об ошибках, произошедших при работе неуправляемых функций, необходимо пользоваться стандартными средствами, предоставленными в общей библиотеке классов — `Marshal.GetLastWin32Error()`. Другого пути, похоже, нет.

Более сложные случаи взаимодействия

Ранее мы рассматривали взаимодействие с операционной системой только с точки зрения вызова функций. Теперь настало время обратиться к более сложным случаям, которые предполагают использование структур флагов, указателей на функции, а также нестандартных типов.

Передача структур

Некоторым функциям необходимо передавать большое количество параметров. Иногда их количество достигает десятков. Перебрасывать их через стек, как обычные параметры, было бы крайне нежелательно, из-за вероятного сильного снижения производительности. К тому же, этот способ просто неудобен для работы с функциями. Для решения этой проблемы придумали объединять группы параметров в структуры и передавать лишь указатель на первый байт данной структуры. Такой подход избавляет функцию от лишних обращений к стеку и упрощает работу с ее параметрами. К тому же это позволяло модифицировать поведение функции, оставляя при этом совместимость со старыми версиями. Можно было добавлять все новые и новые члены в конец структуры. При этом конфликта со старыми версиями функций не возникает, поскольку они не могут обращаться к данным членам, так как ничего о них не знают. Данный подход широко применялся при проектировании Windows API. Там для организации контроля над версиями функций, используется дополнительный параметр, указывающий размер структуры. А по нему уже вычисляется версия функции, с которой хочет работать приложение.

Примечание

К примеру, можно указать на структуру `STARTUPINFO` — ее первый член `cb` должен обязательно указывать размер структуры.

Указание размеров структур необходимо, чтобы в более новых версиях функций не прочитать лишней паразитной информации, находящейся за границей структуры.

Это была предыстория, а теперь полностью окунемся в проблему с точки зрения программирования для .NET. С определением структур в .NET не возникает, казалось бы, никаких проблем: используем ключевое слово `struct`, объявляем структуру и все. Но оказывается, не все так просто. Основная гадость заключается в том, что среда исполнения .NET при резервировании памяти под структуру, потенциально может нарушить ее целостность. То есть члены структуры могут располагаться в памяти не последовательно, что является в нашем случае необходимым, а могут быть рассортированы средой исполнения на свое усмотрение. Соответственно указатель на структуру теряет всякий смысл, поскольку для доступа к членам структуры используется аддитивная арифметика.

Примечание

Аддитивная арифметика предполагает доступ к членам структуры, при помощи сложения адреса начала структуры со смещением члена внутри стуктуры.

Для наглядности объясним то же самое на примере. В качестве подопытной выберем простейшую структуру `POINT`, используемую в Windows API для представления точки на плоскости, с использованием значений абсциссы (y) и ординаты (x). Определение структуры выглядит так.

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT, *PPOINT;
```

Расположение членов данной структуры в памяти должно быть следующим (рис. 11.10).

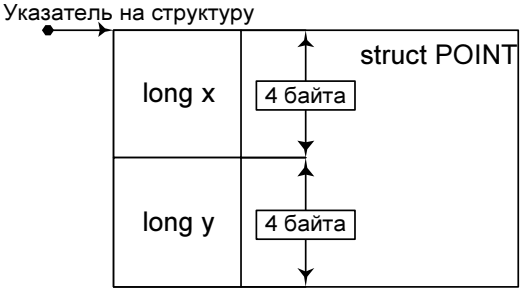


Рис. 11.10. Последовательное расположение членов структуры в памяти, необходимое для работы с ней из неуправляемого кода

Члены структуры должны располагаться в памяти последовательно друг за другом. Это позволяет получить доступ к любому члену структуры, прибавляя смещение члена внутри структуры, к адресу ее начала. Но при описании структур стандартными средствами .NET, они могут терять свою целостность — их члены могут расползтись в памяти непредсказуемым для нас образом (рис. 11.11).

Таким образом, передача указателя на начало структуры в памяти теряет всякий смысл, поскольку данной информации окажется не достаточно для доступа к ее членам.

Правда, возникновение нарушений внутреннего расположения членов структуры в памяти вовсе не обязательно, она может и не потерять своей целостности. Но все же вероятность расщепления структуры достаточно велика, особенно после нескольких сборок мусора, в результате которых рас-

положение отдельных членов структуры может измениться. Поскольку для доступа к членам структуры среда исполнения использует не прямые указатели, а ссылки, динамически обновляющиеся сборщиком мусора, то проблем с использованием членов структуры внутри управляемого кода не возникнет. Чего нельзя сказать о неуправляемом коде, для него подобное поведение среды исполнения окажется фатальным.

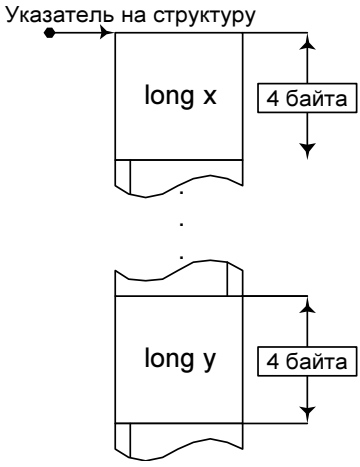


Рис. 11.11. Нарушение последовательного расположения членов структуры в памяти

Для ручного управления распределением членов структур в памяти предназначены два следующих атрибута: `StructLayout` и `FieldOffset`. Первый позволяет задавать общие свойства хранения членов структуры в памяти, а второй действует персонально на указанный член структуры.

Первым делом мы рассмотрим атрибут `StructLayout`, он подробно описан далее (табл. 11.4).

Таблица 11.4. *Описание членов структуры System.Runtime.InteropServices.StructLayout*

Член класса	Описание
`StructLayout(short)`	Позволяет указать вид хранения членов структуры в памяти
`CharSet`	Определяет кодировку, в которую будут преобразованы строки (члены структуры) при маршалинге в неуправляемый код
`Pack`	Определяет выравнивание членов структуры в памяти
`Size`	Указывает общий размер структуры в байтах (Предназначен для использования компиляторами.)

StructLayout.Pack

Данное поле может принимать одно из следующих значений 0, 1, 2, 4, 8, 16, 32, 64 или 128. Где ноль означает выравнивание, принятое по умолчанию для данной системы.

Конструктор

```
public StructLayoutAttribute(
   short layoutKind
);
```

В качестве параметра конструктору должен быть передан один из членов перечисления `LayoutKind`. Описание его членов приведено далее (табл. 11.5).

Таблица 11.5. *Описание членов структуры System.Runtime.InteropSercices.LayoutKind*

Член перечисления (Флаг)	Описание
`Auto`	Среда исполнения будет размещать члены структуры в памяти по своему усмотрению
`Explicit`	Члены структуры будут располагаться в соответствии с информацией, предоставленной при помощи атрибута `FieldOffset`, примененного персонально к каждому члену структуры
`Sequential`	Предписывает среде исполнения размещать члены структуры, в последовательности их описания в исходном коде. В данном случае члены могут быть выровнены при помощи именованного параметра `Pack` атрибута `StructLayout`

Отдельно стоит рассказать про особенности работы члена `Explicit`. Его использование предполагает обязательное применение атрибута `FieldOffset`, к каждому полю класса. Конструктору данного атрибута в качестве параметра необходимо обязательно передать относительное смещение данного поля от начала структуры. Что в свою очередь позволяет создавать объединения, указывая для разных полей одинаковое смещение от начала структуры. Для большей ясности приведем пример (листинг 11.14).

Листинг 11.14. Искусственное создание объединения средствами атрибута FieldOffset

```
/*
  Листинг 11.14
  File:   Some.cs
  Author: Дубовцев Алексей
```

```
*/

// Подключим основное пространство имен общей библиотеки классов
using System;
// Подключим пространство имен, отвечающие за взаимодействие
// с операционной системой
using System.Runtime.InteropServices;

// Опишем структуру, указав при помощи атрибута, что
// будет использоваться ручное выравнивание членов структуры
[StructLayout(LayoutKind.Explicit)]
struct StructU
{
  // Укажем адрес данного члена относительно начала структуры
  [FieldOffset(0)]
  public int i;

  // Для следующего члена укажем адрес так, чтобы он совпадал
  // с первым членом
  [FieldOffset(0)]
  public int u;
};

// Основной класс нашего приложения
class App
{

  // Опишем экземпляр нашей структуры (объединения)
  static StructU st;
  public static void Main()
  {
    // Установим значение члена структуры равным 4, автоматически
    // изменив значение второго члена, так как в памяти они
    // располагаются по одному и тому же адресу
    st.i = 4;

    // Выведем на консоль значения обоих членов нашей структуры
    // (объединения)
    Console.WriteLine("i:{0},u:{1}",st.i,st.u);
  }
};
```

В результате работы данной программы на консоль будет выведена следующая строка

```
i:4,u:4
```

Этот результат ясно показывает, что мы создали настоящее объединение.

Если установить нормальное смещение для второго члена структуры, то фокус не пройдет и значение второго члена будет не определено. Убедиться в этом можно, изменив определение структуры на следующее:

```
[FieldOffset(4)]
public int u;
```

Правда, здесь есть одно *но*. При самостоятельном указании смещения необходимо обязательно учитывать разрядность типов для аппаратной платформы, на которой исполняется приложение. Предыдущее определение будет абсолютно верным для 32-разрядных платформ, поскольку для них тип `int` занимает 32 бита или 4 байта, что одно и то же. А вот для 64-разрядных платформ разрядность типа `int` возрастет до 64 бит (8 байт), соответственно, в этом случае операция с объединением не пройдет.

Внимание

При использовании атрибута `FieldOffset` обязательно учитывайте разрядность типов в соответствии с аппаратной платформой, на которой будет исполняться ваше приложение. Данное предупреждение особенно актуально в свете намечающихся тенденций перехода мировой программной индустрии на 64-битную архитектуру.

Первый и второй члены структуры будут пересекаться в памяти. При этом предсказать зависимость второго члена от первого будет достаточно сложно.

В завершение темы передачи структур, хотелось бы продемонстрировать использование данной технологии на реальном примере. Для демонстрации мы воспользуемся функцией `CreateProcess`, принимающей в качестве параметров указатели на две большие и интересные структуры: `STARTUPINFO` и `PROCESS_INFORMATION`. Исходный код примера вы найдете далее (листинг 11.15).

Листинг 11.15. Пример использования структур при работе с API функцией CreateProcess

```
/*
  Листинг 11.15
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключаем основное пространство имен общей библиотеки классов
```

```
using System;

// Подключаем пространство имен, отвечающее за взаимодействие
// с операционной системой
using System.Runtime.InteropServices;

// Подключаем пространство имен, отвечающее за предоставление
// пользовательского интерфейса
using System.Windows.Forms;

// Опишем структуры, которые необходимы для использования функции
// CreateProcess

// Определим для данной структуры последовательное выравнивание
// А также Unicode-формат передачи строк
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
struct STARTUPINFO
{
  // В комментариях указаны определения членов на языке C
  //DWORD   cb;
  public UInt32 cd;
  //LPTSTR  lpReserved;
  public String lpReserved;
  //LPTSTR  lpDesktop;
  public String lpDesktop;
  //LPTSTR  lpTitle;
  public String lpTitle;
  //DWORD   dwX;
  public UInt32 dwX;
  //DWORD   dwY;
  public UInt32 dwY;
  //DWORD   dwXSize;
  public UInt32 dwXSize;
  //DWORD   dwYSize;
  public UInt32 dwYSize;
  //DWORD   dwXCountChars;
  public UInt32 dwXCountChars;
  //DWORD   dwYCountChars;
  public UInt32 dwYCountChars;
```

```
  /DWORD   dwFillAttribute;
  ublic UInt32  dwFillAttribute;
  //DWORD   dwFlags;
  public UInt32 dwFlags;
  //WORD    wShowWindow;
  public UInt16  wShowWindow;
  //WORD    cbReserved2;
  public UInt16  cbReserved2;
  //LPBYTE  lpReserved2;
  public UInt32 lpReserved2;
  //HANDLE  hStdInput;
  public UInt32 hStdInput;
  //HANDLE  hStdOutput;
  public UInt32  hStdOutput;
  //HANDLE  hStdError;
  public UInt32  hStdError;
};

// Определим последовательное выравнивание,
// формат строк определять не надо, поскольку в данной структуре
// не представлено ни одного указателя на строку
[StructLayout(LayoutKind.Sequential)]
struct PROCESS_INFORMATION
{
  //HANDLE hProcess;
  public UInt32 hProcess;
  //HANDLE hThread;
  public UInt32 hThread;
  //DWORD dwProcessId;
  public UInt32 dwProcessId;
  //DWORD dwThreadId;
  public UInt32 dwThreadId;
};

// Определим класс, который будет содержать определения необходимых
// нам импортируемых функций
class Win32API
{
  // Здесь в комментариях специально оставлены "родные" описания
```

```
  // параметров функции на языке C, для того чтобы можно
  // было сравнить их с типами, используемыми в управляемой среде
  // .NET

  // Импортируем функцию CreateProcess, которая предназначена
  // для создания процесса.
  // Будем использовать Unicode-версию функции
  [DllImport("kernel32.dll", EntryPoint = "CreateProcessW",↳
CharSet = CharSet.Unicode)]
  public static extern Int32 CreateProcess(
    // Имя файла, в котором расположено приложение
    //LPCTSTR lpApplicationName,
    String lpApplicationName,
    // Командная строка для передачи приложению
    //LPTSTR lpCommandLine,
    String lpCommandLine,
    // Атрибуты защиты процесса
    //LPSECURITY_ATTRIBUTES lpProcessAttributes,
    UInt32 lpProcessAttributes,
    // Атрибуты защиты основного потока приложения
    //LPSECURITY_ATTRIBUTES lpThreadAttributes,
    UInt32 lpThreadAttributes,
    // Опция наследования описателей базового процесса
    //BOOL bInheritHandles,
    UInt32 bInheritHandles,
    // Флаги создания
    //DWORD dwCreationFlags,
    UInt32 dwCreationFlags,
    // Блок окружения
    //LPVOID lpEnvironment,
    UInt32 lpEnvironment,
    // Имя текущей директории
    //LPCTSTR lpCurrentDirectory,
    String lpCurrentDirectory,
    // Стартовая информация приложения
    //LPSTARTUPINFO lpStartupInfo,
    ref STARTUPINFO lpStartupInfo,
    // Информация о стартовавшем потоке
    //LPPROCESS_INFORMATION lpProcessInformation
```

```
    ref PROCESS_INFORMATION lpProcessInformation);

  // Импортируем функцию TerminateProcess, которая предназначена
  // для принудительного завершения работы процесса
  [DllImport("kernel32.dll", EntryPoint = "TerminateProcess")]
  public static extern Int32 TerminateProcess(
    // handle to the process
    //HANDLE hProcess,
    UInt32 hProcess,
    // exit code for the process
    //UINT uExitCode
    UInt32 uExitCode
  );

};

// Основной класс приложения
class App
{
  // Точка входа в приложение
  public static void Main()
  {
    // Создаем по экземпляру каждой из структур, описанных нами выше
    STARTUPINFO sti = new STARTUPINFO();
    PROCESS_INFORMATION pi = new PROCESS_INFORMATION();
    // Запускаем процесс
    // в качестве тестового процесса выберем Блокнот
    if (Win32API.CreateProcess(
        "c:\\windows\\notepad.exe",
        null,0,0,0,0,0,null,ref sti, ref pi) == 1)
    {
      // Если запуск процесса удался, то предложим
      // нажать Ok для того, чтобы завершить его работу
      MessageBox.Show("Ckick ok to terminate application");
      // Принудительно завершаем работу приложения
      Win32API.TerminateProcess(pi.hProcess,0);
    }
  }
};
```

В результате работы данного приложения будет запущен Блокнот, после чего пользователь увидит диалоговое окно с предложением закрыть приложение (рис. 11.12).

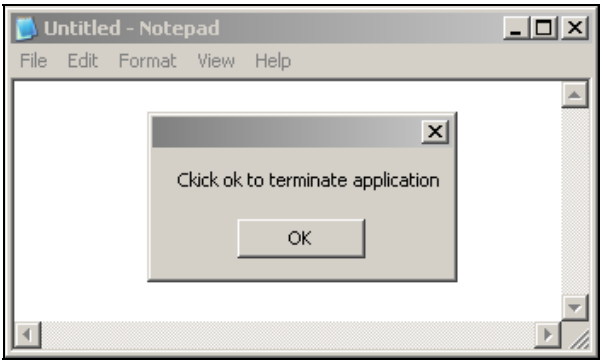


Рис. 11.12. Блокнот, запущенный из управляемого .NET-приложения

Передача управляемых структур в неуправляемый код предельно проста, пока они используются как обычные параметры для пересылки информации. Но если требуется оставить структуру в распоряжении неуправляемого кода, на продолжительное время, а не для однократного вызова — придется подумать о механизмах закрепления ее в памяти. Они должны обеспечить, что сборщик мусора не передвинет или же вообще удалит ненароком, управляемую структуру.

Использование флагов

Для большинства неуправляемых функций, включая Windows API, в качестве параметров, определяющих их поведение, принято передавать флаги. Флаги являются числами, которые рассматриваются как группы битов. Состояние битов и будет определять поведение функции. Для примера можно привести знакомую нам функцию `CreateProcess`, ее прототип представлен ниже:

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  // Данный параметр принимает флаг, в котором при помощи битов указаны
  // требования к поведению функции. Точнее, рассматриваемый
  // параметр определяет дополнительные свойства создаваемого процесса
```

```
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Нас будет интересовать шестой параметр данной функции. Он предполагает передачу флагов, которые указывают дополнительные свойства создаваемых процессов.

Приведем описание значений всех этих флагов, используя синтаксис языка C# (листинг 11.16).

Листинг 11.16. Определение флагов в управляемом коде, с помощью атрибута Flags

```
// Укажем, что это не просто перечисление, а флаги
// компилятор, обнаружив атрибут Flags, разрешит использование
// битовых операций для членов данного перечисления
[Flags]
enum CreationFlags
{
 // 01
 DEBUG_PROCESS                    = 0x00000001,
 // 10
 DEBUG_ONLY_THIS_PROCESS          = 0x00000002,
 // 100

 CREATE_SUSPENDED                 = 0x00000004,
 // 1000
 DETACHED_PROCESS                 = 0x00000008,

 // 10000
 CREATE_NEW_CONSOLE               = 0x00000010,

 // 1000000000
 CREATE_NEW_PROCESS_GROUP         = 0x00000200,
 // 10000000000
 CREATE_UNICODE_ENVIRONMENT       = 0x00000400,
```

```
 // 100000000000
 CREATE_SEPARATE_WOW_VDM          = 0x00000800,
 // 1000000000000
 CREATE_SHARED_WOW_VDM            = 0x00001000,
 // 10000000000000
 CREATE_FORCEDOS                  = 0x00002000,

 // 100000000000000000000000
 CREATE_BREAKAWAY_FROM_JOB        = 0x01000000,
 // 1000000000000000000000000
 CREATE_PRESERVE_CODE_AUTHZ_LEVEL = 0x02000000,

 // 10000000000000000000000000
 CREATE_DEFAULT_ERROR_MODE        = 0x04000000,
 // 100000000000000000000000000
 CREATE_NO_WINDOW                 = 0x08000000,
};
```

Ключевым моментом в понимании использования флагов является тот факт, что каждый флаг задает определенный бит. Соответственно, если все флаги определяют различные биты, то появляется возможность комбинировать их в одном-единственном числе, при помощи булевой операции "или".

Далее (табл. 11.6) приведены соответствия битов и шестнадцатеричных значений флагов, описанных выше.

Таблица 11.6. *Соответствие шестнадцатеричных значений флагов и соответствующих им битов*

Шестнадцатеричное значение	Двоичное представление
0x00000001	00000000000000000000000000000001
0x00000002	00000000000000000000000000000010
0x00000004	00000000000000000000000000000100
0x00000008	00000000000000000000000000001000
0x00000010	00000000000000000000000000010000
0x00000200	00000000000000000000001000000000
0x00000400	00000000000000000000010000000000
0x00000800	00000000000000000000100000000000
0x00001000	00000000000000000001000000000000

Таблица 11.6 *(окончание)*

Шестнадцатеричное значение	Двоичное представление
0x00002000	00000000000000100000000000000
0x01000000	00010000000000000000000000000
0x02000000	00100000000000000000000000000
0x04000000	01000000000000000000000000000
0x08000000	10000000000000000000000000000

Как нетрудно заметить, числа подобраны таким образом, чтобы каждое из них задавало определенный бит и они не пересекались. Чтобы добиться этого, необходимо знать правила двоичной арифметики, а при определении флагов нужно пересчитывать каждое число... или же прибегнуть к следующему трюку. Можно положить первый элемент перечисления равным единице, а для остальных — последовательно провести операцию битового сдвига. Продемонстрируем это на примере (листинг 11.17).

Листинг 11.17. Определение флагов при помощи операции битового сдвига

```
// Определим флаги
[Flags]
enum MyFlags
{
  // В комментариях указаны значения битов для каждого флага
  // в целях экономии места, указаны только младшие 8 битов,
  // приведение остальных в данном случае не имеет смысла,
  // поскольку они всегда будут равны нулю

  //...-0000-0001
  Flag1 = (1<<0),

  //...-0000-0010
  Flag2 = (1<<1),

  //...-0000-0100
  Flag3 = (1<<2),

  //...-0000-1000
  Flag4 = (1<<3),
```

```
  //...-0001-0000
  Flag5 = (1<<4),

  //...-0010-0000
  Flag6 = (1<<5),

  //...-0100-0000
  Flag7 = (1<<6),

  //...-1000-0000
  Flag8 = (1<<7),
};
```

При использовании перечислений для задания флагов, следует помнить, что по умолчанию используется тип `Int32`. Соответственно, количество битов (битовых сдвигов) не должно превышать тридцати двух, иначе неминуемы совпадения. Если требуется большее количество флагов, необходимо изменить базовый тип для перечисления. Делается это так:

```
enum MyFlags: base-type
```

В качестве `base-type` обязательно должен быть указан один из следующих типов: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`.

В качестве альтернативы, при необходимости определить большее, чем по умолчанию, количество флагов, можно использовать несколько перечислений. Это позволит избежать путаницы при их использовании.

Передача указателей на функции

Некоторые функции Windows API имеют в качестве одного из своих параметров указатель на пользовательскую функцию. Обращение к данной функции будет происходить прямо из недр механизмов Windows API, с передачей ей всей необходимой информации или же наоборот получения некоторой информации от пользовательской функции. Данный подход называется — функции обратного вызова (Callback). К сожалению, в русском языке не нашлось адекватного термина, поэтому мне пришлось использовать его дословный перевод. Но как говорит старинная русская поговорка: хоть горшком назови, только в печку не ставь.

Использование функций обратного вызова очень удобно, когда требуется передать большие объемы информации заранее неизвестного вида. Данный механизм позволяет отказаться от введения специализированных коллекций, которые, кстати, могут быть неудобны для использования в приложениях.

При использовании функций обратного вызова, внешней программе нужно будет лишь вызвать пользовательскую функцию для каждого члена коллекции, которую мы хотим передать.

Для примера можно рассмотреть функцию `EnumWindows`, предоставляющую пользователю информацию обо всех зарегистрированных в системе окнах.

```
BOOL EnumWindows(
    // Указатель на пользовательскую функцию
    WNDENUMPROC lpEnumFunc,
    // Дополнительный параметр, передаваемый пользовательской функции
    LPARAM lParam
);
```

В качестве первого параметра должен быть передан указатель на пользовательскую функцию, которая должна иметь следующий прототип.

```
BOOL CALLBACK EnumWindowsProc(
       // Описатель окна
   HWND hwnd,
   // Дополнительный параметр, который был передан функции EnumWindows
   LPARAM lParam
);
```

Данная функция будет вызвана для каждого зарегистрированного в системе окна, где в первом параметре будет передан его описатель.

Если бы функцией `EnumWindows` не использовалась технология функций обратного вызова, то пришлось бы передавать массив описателей, а затем почленно их разбирать. Кроме этого, необходимо было бы удалять данный массив, чтобы избежать утечки памяти. В данном случае, трудно сказать, какой из подходов лучше, но метод обратного вызова явно лаконичнее.

Вернемся к управляемому коду. Для использования технологии обратного вызова требуется передать указатель на управляемую функцию, которая будет вызываться из неуправляемого кода. Здесь есть две проблемы. Во-первых, управляемая среда не поддерживает указателей на функции. А во-вторых, между управляемыми и неуправляемыми функциями наблюдаются существенные архитектурные различия (организация стека, передача параметров, представление типов), которые не позволят напрямую обращаться к управляемому коду из среды операционной системы. С первым препятствием легко справиться, используя косвенные ссылки на управляемые функции — делегаты. Вторая проблема решается самой средой исполнения, прозрачно для программистов. Она создаст специальную заглушку-переходник, на которую передаст указатель в неуправляемое пространство (рис. 11.13), причем сделано все это будет полностью автоматически.

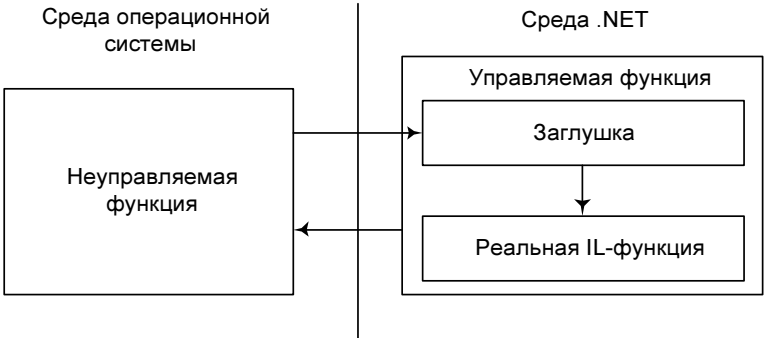


Рис. 11.13. Механизм обращения к управляемости функций из среды ОС

Код операционной системы будет обращаться к заглушке, которая в свою очередь, после преобразования параметров, будет обращаться уже к реальному IL-коду. Помимо передачи параметров заглушка занимается возвращением значения вызванной функции. Таким образом, из неуправляемой среды функция выглядит как единое целое, и вызов происходит в штатном режиме.

Продемонстрируем данную технологию примером. Он будет обращаться к функции `EnumWindows`, предоставляя ей указатель на свою управляемую функцию. Для реализации задуманного нам придется ввести делегат, соответствующий прототипу функции обратного вызова.

```
delegate bool EnumWindowsCallBack(int hwnd, int lParam);
```

Напомню, что сам делегат является только типом и не более. Для того чтобы его использовать, необходимо создать экземпляр данного типа, ссылающийся на нашу управляемую функцию.

```
EnumWindowsCallBack refToReportFunction = new EnumWindowsCallBack(Report);
```

Это ссылка на управляемую функцию, которую мы можем передать в потусторонний (неуправляемый) код. Среда исполнения при использовании делегата автоматически создаст заглушку и передаст указатель на нее неуправляемому коду (листинг 11.18).

Листинг 11.18. Пример взаимодействия с неуправляемым кодом, с использованием технологий обратного вызова

```
/*
  Листинг 11.18
  File:   Some.cs
  Author: Дубовцев Алексей
*/
```

```
// Подключим основное пространство имен общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за взаимодействие
// с операционной системой
using System.Runtime.InteropServices;

// Так выглядит описание необходимых нам функций в Platform SDK
/*
WNDENUMPROC
BOOL CALLBACK EnumWindowsProc(
    // Описатель окна
    HWND hwnd,
    // Значение, определенное приложением
    LPARAM lParam
);*/

// Введем делегат, соответствующий прототипу функции обратного вызова
delegate bool EnumWindowsCallBack(int hwnd, int lParam);

// Основной класс нашего приложения
class App
{
  // Импортируем необходимые функции
  // Функция, предоставляющая описатели всех зарегистрированных в системе
  // окон
  [DllImport("user32.dll")]
  public static extern int EnumWindows(
    // Функция обратного вызова
    //WNDENUMPROC lpEnumFunc,
    EnumWindowsCallBack function,
    // Параметр для передачи нашей функции обратного вызова
    // LPARAM lParam
    int lParam);

  [DllImport("user32.dll", CharSet = CharSet.Unicode)]
   // Функция позволяет узнать класс окна, по его описателю
  public static extern int GetClassNameW(
    // Описатель окна
```

```
    //HWND hWnd,
    int hWnd,
    // Имя класса окна
    //LPTSTR lpClassName,
    String lpClassName,
    // Размер буфера для передачи класса окна
    //int nMaxCount
    int nMaxCount
  );

  // Точка входа в приложение
  public static void Main()
  {
    // Создадим делегат, ссылающийся на функцию Report,
    // расположенную ниже по классу
    EnumWindowsCallBack refToReportFunction =
        new EnumWindowsCallBack(Report);

      // Вызовем API-функцию, перечисляющую описатели окон,
      // путем обращения к пользовательской функции
      // обратного вызова
         EnumWindows(refToReportFunction,0);
  }

  // Управляемая функция обратного вызова, именно она будет
  // вызываться из недр неуправляемого кода
  public static bool Report(int hwnd, int lParam)
  {
    // Создадим строку заданного (100 широких двухбайтных
    // символа) размера
    String strWndClass = new String(new char[100]);
    // Запросим имя класса окна по его описателю
    GetClassNameW(hwnd,strWndClass,100);
    // Выведем на консоль имя класса
    Console.WriteLine(strWndClass);
    // Сообщим о желании продолжать работу
    return true;
  }
}
```

В результате работы данного приложения на консоль будут выведены следующие строки.

```
BaseBar
WorkerW
SysFader
OfficeTooltip
tooltips_class32
CiceroUIWndFrame
tooltips_class32
tooltips_class32
tooltips_class32
tooltips_class32
tooltips_class32
tooltips_class32
Shell_TrayWnd
CiceroUIWndFrame
CiceroUIWndFrame
ALSMTrayClass
tooltips_class32
ConsoleWindowClass
MsoCommandBarPopup
MsoCommandBarShadow
MsoCommandBarShadow
MsoCommandBarShadow
OpusApp
ActiveClipboard
Connections Tray
SystemTray_Main
OpusApp
MS_WebcheckMonitor
OleDdeWndClass
MsoStdCompMgr
Afx:00400000:0
#32770
AcrobatTrayIcon
CTrayIconWndClass
NVMediaCenter
RunDLL
LV8_Launcher
```

```
WorkerW
WorkerW
DDEMLEvent
DDEMLMom
NVSVCPMMWindowClass
tooltips_class32
CicLoaderWndClass
GDI+ Hook Window Class
Progman
```

Как нетрудно увидеть — это названия классов всех зарегистрированных в системе окон. Очевидно, что результат будет зависеть от каждой конкретной системы, а также от количества и типа запущенных приложений.

При использовании функций обратного вызова, будьте внимательны с исключениями. Если ваша функция будет генерировать исключения, то они могут быть перехвачены неуправляемым кодом, поскольку они базируются на SEH. Нормально обработать такие исключения неуправляемый код, скорее всего, не сможет, и до вас они тоже не дойдут. Соответственно, логика приложения будет нарушена, что может привести к плачевным последствиям.

Как видите, применение функций обратного вызова до примитивности просто. Главное, быть внимательным при определении прототипа делегата.

Соответствие неуправляемых и управляемых типов

При написании кода, взаимодействующего с неуправляемым кодом, наиболее важным моментом является точное соответствие типов. Далее (табл. 11.7) приведено соответствие управляемых и "родных" для операционной системы типов. Кроме действительных имен типов, для наглядности, приведены и макросы WinAPI.

Таблица 11.7. *Соответствие управляемых и "родных" для операционной системы типов*

Тип WinAPI	Тип данных языка C(++)	Управляемый тип	Дополнительное описание (разрядность)
HANDLE	void *	System.IntPtr	32 бита
BYTE	unsigned char	System.Byte	8 бит
SHORT	Short	System.Int16	16 бит
WORD	unsigned short	System.UInt32	16 бит
INT	Int	System.Int32	32 бита

Таблица 11.7 *(окончание)*

Тип WinAPI	Тип данных языка C(++)	Управляемый тип	Дополнительное описание (разрядность)
UINT	unsigned int	System.UInt32	32 бита
LONG	Long	System.Int32	32 бита
BOOL	Long	System.Int32	32 бита
DWORD	unsigned long	System.UInt32	32 бита
ULONG	unsigned long	System.UInt32	32 бита
CHAR	Char	System.Char	8 бит (символ ANSI)
LPSTR	char *	System.String или System.StringBuilder	Строка ANSI
LPCSTR	const char *	System.String или System.StringBuilder	Строка ANSI
LPWSTR	Wchar_t *	System.String или System.StringBuilder	Строка UNICODE
LPCWSTR	const wchar_t *	System.String или System.StringBuilder	Строка UNICODE
FLOAT	Float	System.Single	32 бита
DOUBLE	Double	System.Double	64 бита

Данная система правил используется по умолчанию, но необходимо отметить, что вы имеете возможность изменить поведение среды исполнения при преобразовании управляемых типов. Более подробно об этом будет рассказано позднее.

11.4. Взаимодействие с .NET через DLL

Теперь мы рассмотрим задачу, обратную рассмотренной в предыдущем разделе. Мы будем взаимодействовать с управляемым .NET-кодом, посредством обычных динамических библиотек, которые будут служить шлюзом в среду .NET. Мы создадим динамическую библиотеку, код которой будет полностью управляемым. Затем мы обратимся к функции, экспортируемой данной библиотекой из обычного, "родного" для данной операционной системы кода.

К сожалению, создать подобную динамическую библиотеку можно только при помощи языка MC++. Компиляторы других языков не обладают необ-

ходимыми в данном случае возможностями. Здесь имеются в виду компиляторы, входящие в стандартную поставку .NET Framework.

Если ранее вы создавали динамические библиотеки при помощи компилятора C++ от Microsoft, то наверняка заметите, что процессы создания очень похожи. Ниже приведен код динамической библиотеки (листинг 11.19).

Листинг 11.19. Динамическая библиотека, построенная при помощи технологий .NET

```
/*
  Листинг 11.19
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Укажем на связь со сборкой общей библиотеки классов
#using <mscorlib.dll>

// Подключим основное пространство имен общей библиотеки классов
using namespace System;

// Данная функция будет экспортирована из динамической
// библиотеки, как обычная неуправляемая.
// Она выводит на консоль сообщение, переданное ей в виде
// Unicode-строки
void SayHello( wchar_t* message )
{
  // Выводим на консоль строку
  Console::WriteLine((String*)message);
}
```

Помимо исходного кода, необходимо также ввести DEF-файл, в котором будут указаны экспортируемые из библиотеки функции (листинг 11.20).

Листинг 11.20. DEF-файл, указывающий функции, которые необходимо экспортировать из динамической библиотеки

```
;
;  Листинг 11.20
;  File:   Some.def
```

```
;  Authod: Дубовцев Алексей
;

LIBRARY Hello.dll
EXPORTS
  SayHello
```

Теперь нам предстоит самый ответственный момент: компиляция нашей библиотеки. Сделать это можно при помощи двух следующих команд.

```
cl Some.cpp /c /clr /Zc:wchar_t
link Some.obj /dll /def:Some.def mscoree.lib /out:Hello.dll /nodefaultlib
/noentry
```

Обратите внимание на параметр `/clr`, он указывает компилятору на то, что используется управляемый код. Параметр `/Zc:wchar_t` предписывает компилятору использовать встроенный тип `wchar_t`, а не его макроопределение.

Размер динамической библиотеки должен получиться немногим более четырех Кбайт.

Теперь необходимо создать обычное неуправляемое приложение, которое будет использовать нашу "хитрую" библиотеку. Данная программа будет написана на обычном C++ (листинг 11.21). Использовать ее мы будем в статическом режиме, воспользовавшись для связывания файлом Hello.lib, созданным линкером на этапе компиляции библиотеки.

Листинг 11.21. Неуправляемое приложение, косвенно обращающееся к управляемому коду через специальную динамическую библиотеку

```
/*
  Листинг 11.21
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Подключим стандартный заголовочный файл Windows API
#include <windows.h>

// Опишем прототип импортируемой нами функции
// по-хорошему, конечно, мы должны были это сделать
// в отдельном заголовочном файле, указывающем на
// принадлежность данной функции к конкретной библиотеке.
// Но мне крайне не хотелось разбивать данный пример на
```

```
// несколько файлов, от этого он бы только потерял
// наглядность
void SayHello(wchar_t* param);

// Точка входа в наше приложение
void main()
{
  // Вызовем функцию, импортируемую из динамической библиотеки
  SayHello((wchar_t*)L"Hello World");
}
```

Скомпилировать данный пример можно при помощи двух следующих команд.

```
cl Some.cpp /c /Zc:wchar_t
link Some.obj Hello.lib
```

Обратите внимание на библиотеку Hello.lib, подключаемую при линковке, она укажет на связь с библиотекой Hello.dll.

Для того чтобы снять сомнения по поводу самодостаточности библиотеки Hello.dll, которые могут возникнуть у некоторых читателей, в связи с использованием библиотеки Hello.dll, приведем пример загрузки и использования динамической библиотеки, полностью в ручном режиме (листинг 11.22).

Листинг 11.22. Использование библиотеки-шлюза в управляемый код полностью в динамическом режиме

```
/*
  Листинг 11.22
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Подключим стандартный заголовочный файл Windows API
#include <windows.h>

// Опишем тип указателя на функцию, которую мы будем импортировать
// из динамической библиотеки
typedef void (*SayHelloPointerType) (wchar_t* param);

// Точка входа в приложение
void main()
```

```
{
  // Описатель библиотеки, которую мы будем загружать
  HANDLE hLibrary;

  // Загрузим нашу библиотеку
  hLibrary = LoadLibrary("Hello.dll");

  // Объявим указатель на функцию
  SayHelloPointerType pSayHello;

  // Импортируем функцию из динамической библиотеки
  pSayHello =↳
(SayHelloPointerType)GetProcAddress((HMODULE)hLibrary,"SayHello");

  // Вызовем функцию стандартными средствами языка C++
  pSayHello(L"Hello World");

  // А теперь обратимся к функции на чистом ассемблере,
  // дабы уверить читателя в том, что не используется
  // каких-либо скрытых сервисов
  DWORD p = (DWORD)(DWORD*)L"Hello World";

  __asm {
    // Передадим параметр функции
    push p
    // Вызовем функцию
    call pSayHello
  }
}
```

Для компиляции данного исходного кода можно воспользоваться командой

```
cl Some.cpp
```

Как нетрудно увидеть, при компиляции данного приложения не использовались никакие дополнительные библиотеки и сервисы. Тем не менее, в результате его работы на консоль будут выведены следующие строки

```
Hello World
Hello World
```

Что с одной стороны неудивительно, поскольку мы этого, собственно, и добивались. Но если посмотреть повнимательней, то становится заметна одна

странность — вызов происходил в обычном для неуправляемого кода режиме, а управление каким-то образом было передано управляемой функции.

Как это устроено?

Для того чтобы ответить на этот вопрос, необходимо обратиться к устройству динамической библиотеки. Дизассемблируем ее и посмотрим на код экспортируемой функции `SayHello` (листинг 11.23).

Листинг 11.23. Заглушка в неуправляемом коде, используемая для передачи управления среде исполнения

```
public SayHello
SayHello  proc near
  jmp  dword_10008030
SayHello  endp
```

Где `dword_10008030` — некоторый адрес внутри динамической библиотеки. На самом деле, это специальная таблица перехода, которая заполняется во время загрузки библиотеки. Но как же происходит ее заполнение, кто это делает и куда она ведет? Для того чтобы ответить на этот вопрос, взглянем на список DLL, импортируемых нашей библиотекой (прошу простить меня за невольный каламбур). Воспользуемся утилитой `dumpbin`, запустив ее следующей командной строкой

```
dubmpbin /imports Hello.dll
```

В результате утилита выдаст следующий лог:

```
Dump of file Hello.dll


File Type: DLL


  Section contains the following imports:


    mscoree.dll
              10002000 Import Address Table
              100025E0 Import Name Table
                     0 time date stamp
                     0 Index of first forwarder reference


                    59 _CorDllMain
```

```
  Summary

        1000 .data
        1000 .rdata
        1000 .reloc
        1000 .text
```

Как видно, используется лишь одна динамическая библиотека `mscoree.dll`, статически слинкованная с нашей. Оказывается, что библиотека `mscoree.dll` является загрузчиком виртуальной машины .NET.

Теперь осталось только определить, каким образом управление передается этой самой библиотеке. После дизассемблирования выяснилось, что обращение среды исполнения к библиотеке-загрузчику происходит в функции инициализации нашей библиотеки.

```
public start
DllMain  proc near
  jmp  ds:_CorDllMain
DllMain  endp
```

Несмотря на то, что мы явным образом не создавали функцию инициализации, компилятор создал код за нас.

Теперь картина работы нашей программы восстановлена полностью. Во время загрузки нашей динамической библиотеки `Hello.dll`, загрузчик операционной системы определяет, что с ней статически связана библиотека `mscoree.dll`. Он проецирует ее на адресное пространство неуправляемого процесса, использующего нашу "хитрую" библиотеку. После завершения этапа сканирования и проецирования всех необходимых библиотек, загрузчик обращается к функции `DllMain` нашей библиотеки, откуда управление передается загрузчику `mscoree.dll` виртуальной машины .NET.

При вызове функции `DllMain` загрузчик помещает в стек три параметра. Напомним прототип функции `DllMain`.

```
BOOL STDMETHODCALLTYPE DllMain(
   // Описатель загружаемой библиотеки
   HINSTANCE   hInst,
   // Повод вызова для данной функции
   DWORD       dwReason,
   // Зарезервировано для дальнейшего использования
   LPVOID      lpReserved
)
```

Все эти три параметра прямиком попадают в функцию `_CorDllMain`, поскольку оказывается, что функция инициализации нашей библиотеки явля-

ется всего лишь заглушкой, не взаимодействующей со стеком. Далее среда исполнения по описателю нашей библиотеки, переданного в качестве первого параметра `DllMain`, отыскивает в памяти образ файла нашей библиотеки `Hello.dll`. Затем из секции `.rdata` извлекается сборка с управляемым кодом нашего .NET-приложения. Среда исполнения загружает ее в память, а в файле динамической библиотеки устанавливает связи между экспортируемыми заглушками и управляемыми функциями сборки. Общая схема данного механизма представлена на рис. 11.14.

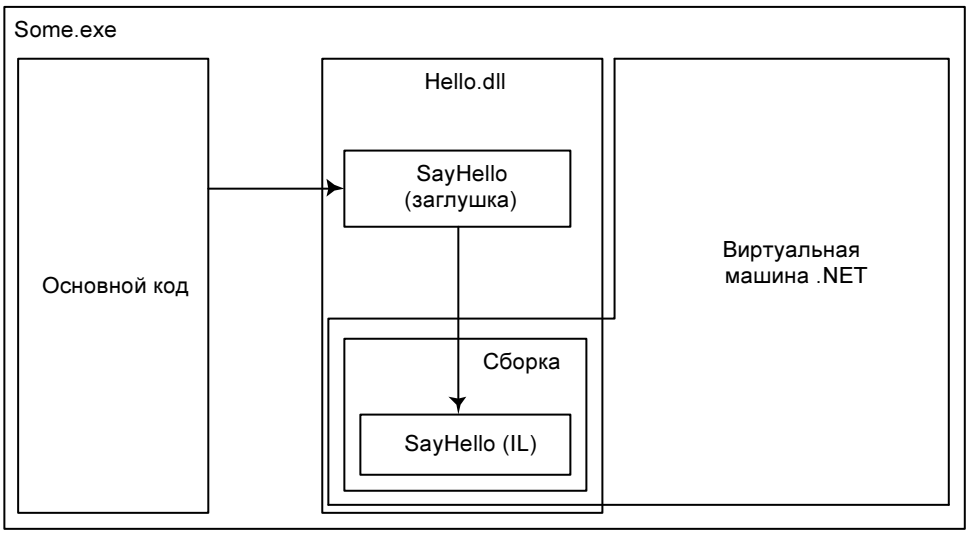


Рис. 11.14. Общая схема механизма взаимодействия с управляемым кодом через классическую динамическую библиотеку

Таким образом, вызов функции осуществляется при непосредственном участии среды исполнения. Следовательно, библиотека не будет работать в системе, без установленной среды исполнения. При попытке запуска приложений, ссылающихся на такую библиотеку, будет выдано предупреждение о невозможности разрешить все связи библиотеки, а в качестве проблемной будет указана `mscoree.dll`.

Глава 12

Использование COMпонентов при помощи .NET

В данной главе будет рассмотрено взаимодействие с операционной системой из управляемого .NET-кода через подсистему COM.

Всего существует два подхода к взаимодействию с COM из .NET. Первый — автоматизированный, при помощи особых сборок, генерируемых на стадии разработки приложения специальными утилитами. Второй подход предполагает взаимодействие в ручном режиме, когда программист должен самостоятельно контролировать процесс взаимодействия с COM, при помощи специализированных сервисов. Для введения в тему, мы рассмотрим первый и наиболее простой вариант взаимодействия, а затем перейдем к более сложным случаям.

12.1. Автоматизированный подход

Для экспериментов будет использоваться COM-объект Microsoft Shell Controls And Automation (Управление и автоматизация оболочки от Microsoft), управляющий пользовательской оболочкой Windows, то есть Проводником.

Примечание

Под Проводником здесь подразумевается не только само приложение, а вся пользовательская оболочка Windows, включающая в себя панель задач, рабочий стол, окна настройки. Сюда также относится пространство имен объектов окружения, включающее панель управления, настройки сети и т. п.

Данный компонент, так же как и его библиотека типов, расположены в файле `Shell32.dll`, который без труда можно обнаружить в системном каталоге Windows. Для того чтобы наладить взаимодействие с данным компонентом из среды .NET, нам необходимо создать специальную сборку-переходник. Сделать это можно при помощи утилиты `tlbimp.exe`, запустив ее следующим образом:

```
tlbimp.exe Shell32.dll /out:Shell32NET.dll
```

В результате получаем сборку Shell32NET.dll, сгенерированную по библиотеке типов компонента, расположенного в ресурсах динамической библиотеки Shell32.dll. Созданная сборка является всего лишь заглушкой, указывающей среде исполнения на необходимость взаимодействия с подсистемой COM, с помощью расположенных в ней типов.

Теперь рассмотрим приложение, которое будет взаимодействовать с COM-объектом, используя нашу сборку. Будем экпериментировать с COM-классом Shell, который отвечает за управление пользовательской оболочкой Windows.

Примечание

Описание объекта Shell вы можете найти в MSDN в разделе User Interface Design and Development\Windows Shell\Shell Reference\Shell Objects for Scripting and Microsoft Visual Basic.

Исходный код приложения, работающего с COM-объектом Shell, представлен далее (листинг 12.1).

Листинг 12.1. Простейший пример взаимодействия с COM из .NET-кода

```
/*
  Листинг 12.1
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключаем основное пространство имен общей библиотеки классов
using System;

// Подключаем пространство имен сборки переходника
using Shell32NET;

// Основной класс нашего приложения
class App
{
  // Точка входа в приложение
  public static void Main()
  {
    // Создадим новый экземпляр COM-объекта
```

```
    // Класс Shell описан в сборке Shell32NET
    Shell sh = new Shell();

    // Вызовем его метод COM-объекта
    sh.FileRun();
  }
}
```

Для компиляции приложения необходимо при помощи ключа `/reference` указать на связь со сборкой `Shell32NET.dll`. Сделать это можно следующим образом:

```
csc /reference:Shell32NET.dll Some.cs
```

В результате мы должны получить исполняемый файл Some.exe, после запуска которого мы увидим на экране диалоговое окно с предложением запустить файл. Это то же самое окно, которое появляется при выборе пункта **Run** (Выполнить) из меню **Start** (Пуск) (рис. 12.1).

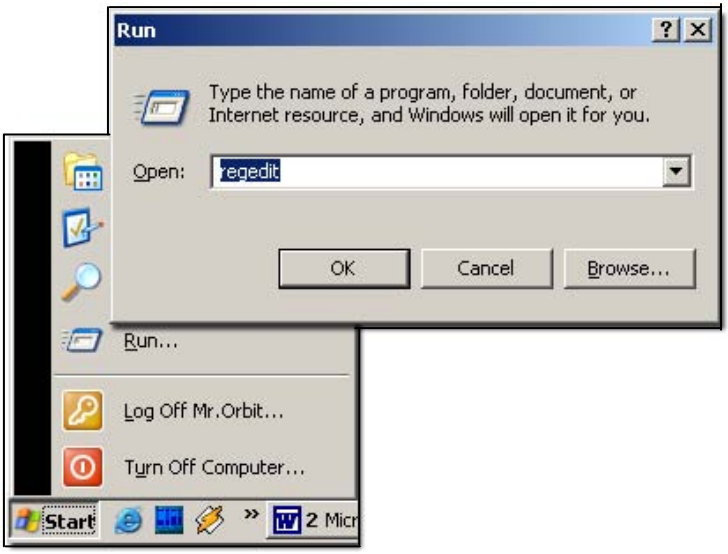


Рис. 12.1. Результат первого взаимодействия с COM-объектом из управляемого кода — диалоговое окно запуска файла

При помощи объекта `Shell` можно вывести на экран большинство диалоговых окон из стандартной пользовательской оболочки Windows.

Особенности подключения COM-объектов из среды Microsoft Visual Studio .NET

Полностью рассматривать процесс разработки приложений, взаимодействующих с COM, в среде Visual Studio .NET мы не будем, поскольку, с точки зрения написания кода, он аналогичен рассмотренному в примере. Нас будут интересовать лишь особенности подключения объектов, с точки зрения самой среды разработки Visual Studio .NET. Любителей интерактивной среды разработки сразу же обрадую — использовать консольные утилиты больше не придется.

Первым делом необходимо установить в проекте связь с COM-компонентом. Для этого в Solution Explorer щелкните правой кнопкой мыши на пункте **Reference**, затем из контекстного меню выберите пункт **Add Reference** (рис. 12.2).

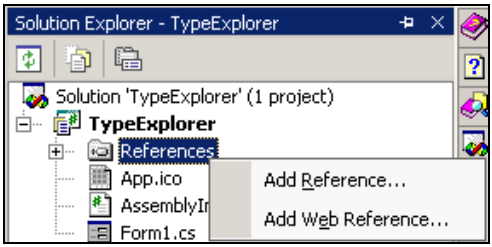


Рис. 12.2. Первый шаг к добавлению необходимой связи с COM-объектом

В результате, на экране должно появиться диалоговое окно **Add Reference** (рис. 12.3), в котором выберем закладку **COM**.

На ней будут перечислены все зарегистрированные в системе COM-объекты. Выберем объект **Microsoft Shell Controls And Automation**, после этого нажмем кнопку **Select**, находящуюся справа, добавив тем самым наш компонент к списку.

Примечание

Если в основном списке компонентов не окажется нужного, то, нажав кнопку **Browse,** можно самостоятельно выбрать файл (возможные расширения включают — dll, tlb, olb, ocx, exe), в котором располагается компонент.

После закрытия данного диалогового окна при помощи кнопки **OK**, среда Visual Studio .NET автоматически создаст сборки-заглушки для каждого компонента. При желании их легко найти в одном из подкаталогов вашего приложения. Visual Studio .NET будет давать имена сборкам-заглушкам по следующему правилу: Interop.ИмяФайлаИзКоторогоБылаПолученаИнформацияОТипах.dll.

Теперь все связи установлены, и вы можете использовать компонент из среды Visual Studio .NET обычным образом — как простой .NET-класс.

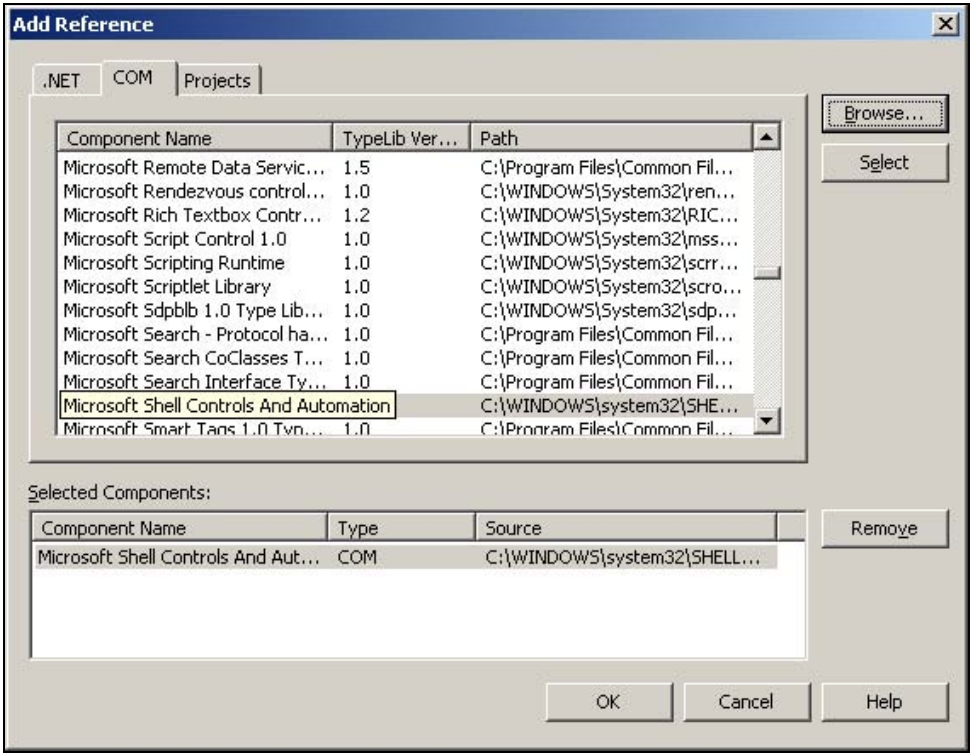


Рис. 12.3. Окно добавления связи с COM-объектом

Как это устроено или что там у него внутри?

Чтобы разобраться в том, как устроено наше приложение, необходимо посмотреть его реальный IL-код. Для этого воспользуемся утилитой `ildasm`, запустив ее следующим образом:

```
ildasm Some.exe /out:Some.il
```

В результате мы получим следующий IL-код (листинг 12.2).

Листинг 12.2. Исходный IL-код сборки-заглушки

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 1.1.4322.573
//  Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )        // .z\V.4..
```

```
  .ver 1:0:3300:0
}

// Ссылка на сборку, созданную нами при помощи утилиты tlbimp
.assembly extern Shell32NET
{
  .ver 1:0:0:0
}

.assembly Some
{
  // --- The following custom attribute is added automatically,↳
 do not uncomment -------
  //  .custom instance void ↳ [mscor-
lib]System.Diagnostics.DebuggableAttribute::.ctor(bool,
 // bool) = ( 01 00 00 01 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}

.module Some.exe
// MVID: {90043DC9-AB42-4946-9F25-82EDFE893302}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001

.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
} // end of class App

.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  // Точка входа в приложение
  .method public hidebysig static void  Main() cil managed
  {
```

```
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  1
    .locals init (class [Shell32NET]Shell32NET.Shell V_0)
    // Shell sh = new Shell();
    // Обращение к классу Shell компилятор перенаправил к ShellClass

    IL_0000:  newobj     instance void ↳
[Shell32NET]Shell32NET.ShellClass::.ctor()
    IL_0005:  stloc.0
    IL_0006:  ldloc.0

    IL_0007:  callvirt   instance void ↳
[Shell32NET]Shell32NET.IShellDispatch::FileRun()
    IL_000c:  ret
  } // end of method App::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method App::.ctor

}
```

Первое, на что следует обратить внимание в данном листинге, — это подмена компилятором класса `Shell` на `ShellClass`. Теперь, когда мы знаем, к какому классу происходит реальное обращение из нашего приложения, обратимся к сборке, созданной нами при помощи утилиты `tlbimp`. Эта сборка позволяла нам с легкостью использовать COM-компонент, без написания какого-либо дополнительного кода. Давайте заглянем внутрь сборки и посмотрим, как устроен класс `ShellClass`, к которому мы обращались. Для этого аналогичным способом воспользуемся утилитой `ildasm`. Приведем листинг частично, поскольку целиком он слишком велик (листинг 12.3).

Листинг 12.3. Класс ShellClass, предоставляющий доступ к интересующему нас объекту

```
// Обратите внимание на спецификатор import, он указывает
// на то, что класс импортирован из среды COM
  .class public auto ansi import ShellClass
         extends [mscorlib]System.Object
         implements Shell32NET.IShellDispatch,
                    Shell32NET.Shell
  {
    .custom instance void↳
[mscorlib]System.Runtime.InteropServices.TypeLibTypeAttribute::↳
.ctor(int16) = ( 01 00 02 00 00 00 )

     // Guid класса
    .custom instance void ↳
[mscorlib]System.Runtime.InteropServices.GuidAttribute::.ctor(string) = ↳
( 01 00 24 31 33 37 30 39 36 32 30 2D 43 32 37 39  // ..$13709620-C279
2D 31 31 43 45 2D 41 34 39 45 2D 34 34 34 35 35   // -11CE-A49E-44455
33 35 34 30 30 30 30 00 00 )                      // 3540000..


    // Определяет способ взаимодействия с COM-объектом
    .custom instance void ↳
[mscorlib]System.Runtime.InteropServices.ClassInterfaceAttribute::↳
.ctor(int16) = ( 01 00 00 00 00 00 )

       ...

    // Метод FileRun к которому мы обращались
    .method public hidebysig newslot virtual
            instance void  FileRun() runtime managed internalcall
    {
       // Атрибут, указывающий COM-идентификатор метода, необходимы для
       // обращения к нему через интерфейсы автоматизации
      .custom instance void ↳
[mscorlib]System.Runtime.InteropServices.DispIdAttribute::↳
.ctor(int32) = ( 01 00 09 00 02 60 00 00 )
```

```
      .override Shell32NET.IShellDispatch::FileRun
    }
...
```

Информации, представленной в данном коде, оказывается вполне достаточно для организации взаимодействия с подсистемой COM. Рассмотрим код подробнее.

Во-первых, при определении данного класса использовался модификатор `import`. При попытке создания такого класса среда исполнения сканирует данный класс на наличие атрибута `GuidAttribute` (рис. 12.4).

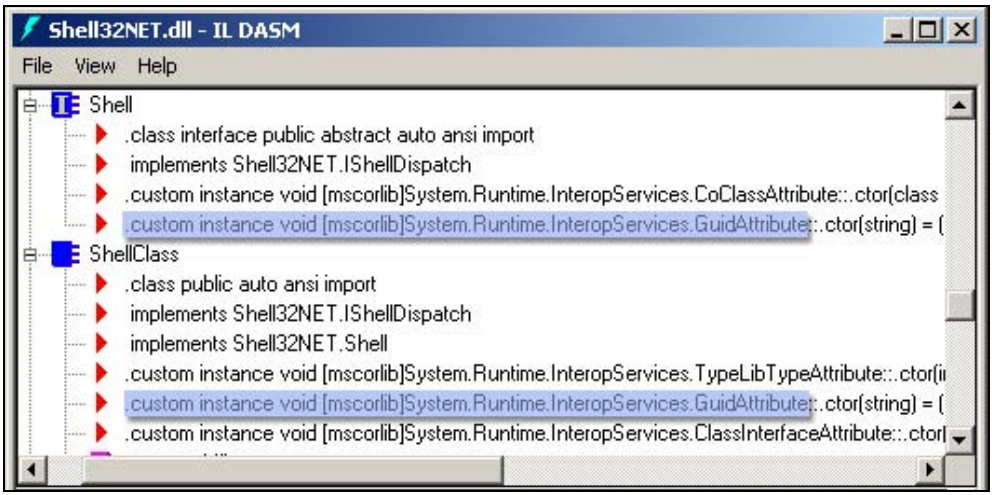


Рис. 12.4. Атрибут `GuidAttribute`, задающий идентификатор COM-объекта

Данный атрибут определяет идентификатор `CLSID` COM-класса, при помощи которого происходит создание экземпляра COM-класса. Далее среда проверяет значение атрибута `ClassInterfaceAttribute`, в нашем случае он будет равен `ClassInterfaceType.AutoDispatch`, что предписывает среде исполнения взаимодействовать с COM-объектом при помощи технологии автоматизации.

Примечание

Пространство имен COM построено на специальных 128-битных `GUID`-идентификаторах. Есть два основных логических вида идентификаторов — `CLSID` для классов и `GUID` для интерфейсов. Физически они не отличаются. `CLSID` служит для идентификации COM-классов, для взаимодействия с которыми используются интерфейсы. Любой интерфейс представляет собой объединение нескольких функций в группу и не более того.

Для создания COM-объекта вам необходимо указать CLSID класса, а также идентификатор интерфейса, через который вы хотите взаимодействовать с объектом. После этого будет сформирована таблица указателей на функции, которая на самом деле и является тем самым интерфейсом. Через данную таблицу вы можете обращаться к функциям компонента. В общем, технология по своей сути очень похожа на динамические библиотеки, за исключением того, что здесь функции передаются не по одиночке, а группами.

При создании скриптовых языков, способных взаимодействовать с COM, перед их разработчиками встала серьезная проблема. Данные языки не поддерживали работу с указателями, а тем более с таблицами указателей (интерфейсами). Специально для таких языков была создана технология автоматизации, которая позволяла обращаться к объектам не через указатели, а при помощи обычных символьных удобочитаемых имен. Суть данной технологии очень проста. Каждый компонент обязан предоставлять заранее известный интерфейс IDispatch, который позволят косвенно обращаться к методам своих интерфейсов при помощи имен. Данный интерфейс прозрачно для программистов используется скриптовыми интерпретируемыми машинами, которые перенаправляют вызовы с верхнего уровня языка прямиком в IDispatch компонента.

В нашем случае IDispatch будет использовать среда исполнения, переадресуя в него управляемые вызовы, опять же абсолютно прозрачно для нас.

Единственный недостаток автоматизации — это скорость работы. Судите сами, при использовании автоматизации для обращения к члену интерфейса требуется, по крайней мере, сравнить строки, представляющие его имя. А при прямом использовании компонента вызов происходит мгновенно: в регистр ECX загружается указатель на объект и вызывается инструкция call с адресом метода.

Теперь, когда у среды исполнения есть вся необходимая информация по объекту, она создает COM-объект. Для этого она обращается к функции CoCreateInstance.

```
STDAPI CoCreateInstance(
  // Идентификатор COM-класса, который мы хотим создать
  REFCLSID rclsid,
  LPUNKNOWN pUnkOuter,
  DWORD dwClsContext,
  // Идентификатор интерфейса, через который мы будем работать с классом
  REFIID riid,
  // Сюда возвращается указатель на интерфейс
  LPVOID * ppv
);
```

В качестве первого параметра будет передан идентификатор класса, указанный в атрибуте GuidAttribute. А четвертый параметр укажет на необходимость предоставления интерфейса автоматизации IDispatch, через который среда исполнения будет работать с объектом. Рассмотрим данный интерфейс (листинг 12.4).

Листинг 12.4. Основной интерфейс поддержки механизма автоматизации — IDispatch

```
MIDL_INTERFACE("00020400-0000-0000-C000-000000000046")
IDispatch : public IUnknown
{
public:
  // Позволяет определить, предоставляет ли объект информацию
  // о типах
  virtual HRESULT STDMETHODCALLTYPE GetTypeInfoCount(
              /* [out] */ UINT *pctinfo) = 0;

  // Позволяет получить информацию о типах объекта при помощи
  // интерфейса ITypeInfo
  virtual HRESULT STDMETHODCALLTYPE GetTypeInfo(
              /* [in] */ UINT iTInfo,
              /* [in] */ LCID lcid,
              /* [out] */ ITypeInfo **ppTInfo) = 0;

  // Позволяет получить идентификатор члена интерфейса (это всего
  // LONG, то есть обычное число), по строке, представляющей его имя.
  virtual HRESULT STDMETHODCALLTYPE GetIDsOfNames(
              /* [in] */ REFIID riid,
              /* [size_is][in] */ LPOLESTR *rgszNames,
              /* [in] */ UINT cNames,
              /* [in] */ LCID lcid,
              /* [size_is][out] */ DISPID *rgDispId) = 0;

  // Позволяет обратиться к члену интерфейса, по его специальному
  // идентификатору (DISPID), который можно получить при помощи
  // функции GetIDsOfNames
  virtual /* [local] */ HRESULT STDMETHODCALLTYPE Invoke(
        /* [in] */ DISPID dispIdMember,
        /* [in] */ REFIID riid,
        /* [in] */ LCID lcid,
        /* [in] */ WORD wFlags,
        /* [out][in] */ DISPPARAMS *pDispParams,
        /* [out] */ VARIANT *pVarResult,
        /* [out] */ EXCEPINFO *pExcepInfo,
        /* [out] */ UINT *puArgErr) = 0;
};
```

Наиболее интересными являются два последних метода данного интерфейса GetIDsOfNames и Invoke. Первый позволяет получить идентификатор члена (метода) интерфейса, который будет необходим для обращения к нему. А вторая функция производит непосредственное обращение к членам интерфейса по их DISPID-идентификаторам.

Примечание

Такая двухуровневая система вызовов введена не случайно. В принципе, конечно, можно было предоставить только один метод Invoke и заставить пользователей передавать ему не идентификатор члена, а его полное строковое имя. Но разработчики модели COM заботились о скорости работы данной технологии. Дополнительный уровень идентификации был введен для того, чтобы было можно заранее откешировать необходимые идентификаторы, а соответственно ускорить обращение к членам интерфейсов. К тому же, в большинстве случаев, идентификаторы членов интерфейсов указываются в библиотеках типов компонентов, что позволяет отказаться от обращения к методу GetIdsOfNames.

В нашем случае среда исполнения будет использовать только фукнцию Invoke, DISPID-идентификатор метода FileRun указан в сборке при помощи атрибута DispIdAttribute.

Разберем более подробно механизм вызова метода FileRun. Для этого нам понадобится определение метода в Shell32NET.dll-библиотеке.

```
// Метод FileRun, к которому мы обращались
.method public hidebysig newslot virtual
  instance void  FileRun() runtime managed internalcall
{
   // Атрибут, указывающий идентификатор метода, для обращения к
   // нему через интерфейсы автоматизации
   .custom instance void [mscor-
lib]System.Runtime.InteropServices.DispIdAttribute::
.ctor(int32) =        ( 01 00 09 00 02 60 00 00 )
   .override Shell32NET.IShellDispatch::FileRun
}
```

В определении данного метода применено два интересных спецификатора: runtime и interlnalcall. Первый — указывает на то, что метод не имеет явного тела, заданного при помощи IL-кода. Данный спецификатор необходим для загрузчика сборок среды исполнения. Второй — определяет, что обращение к методу должно быть перенаправлено к внутренним сервисам среды исполнения. Последние расшифровывают, для чего используется данный метод, и перенаправляют вызов к соответствующему COM-объекту.

Сервисы взаимодействия с COM просканируют метод на наличие атрибута DispIdAttribute. Далее возможны два варианта. Если атрибут DispIdAttribute

будет обнаружен, его значение будет считано и передано методу `Invoke` интерфейса `IDispatch`. В противном случае, то есть если атрибута не окажется, понадобится еще один шаг, для установления значения `DispId`: среда исполнения обратится к методу `GetIDsOfNames` интерфейса `IDispatch`, передав ему символьное имя метода, указанное в метаданных сборки. Все эти действия кешируются средой исполнения, для того чтобы ускорить вызовы COM-методов при последующих обращениях к ним. Для каждого COM-объекта среда исполнения создает персональную заглушку, которая отвечает за общение с ним. Но об этом чуть позднее.

Приведенное описание механизма вызова функции, конечно же, очень сильно упрощено. Была описана только часть механизма взаимодействия, отвечающая за идентификацию методов, которые необходимо вызвать. Кроме этого среда исполнения осуществляет большую работу по маршализации (переносу и конвертированию) данных между средами. Но, в общих чертах, все описанное верно.

Вернемся ненадолго к нашей библиотеке `Shell32NET.dll`, которую мы сгенерировали при помощи утилиты `tlbimp`. Данная утилита, как мы уже успели убедиться, создает сборку, в которой, по сути дела, расположена только информация о типах и ничего более (ни строчки кода). Фактически, утилита `tlbimp` производит преобразование библиотеки типа, расположенной в `Shell32.dll` в сборку. Но как нам просмотреть код библиотеки, а не сборки? Для этого необходимо воспользоваться утилитой OLE/COM Object Viewer (oleview.exe), которая входит в стандартную поставку Visual Studio.NET. Запустите утилиту и выберите пункт **File->View TypeLib**... или щелкните по второй справа иконке с изображением красных треугольников (рис. 12.5).

Рис. 12.5. Открытие библиотеки типов при помощи утилиты OLE/COM Object Viewer

Далее необходимо будет выбрать файл, в котором расположена библиотека типов компонента. Обычно она располагается в файле динамической библиотеки компонента в качестве ресурса. Хотя иногда поставляется и в виде отдельного TLB-файла. Утилита OLE/COM Object Viewer представит содержание динамической библиотеки в виде исходного файла на языке ILD (In-

terface Definition Language, Язык определения интерфейсов), а также построит удобное дерево навигации по коду (рис. 12.6).

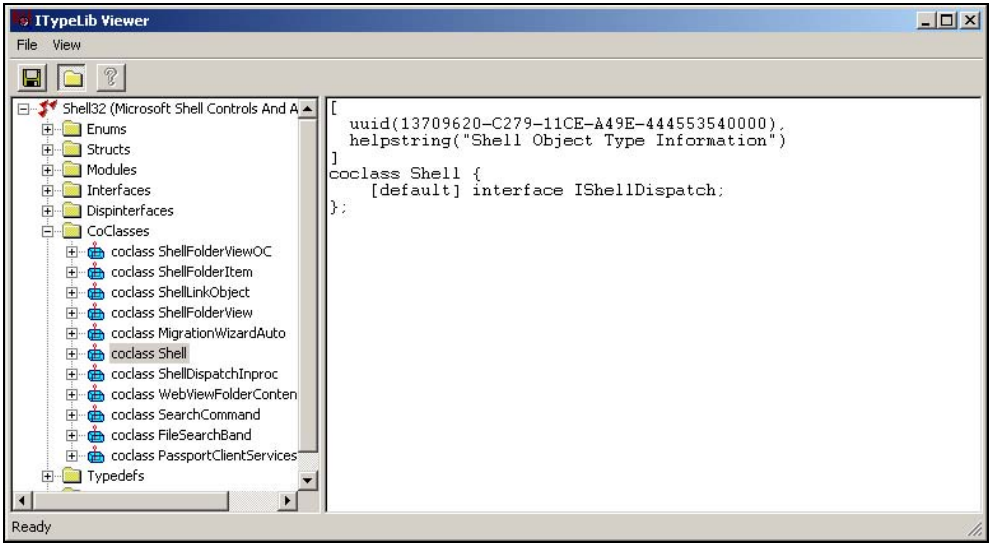


Рис. 12.6. Исходный код библиотеки типов на языке IDL, просматриваемый при помощи утилиты OLE/COM Object Viewer

Исходный код на языке IDL по сути дела представляет собой ассоциативную карту. Которая каждому (практически) элементу кода сопоставляет идентификатор: `GUID`, `CLSID` или `DISPIP`. Помимо этого здесь представлена информация о типах и дополнительных параметрах маршализации.

При работе с COM-объектами эта утилита не раз сослужит вам хорошую службу. Она поможет разобраться с самыми неприятными и коварными проблемами. С таким мощным инструментом от взгляда профессионала не ускользнет ни одна проблема.

12.2. Динамическое взаимодействие в ручном режиме

А теперь рассмотрим взаимодействие с COM-объектами полностью в ручном режиме. Даже если вас полностью устраивает предыдущий метод взаимодействия, то с ручным подходом все-таки стоит ознакомиться. Только в этом случае придет понимание фундаментальных основ работы с подсистемой COM из управляемого кода.

Как уже рассказывалось ранее, создание COM-объектов происходит при помощи функции `CoCreateInstanse`, приведем ее прототип:

```
STDAPI CoCreateInstance(
```

```
  // Идентификатор COM-класса — экземпляр, который требуется создать
  REFCLSID rclsid,
  // Данный параметр
  // необходим для агрегации объектов
  LPUNKNOWN pUnkOuter,

  // Определяет контекст создания объекта.
  // параметр определяет, будет ли создан объект
  // в контексте вашего процесса или как отдельный процесс, а также
  // позволяет задать еще несколько типов настроек, которые мы
  // рассматривать не будем.
  DWORD dwClsContext,

  // А это идентификатор интерфейса, который мы хотим получить
  // от данного объекта.
  REFIID riid,

  // Адрес, в который необходимо положить указатель на интерфейс
  LPVOID * ppv
);
```

В качестве основного здесь выступает первый параметр, через который передается идентификатор объекта, который необходимо создать. Идентификатор представляет собой 128-битное число в формате `GUID`. В нашем случае он будет выглядеть так `{13709620-C279-11CE-A49E-444553540000}`.

Но постойте, ведь данный объект предназначен для использования в скриптовых языках, а они не поддерживают работу с `GUID`. Каким же образом скриптовая машина, не имея под рукой библиотеки типов компонента, узнает необходимый идентификатор `CLSID` объекта? Для того чтобы разрешить данную проблему, был введен дополнительный класс идентификаторов — `ProgId`. Это обычная строка, которой в реестре сопоставлен `CLSID` объекта. К примеру, для нашего объекта `ProgId` равен `Shell.Application.1`. Соответственно в реестре существует ключ с таким именем, в котором указан `CLSID` (рис. 12.7).

А подсистема COM предоставляет специальный сервис, который позволяет получить `CLSID`, указав `ProgID` компонента.

```
HRESULT CLSIDFromProgID(
  // Передаем ProgID
  LPCOLESTR lpszProgID,
  // Получаем CLSID
  LPCLSID pclsid
);
```

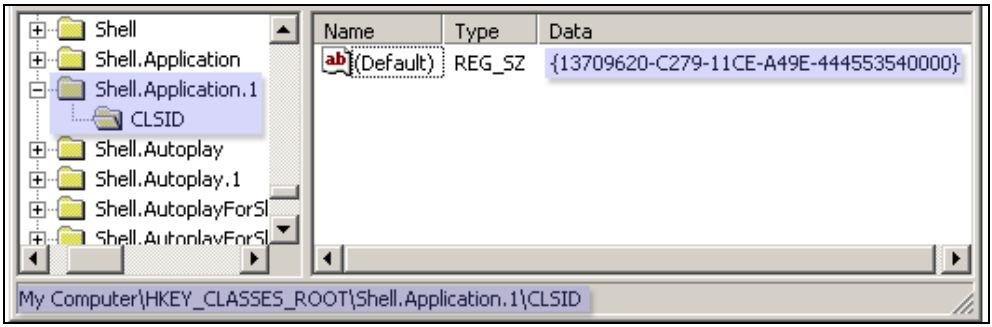


Рис. 12.7. Ключ реестра, содержащий `CLSID`

Таким образом, создание компонента в скриптовых языках происходит в два этапа:

1. При помощи функции `CLSIDFromProgID` выясняется `CLSID` компонента по его `ProgID`.
2. Далее создается сам объект при помощи функции `CoCreateInstance`, которой в качестве первого параметра передается `CLSID`, полученный на первом шаге.

Для наглядности продемонстрируем скрипт, использующий объект `Shell` (листинг 12.5 и листинг 12.6).

Листинг 12.5. Пример на Jscript, использующий COM-объект Shell

```
/*
   Листинг 12.5
   File:   Some.js
   Author: Дубовцев Алексей
*/
ShellObj = new ActiveXObject("Shell.Application.1");
ShellObj.FileRun();
```

Листинг 12.6. Пример на VBScript, использующий COM-объект Shell

```
'
'  Листинг 12.6
'  File:   Some.vbs
'  Author: Дубовцев Алексей
```

```
'
Dim ShellObj
Set ShellObj = CreateObject("Shell.Application.1")
ShellObj.FileRun()
```

Оба эти примера по функциональности полностью идентичны, они выводят все то же диалоговое окно **Run** (рис. 12.8).

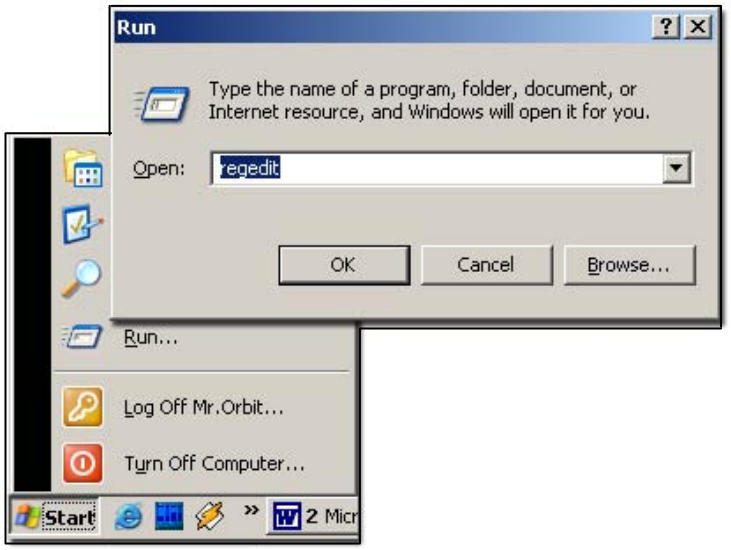


Рис. 12.8. Результат взаимодействия с объектом `Shell` из скриптовых языков

Приведенные скрипты успешно работают, без использования библиотек типов и предварительного обращения к динамической библиотеке `Shell32.dll` типов компонента. Следовательно, то же самое можно сделать при помощи .NET. Попробуем осуществить задуманное.

Для этого нам придется воспользоваться сервисами динамического управления типами (технология отражения), представленными классами пространства имен `System.Reflection`. Мы воспользуемся хитрым трюком — создадим псевдотип, который будет перенаправлять наши запросы к реальному COM-классу. Создать такой тип нам помогут следующие методы, входящие в состав класса `Type` (листинг 12.7).

Листинг 12.7. Функции, отвечающие за предоставление доступа к COMпоненту через его ProgID

```
// Возвращает тип, который будет связан с COM-объектом,
// ProgID, которого мы передадим в данную функцию
public static Type Type.GetTypeFromProgID(
```

```
   // Строка, представляющая ProgID объекта
   string progID
);

// Позволяет получить тип, который будет связан с COM-объектом по его CLSID
public static Type Type.GetTypeFromCLSID(
   Guid clsid
);

// Данные прототипы аналогичны предыдущим, за исключением того, что
// позволяют контролировать выброс исключения TypeLoadException,
// которое произойдет при неудачной попытке обращения к COM-объекту
public static Type Type.GetTypeFromProgID(
  string progID,

  // Если передано true - то в случае возникновения проблем будет
  // выброшено исключение, если false, то исключение выброшено
  // не будет
  bool throwOnError
);


public static Type Type.GetTypeFromCLSID(
  Guid clsid,

  // Если передано true - то в случае возникновения проблем будет
  // выброшено исключение, если false, то исключение выброшено
  // не будет
  bool throwOnError
);
```

Тип, который мы создадим при помощи данных методов, сам по себе не позволит связываться с COM-объектом. Он лишь будет хранить необходимую информацию для взаимодействия с ним. Для того чтобы обратиться к объекту, надо создать экземпляр данного типа. Сделаем это при помощи метода `CreateInstance` класса `Activator`.

```
public static object Activator.CreateInstance(
   Type type
);
```

Действие данного метода аналогично оператору `new`, только здесь тип указывается не явно, а передается в качестве объекта `Type`. Основная проблема заключается в том, что данный метод возвращает объект типа `Object`, который не содержит необходимых нам методов для взаимодействия с COM-объектом. Более того, он вообще не содержит ни одного метода, который позволял бы обратиться к COM-классу. Для того чтобы обратиться к интересующему нас методу, придется использовать сервисы динамического управления типами, позволяющие осуществлять вызовы скрытых в данном случае методов. В качестве такого сервиса будет выступать метод `InvokeMember` класса `Type`.

```
// Позволяет обращаться к членам типов
public object Type.InvokeMember(
  // Имя члена
  string name,
  // Указывает тип взаимодействия с членом (вызов метода, получение/
  // установка свойства и т. д.)
  BindingFlags invokeAttr,
  // Необходим для сложных случаев взаимодействия
  Binder binder,
  // Объект, для которого необходимо вызвать метод
  object target,
  // Аргументы, которые необходимо передать
  object[] args
);
```

Полный код приложения, взаимодействующего с COM-объектом в динамическом режиме, окажется гораздо проще, чем можно было бы ожидать (листинг 12.8).

Листинг 12.8. Управляемое приложение, взаимодействующее с COM-объектом полностью в динамическом режиме

```
/*
  Листинг 12.8
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за динамическое
```

```
// управление типами
using System.Reflection;

// Основной класс приложения
class App
{
  // Точка входа в приложение
  public static void Main()
  {
    // Получим тип, дающий нам связь с COM-объектом
    // Создадим тип, косвенно связанный с необходимым
    // нам COM-объектом. Для указания COM-объекта используем
    // ProgID
    Type ShellType = Type.GetTypeFromProgID("Shell.Application.1");

    // Создадим данный COM-объект, при помощи ранее запрошенного
    // типа.
    // Именно здесь настоящий COM-объект будет создан при помощи
    // функции CoCreateInstance
    // Создадим экземпляр типа, через который будем взаимодействовать
    // с COM-объектом
    Object ShellObj = Activator.CreateInstance(ShellType);


    // Вот оно то, ради чего писался данный пример.
    // Обратимся к методу интересующего нас COM-объекта,
    // в полностью динамическом режиме
    ShellType.InvokeMember(
      // Имя метода
      "FileRun",
      // Флаг связывания для запуска
      BindingFlags.InvokeMethod,
      // А здесь просто ноль :)
      null,
      // Это ранее созданный нами объект
      ShellObj,
      // Данный метод не принимает параметров
      new object[0]);
  }
};
```

В результате запуска данного приложения на экране появится знакомое нам окно с предложением запустить файл.

Суть данного примера в том, что нам удалось заставить управляемый код взаимодействовать с подсистемой COM, полностью в динамическом режиме, без использования библиотек типов и дополнительных утилит.

Скажем несколько слов по поводу объекта, возвращаемого функцией `GetTypeFromProgID`. На самом деле он имеет тип не `Type`, а `__ComObject`. Данный тип является внутренним типом среды исполнения, отвечающим за работу с внешними COM-объектами. Он перекрывает все основные методы объекта `Type` и перенаправляет обращения к ним через стандартный маршалер среды исполнения в интерфейс `IDispatch` соответствующего объекта.

12.3. Сравнение двух подходов, методы оптимизации

Представленный метод динамического взаимодействия с COM-объектами вручную прекрасно работает. Только у него есть один существенный недостаток — неудобство в использовании. Однако существует два подхода, способных облегчить использование методов динамического взаимодействия:

- написать искусственную оболочку, инкапсулирующую вызовы к системе динамического управления типами;
- использовать специальные средства среды исполнения для создания классов оболочек для COM-объектов, поддерживаемых самой средой исполнения.

"Искусственная оболочка"

Итак, идея состоит в том, чтобы для каждого объекта ввести дополнительный класс, который бы скрывал вызовы к объекту и предоставлял удобный для использования интерфейс (листинг 12.9).

Листинг 12.9. Класс-оболочка, облегчающий взаимодействие с COM-объектом

```
/*
  Листинг 12.9
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки классов
```

```
using System;

// Подключим пространство имен, отвечающее за динамическое
// управление типами
using System.Reflection;

// Класс, инкапсулирующий взаимодействие с COM-объектом при помощи
// динамического управления типами
class Shell
{
  // Закрытые члены класса, необходимые для взаимодействия с объектом
  // в рамках динамического управления типами. Пользователь не должен
  // даже подозревать об их существовании
  protected Type ShellType;
  protected Object ShellObj;

  // Конструктор класса. Он проводит подготовительную работу,
  // необходимую перед обращением к объекту.
  public Shell()
  {
    // Получим тип, дающий нам связь с COM-объектом
    // Создадим тип, косвенно связанный с необходимым
    // нам COM-объектом. Для указания COM-объекта используем
    // ProgID
    ShellType = Type.GetTypeFromProgID("Shell.Application.1");

    // Создадим данный COM-объект, при помощи ранее запрошенного
    // типа.
    // Именно здесь настоящий COM-объект будет создан при помощи
    // функции CoCreateInstance
    // Создадим экземпляр типа, через который будем взаимодействовать
    // с COM-объектом
    ShellObj = Activator.CreateInstance(ShellType);
  }

  // Метод, позволяющий обратиться к методу FileRun COM-объекта
  public void FileRun()
  {
    // Вот оно то, ради чего писался данный пример.
```

```
    // Обратимся к методу интересующего нас COM-объекта
    ShellType.InvokeMember(
        // Имя метода
        "FileRun",
        // Флаг связывания для запуска
        BindingFlags.InvokeMethod,
        // А здесь просто ноль :)
        null,
        // Это ранее созданный нами объект
        ShellObj,
        // Данный метод не принимает параметров
        new object[0]);
  }
};

// Основной класс нашего приложения
class App
{
  // Точка входа в приложение
  public static void Main()
  {
    // Создадим COM-объект
    Shell sh = new Shell();
    // Вызовем метод COM-объекта
    sh.FileRun();

  }
};
```

Результат работы данного примера полностью аналогичен предыдущим.

Как видно, использование COM-объекта существенно упростилось. Мы довели простоту использования до уровня самого первого примера, в котором использовалась специальная сборка, сгенерированная из библиотеки типов компонента.

Специальные средства среды исполнения

В отличие от ручного, способ взаимодействия с COM через динамические библиотеки, использущий технологию автоматизации, — слишком медленный. Этот недостаток можно с легкостью обойти, если при создании классов

заглушек использовать встроенные средства среды исполнения для поддержки взаимодействия с COM.

Теперь немного о самом подходе. Суть его очень похожа на работу утилиты tlbimp.exe, только создавать классы мы будем руками, и делать мы будем это при помощи стандартных средств на языке C#. Для этого нам придется использовать три следующих атрибута:

- `ComImportAttribute`
- `GuidAttribute`
- `InterfaceTypeAttribute`

Рассмотрим их по порядку и во всех подробностях.

ComImportAttribute

Атрибут `ComImportAttribute`, по сути дела, является фиктивным атрибутом, не имеющим параметров и не несущим никакой полезной информации, он лишь указывает, на то, что класс, с которым он связан, является заглушкой для связи с COM-объектом. На уровне IL-кода данный атрибут транслируется компилятором в директиву `import`.

При его использовании необходимо соблюдать одно важное правило: классы, к которым применен данный атрибут, ни в коем случае не должны иметь собственных членов, они должны быть полностью пустыми. Данное требование строго обязательно, иначе при попытке использования данного класса произойдет коллизия загрузки типов и будет выброшено исключение `TypeLoadException`. Но как же тогда использовать данный класс, если в нем запрещено использовать собственные члены? Весь фокус состоит в том, что он будет использоваться не для доступа к объекту, а лишь для его создания. Для обращения к объекту мы будем использовать специальный интерфейс, но об этом чуть позднее.

GuidAttribute

Очевидно, что одного атрибута `ComImportAttribute` для создания COM-класса явно недостаточно. Как минимум необходимо задать `CLSID` класса, который мы хотим использовать. Для его определения предназначен атрибут `GuidAttribute`, имеющий один конструктор с единственным позиционным параметром.

```
public GuidAttribute(
   // Строка, задающая GUID
   string guid
);
```

Где `GUID` должен быть передан в виде строки следующего формата: `"XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"`, в которой вместо символа `X`,

естественно, должны стоять шестнадцатеричные цифры. Если формат строки будет нарушен, то компилятор сообщит об этом, выбросив ошибку:

```
error CS0647: Error emitting 'System.Runtime.InteropServices.
GuidAttribute' attribute -- 'Incorrect uuid format.'
```

Теперь у нас есть все средства, необходимые для создания COM-объектов — два атрибута `ComImportAttribute` и `GuidAttribute`. И мы можем определить собственный класс-заглушку, организующий взаимодействие с нужным нам COM-объектом.

```
[
  // Укажем, что класс импортирован из среды COM
  ComImport,
  // Guid класса Shell (Shell.Application.1)
  Guid("13709620-C279-11CE-A49E-444553540000"),
]
class Shell
{
};
```

Создать COM-объект, связанный с данным классом, можно в привычном для языка синтаксисе.

```
Shell h = new Shell();
```

При исполнении данной конструкции средой исполнения будет создан COM-объект, в соответствии с `GUID`, указанным при помощи атрибута `GuidAttribute`.

При этом создание объекта COM полностью базируется на идентификаторе, указанном при помощи атрибута `GuidAttribute`, а само символьное имя класса никакого значения не имеет, оно может быть произвольным. Использование такого класса также будет приводить к взаимодействую с объектом `Shell.Application.1`.

```
[
  // Укажем, что класс импортирован из среды COM
  ComImport,
  // Guid класса Shell (Shell.Application.1)
  Guid("13709620-C279-11CE-A49E-444553540000"),
]
class VuzyBuzy
{
};
```

Но как же раздобыть столь необходимый GUID или точнее CLSID интересующего объекта? Без него ведь никуда. Здесь будут приведены четыре наиболее простых способа.

Во-первых, можно обратиться к документации, прилагающейся к COM-объекту. Хотя в ней порой бывает очень трудно что-либо отыскать, но все же, в большинстве случаев, там содержится вся необходимая информация. К примеру, описание нашего объекта вы можете найти в MSDN (рис. 12.9).

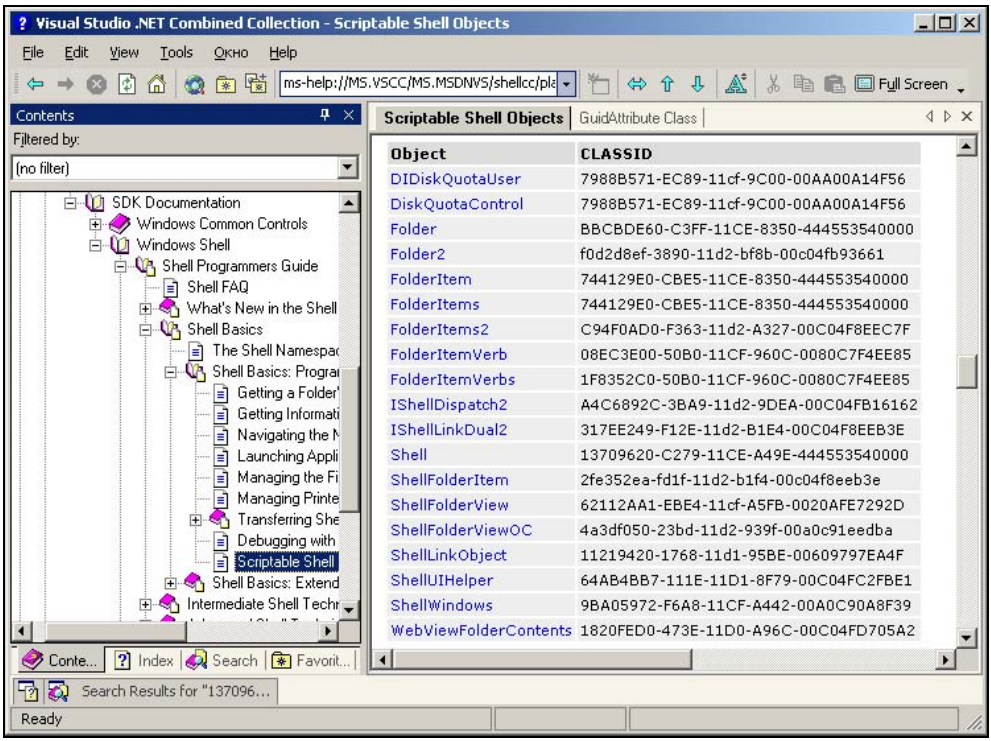


Рис. 12.9. Выяснение CLSID необходимого COM-объекта с помощью документации

Во-вторых, если у вас есть заголовочные файлы, прилагающиеся к компоненту, то найти там необходимую информацию будет еще проще, чем в документации. Для нашего компонента заголовочный файл распространяется в составе Platform SDK, с недавних пор входящего в состав Microsoft Visual C++. Определение необходимого нам идентификатора приведено ниже — оно скопировано прямо из заголовочного файла.

```
EXTERN_C const CLSID CLSID_Shell;


#ifdef __cplusplus
```

```
class DECLSPEC_UUID("13709620-C279-11CE-A49E-444553540000")
Shell;
#endif
```

Необходимо отметить тот факт, что если компонент создавался на языке, отличном от C++, то заголовочных файлов вам обнаружить не удастся.

В-третьих, зачастую вместе с компонентами распространяется исходный код библиотек типов COM, на языке IDL. В нем вы без труда найдете определение необходимого идентификатора.

Для нашего компонента мы сможем найти необходимый файл все в том же Platform SDK — его имя `shldisp.idl`. Приведем часть кода из данного файла, в котором указан необходимый идентификатор.

```
[
   // ВОТ ОН! Именно этот идентификатор нам и нужен!
   uuid(13709620-C279-11CE-A49E-444553540000), // CLSID_Shell
   helpstring("Shell Object Type Information")
]
coclass Shell
{
        [default] interface IShellDispatch;
}
```

В конце концов, даже если к компоненту не прилагается исходного кода библиотеки типов, то, скорее всего, будет прилагаться сама библиотека типов в скомпилированном виде. "Но какой, — скажете вы, — нам от нее прок, если она уже скомпилирована?" Ничего страшного в этом нет, поскольку мы все равно можем просмотреть ее содержимое при помощи утилиты COM/OLE Object Viewer (oleview.exe). Правда, необходимо учитывать тот факт, что зачастую библиотека типов интегрирована в файле динамической библиотеки компонента, в виде ресурса. В нашем случае в файл `Shell32.dll`. Вам необходимо открыть файл, выбрав пункт меню **File -> View TypeLib**. В результате откроется еще одно окно, в котором будет представлен исходный код библиотеки на языке IDL, в дополнение к этому утилита построит дерево объектов, облегчающее навигацию (рис. 12.10).

Единственный вопрос, с которым могут возникнуть трудности, — как узнать, в котором файле находится компонент? Во-первых, можете посмотреть в документации, чаще всего там это указано. А во-вторых, можно просто догадаться. Данный процесс трудно описать, но чаще всего опытные программисты решают данную задачу, пользуясь лишь интуицией.

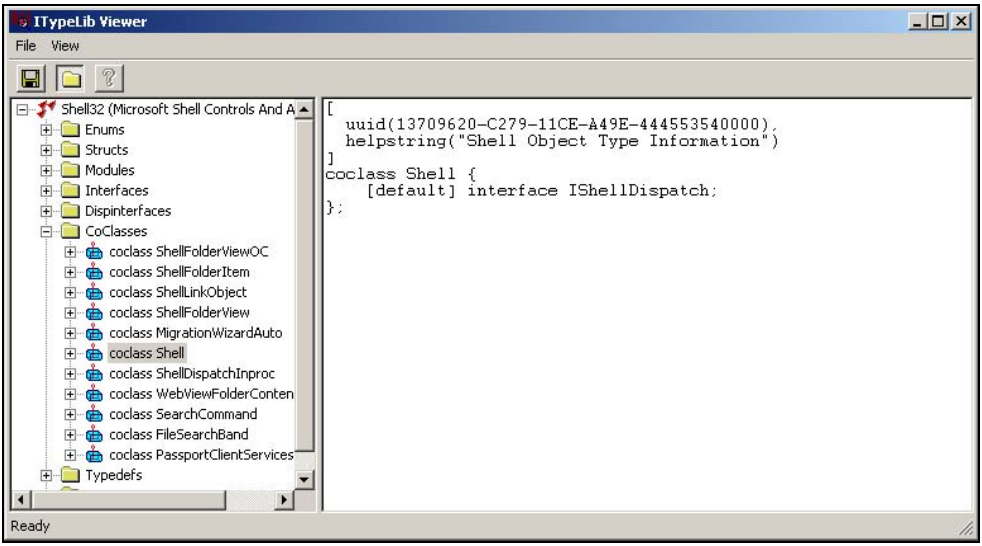


Рис. 12.10. ITypeLib Viewer

InterfaceTypeAttribute

Как мы уже выяснили, классы предназначены лишь для создания COM-объектов, а вся работу по обеспечению взаимодействия с подсистемой COM полностью ложится на интерфейсы. Здесь имеются в виду управляемые, а не COM-интерфейсы.

При описании интерфейсов взаимодействия нам обязательно придется использовать атрибут `InterfaceTypeAttribute`, хотя, как позднее мы узнаем, он не будет ключевым при взаимодействии с объектом.

Конструктор данного атрибута

```
public InterfaceTypeAttribute(
   ComInterfaceType interfaceType
);
```

принимает в качестве своего параметра один из членов перечисления `ComInterfaceType`. Члены данного перечисления определяют, какой тип взаимодействия с объектом будет в итоге выбран средой исполнения. Они подробно описаны далее (табл. 12.1).

Таблица 12.1. *Описание флагов System.Runtime.InteropServices.ComInterfaceType, определяющее способ взаимодействия с COM-объектом*

Член перечисления (Флаг)	Описание
`InterfaceIsDual`	Интерфейс будет рассматриваться как двойной, предоставляющий не только методы `IDispatch`, но также и свои настоящие, через таблицу виртуальных функций

Таблица 12.1 *(окончание)*

Член перечисления (Флаг)	Описание
`InterfaceIsIDispatch`	Интерфейс предоставляет только методы `IDispatch`, соответственно доступ к объекту будет производиться через сервисы автоматизации
`InterfaceIsIUnknown`	Интерфейс помимо обязательного `IUnknown` реализует только собственные методы. Доступ к ним возможен только через таблицу виртуальных функций

Для взаимодействия с нашим объектом выберем второй член `InterfaceIsIDispatch`, поскольку объект предназначен для взаимодействия через сервисы автоматизации. В итоге, определение управляемого интерфейса, предназначенного для взаимодействия с объектом, будет выглядеть следующим образом:

```
[
  // Наш интерфейс импортирован из COM-объекта
  ComImport,
  // Зададим COM-идентификатор
  Guid("D8F015C0-C278-11CE-A49E-444553540000"),
  // Определим тип нашего интерфейса как IDispatch
  InterfaceType(ComInterfaceType.InterfaceIsIDispatch),
]
interface IShell
{
  // Метод вызывает на экран диалоговое окно с предложением
  // запустить файл
  void FileRun();
  // Данный метод позволяет показать определенный апплет
  // панели управления
  void ControlPanelItem([MarshalAs(UnmanagedType.BStr)] String item);
};
```

Именно через данный интерфейс мы будем взаимодействовать с объектом. К данному моменту мы умеем создавать COM-объект и в принципе знаем, как с ним взаимодействовать. Для программистов, знакомых с COM, такой способ взаимодействия окажется привычным, они обычно сначала создают объекты, а потом запрашивают необходимый интерфейс. В COM это делалось при помощи метода `QueryInerface` интерфейса `IUnknown`, который были обязаны поддерживать все без исключения объекты.

```
HRESULT QueryInterface(
```

```
  // Идентификатор интерфейса, грубо говоря - его имя
  REFIID iid,
  // А сюда нам отдадут интерфейс
  void ** ppvObject
);
```

Но как нам осуществить нечто подобное в управляемом коде? Если вы задумали искать метод `QueryInterface` в одном из классов общей библиотеки, то поспешу вас огорчить — его там нет. Все гораздо проще, чем можно было бы предположить: запрос интерфейса производится при помощи преобразования типа. Вам всего лишь нужно преобразовать объект класса `Shell` в интерфейс `IShell`, среда исполнения автоматически перехватит данную операцию и произведет запрос необходимого интерфейса. Выглядит это так:

```
// Создаем COM-объект
Shell ShellObj = new Shell();
// Запрашиваем необходимый нам интерфейс
IShell iShell = (IShell)ShellObj;
```

Если запрос интерфейса от объекта (вызов метода `QueryInterace`) окончится неудачей, то среда исполнения выбросит исключение `InvalidCastException`, со следующим пояснением:

```
Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
   at App.Main()
```

В моем представлении, подобное поведение среды исполнения является крайне некорректным, поскольку не дает информации для понимания сути возникшей проблемы.

А теперь соберем все это воедино и создадим работающий пример (листинг 12.10).

Листинг 12.10. Демонстрация способа взаимодействия с COM-объектом через встроенные декларативные сервисы среды исполнения

```
/*
  Листинг 12.10
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за взаимодействие
```

```
// с операционной системой
using System.Runtime.InteropServices;

// Первым делом опишем интерфейс, который будем использовать
// для взаимодействия с COM-объектом
[
  // Укажем, что интерфейс является заглушкой для связи с COM
  ComImport,
  // Укажем идентификатор интересующего нас интерфейса
  // Данный атрибут наиболее важный из приведенных выше
  Guid("D8F015C0-C278-11CE-A49E-444553540000"),
  // Укажем способ взаимодействия с интерфейсом
  InterfaceType(ComInterfaceType.InterfaceIsIDispatch),
]
interface IShell
{
  // Метод вызывает на экран диалоговое окно с предложением
  // запустить файл
  void FileRun();

  // Данный метод позволяет показать определенный апплет
  // панели управления
  void ControlPanelItem([MarshalAs(UnmanagedType.BStr)] String item);
};

// А теперь опишем класс, при помощи которого мы будем создавать
// наш COM-объект
[
  // Укажем на то, что класс используется для взаимодействия с COM
  ComImport,
  // Укажем идентификатор COM-класса, который соответствует данному
  Guid("13709620-C279-11CE-A49E-444553540000")
]
// Имя класса в данном случае не имеет никакого значения
class Shell
{
};
```

```
// Основной класс нашего приложения
class App
{
  // Точка входа в приложение
  public static void Main()
  {
    // Создадим COM-объект
    Shell ShellObj = new Shell();
    // Запросим у объекта необходимый нам для взаимодействия интерфейс
    IShell iShell = (IShell)ShellObj;

    // Это как и прежде работает, просто закомментировано
    //iShell.FileRun();

    // Вызовем на экран апплет панели управления, отвечающий за настройку
    // параметров дисплея и рабочего стола
    iShell.ControlPanelItem("desk.cpl");
  }
}
```

Скомпилировать данный пример можно при помощи следующей команды:

```
csc Some.csc
```

Для разнообразия был использован метод, показывающий апплет панели управления, отвечающий за настройки дисплея, а также настройку параметров рабочего стола (рис. 12.11).

Наиболее интересным в данном примере является конструкция запроса интерфейса:

```
IShell iShell = (IShell)ShellObj;
```

Людям, хорошо знакомым с ООП (Объектно ориентированное программирование), она наверняка покажется ошибкой. В рамках классического ООП нельзя приводить класс к интерфейсу, если они не имеют родственных связей. Но скоро мы шагнем еще дальше, нарушив все фундаментальные правила ООП, — мы создадим экземпляр интерфейса. Каково звучит: "создать экземпляр абстрактной структуры данных"! А выглядеть это будет так:

```
IShell iShell = new IShell();
```

А идея оказывается проста — среда исполнения позволяет связать с интерфейсом класс, при помощи которого будет создаваться COM-объект, предоставляющий данный интерфейс. Делается это при помощи атрибута `CoClassAttribute`. В качестве единственного параметра конструктор данно-

го атрибута принимает тип класса, который будет использоваться при создании данного интерфейса.

```
public CoClassAttribute(
   Type coClass
);
```

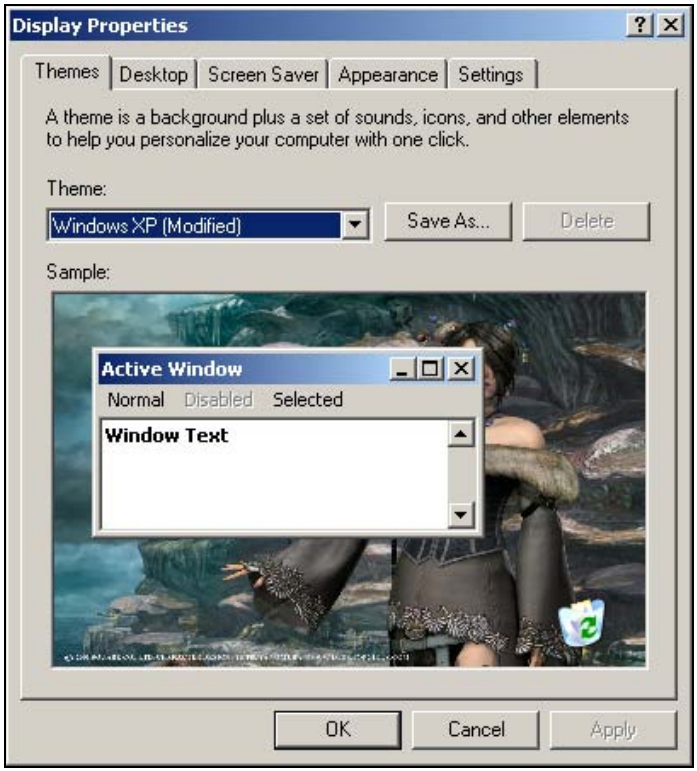


Рис. 12.11. Апплет, отвечающий за настройку параметров рабочего стола, вызванный программно при помощи предыдущего примера

В нашем случае, данный атрибут необходимо будет применить следующим образом:

```
[
  // Наш интерфейс импортирован из COM-объекта
  ComImport,
  // Зададим COM-идентификатор
  Guid("D8F015C0-C278-11CE-A49E-444553540000"),
  // Определим тип нашего интерфейса как IDispatch
  InterfaceType(ComInterfaceType.InterfaceIsIDispatch),
```

```
  // Укажем, что для создания данного интерфейса должен использоваться
  // класс Shell
  CoClass(typeof(Shell))
]
interface IShell
{
  // Метод вызывает на экран диалоговое окно с предложением
  // запустить файл
  void FileRun();
  // Данный метод позволяет показать определенный апплет
  // панели управления
  void ControlPanelItem([MarshalAs(UnmanagedType.BStr)] String item);
};
```

По сути дела, при создании подобного интерфейса будет происходить следующее неявное преобразование:

```
IShell iShell = new IShell();
```

будет преобразовано в

```
IShell iShell = (IShell)(new Shell());
```

Таким образом, мы объединяем в один этап все шаги по созданию объекта и запросу интерфейса. В результате всех этих преобразований, наш листинг изменит свой вид к следующему (листинг 12.11).

Листинг 12.11. Взаимодействие с COM-объектом с использованием декларативных сервисов

```
/*
  Листинг 12.11
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за взаимодействие
// с операционной системой
using System.Runtime.InteropServices;

// Первым делом опишем интерфейс
```

```
[
  // Укажем, что интерфейс является заглушкой для связи с COM
  ComImport,

  // Укажем идентификатор интересующего нас интерфейса
  // Данный атрибут наиболее важный из приведенных выше
  Guid("D8F015C0-C278-11CE-A49E-444553540000"),
  // Укажем способ взаимодействия с интерфейсом
  InterfaceType(ComInterfaceType.InterfaceIsIDispatch),
  // Укажем, что для создания данного интерфейса должен использоваться
  // класс Shell
  CoClass(typeof(Shell)),
]
interface IShell
{
  // Метод вызывает на экран диалоговое окно с предложением
  // запустить файл
  void FileRun();
  // Данный метод позволяет показать определенный апплет
  // панели управления
  void ControlPanelItem([MarshalAs(UnmanagedType.BStr)] String item);
};

// А теперь опишем класс, при помощи которого мы будем создавать
// наш COM-объект
[
  // Укажем на то, что класс используется для взаимодействия с COM
  ComImport,
  // Укажем идентификатор COM-класса, который соответствует данному
  Guid("13709620-C279-11CE-A49E-444553540000")
]
// Имя класса в данном случае не имеет никакого значения
class Shell
{
};

// Основной класс нашего приложения
class App
{
```

```
  // Точка входа в приложение
  public static void Main()
  {
    // Одновременно (для данного уровня) создадим объект и
    // Запросим у него необходимый нам для взаимодействия интерфейс
    IShell iShell = new IShell();

    // Это как и прежде работает, просто закомментировано
    //iShell.FileRun();

    // Вызовем на экран апплет панели управления, отвечающий за настройку
    // параметров дисплея и рабочего стола
    iShell.ControlPanelItem("desk.cpl");
  }
}
```

В результате работы данного приложения на экране появится все тот же апплет панели управления, отвечающий за настройку параметров рабочего стола.

Внесенные изменения не столь кардинальны, как может показаться, ведь мы все-таки не смогли отказаться от класса `Shell`. Мы лишь упростили использование COM-объекта для верхнего уровня программного кода.

Мы довели классы до такого состояния, что использование нашего компонента стало абсолютно тривиальным. Обращение к компоненту вместе с его созданием занимает всего лишь две строки. Все вроде бы хорошо, если не учитывать одного момента — вызовы нашего компонента идут через интерфейсы автоматизации. При этом во время первого обращения к методам COM-интерфейсов, среда исполнения запрашивает у функции `IDispatch->GetIdsOfNames` их COM-идентификаторы, указанные в метаданных по именам членов. Так что при первом обращении будет происходить задержка. При повторном вызове они, конечно же, будут браться из кэша.

Для решения проблемы с первичной задержкой среда исполнения предоставляет способ оптимизации процесса запроса идентификаторов методов интерфейсов. Можно явно указать идентификатор члена, использовав атрибут `DispIdAttribute`.

```
public DispIdAttribute(
   // Числовой идентификатор члена
   int dispId
);
```

В качестве единственного параметра необходимо указать числовой идентификатор члена, с которым связан атрибут. При этом мы получаем дополнительный бонус. При наличии данного атрибута среда исполнения не будет обращать внимание на реальное имя управляемого метода, к которому он применен. Следовательно, его имя можно изменить по своему желанию. Все это продемонстрировано в примере, исходный код которого приведен далее (листинг 12.12).

Листинг 12.12. Пример использования атрибута DispIdAttribute

```
/*
  Листинг 12.12
  File:   Some.cs
  Author: Дубовцев Алексей
*/
// Подключим основное пространство общей библиотеки классов
using System;
// Подключим пространство имен, отвечающее за взаимодействие
// с операционной системой
using System.Runtime.InteropServices;

// Первым делом опишем интерфейс
[
  // Укажем, что интерфейс является заглушкой для связи с COM
  ComImport,

  // Укажем идентификатор интересующего нас интерфейса
  // Данный атрибут наиболее важный из приведенных выше
  Guid("D8F015C0-C278-11CE-A49E-444553540000"),

  // Укажем способ взаимодействия с интерфейсом
  InterfaceType(ComInterfaceType.InterfaceIsIDispatch),

  // Укажем, что для создания данного интерфейса должен использоваться
  // класс Shell
  CoClass(typeof(Shell))
]
interface IShell
{
```

```
  // Метод вызывает на экран диалоговое окно с предложением
  // запустить файл
  void FileRun();

  // Явным образом зададим идентификатор метода
  [DispId(0x60020016)]
  // Теперь, как видите, я могу задавать абсолютно любое имя для
  // моего метода
  // Данный метод позволяет показать определенный апплет
  // панели управления, его реальное имя ControlPanelItem
  void Moe([MarshalAs(UnmanagedType.BStr)] String item);
};


// А теперь опишем класс, при помощи которого мы будем создавать
// наш COM-объект
[
  // Укажем на то, что класс используется для взаимодействия с COM
  ComImport,

  // Укажем идентификатор COM-класса, который соответствует данному
  Guid("13709620-C279-11CE-A49E-444553540000")
]
// Имя класса в данном случае не имеет никакого значения
class Shell
{
};

// Основной класс приложения
class App
{
  // Точка входа в приложение
  public static void Main()
  {
    // Одновременно (для данного уровня) создадим объект и
    // Запросим у него необходимый нам для взаимодействия интерфейс
    IShell iShell = new IShell();
```

```
    // А теперь вызовем непонятный метод, который на самом деле
    // является ControlPanelItem
    iShell.Moe("desk.cpl");
  }
}
```

Приложение все также будет прекрасно работать, и в результате его запуска мы увидим апплет панели управления, позволяющий настроить параметры дисплея.

Динамическое создание COM-объектов

Все предыдущие примеры обладают одним недостатком. Параметры `CLSID` объекта, которые они использовали, были известны заранее и строго прописаны в исходном коде. Данный подход противоречит идеологии и программной модели COM. Создатели COM заложили в нее большой потенциал, о котором, к сожалению, знают немногие даже опытные программисты. Основная идея COM заключается в отделении интерфейса от реализации. Где интерфейс является набором правил для взаимодействия с объектом и только он должен быть неизменен. А объекты COM представляют всего лишь реализацию данных интерфейсов, причем любой интерфейс может быть реализован любым объектом. Таким образом, используя одни и те же интерфейсы, мы можем взаимодействовать с различными, заранее неизвестными COM-объектами. Классическим примером является интерфейс `IUnknown`, реализацию которого имеют все без исключения COM-компоненты. Любое приложение может запросить у любого объекта данный интерфейс, и оно всегда будет знать, как им пользоваться, независимо от какого объекта он был запрошен.

Для интерфейса `IShell` дело обстоит точно таким же образом: он может быть реализован любым другим компонентом. И для того чтобы не остаться на обочине дороги, ведущей к успеху, мы должны уметь динамически создавать любые объекты по своему желанию и запрашивать у них необходимые нам интерфейсы.

Для этого воспользуемся гибридным подходом, в котором будут сочетаться технологии динамического управления кодом, а также статического определения интерфейсов. Код приложения, демонстрирующего данный подход, приведен далее (листинг 12.13).

Листинг 12.13. Демонстрация динамического взаимодействия с COM-объектом

```
/*
  Листинг 12.13
  File:   Some.cs
```

```
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за взаимодействие с
// операционной системой
using System.Runtime.InteropServices;

// Первым делом, опять же, определим интерфейс
[
  // Укажем среде исполнения на то, что данный интерфейс
  // используется для взаимодействия с COM
  ComImport,
  // Укажем идентификатор интерфейса
  Guid("D8F015C0-C278-11CE-A49E-444553540000"),
  // Укажем способ взаимодействия с интерфейсом
  InterfaceType(ComInterfaceType.InterfaceIsIDispatch),
  // Данная возможность нам больше не понадобится, поскольку
  // объект мы будем создавать полностью в динамическом режиме
  //  CoClass(typeof(Shell))
]
interface IShell
{
  // Данный метод показывает на экране диалоговое окно с предложением
  // запустить файл
  void FileRun();
  // Укажем идентификатор метода
  [DispId(0x60020016)]
  // Данный метод показывает на консоли заданный апплет панели
  // управления, реальное имя метода - ControlPanelItem
  void Moe([MarshalAs(UnmanagedType.BStr)] String item);
  //ControlPanelItem
};

// Основной класс приложения
class App
{
```

```
  // Точка входа в приложение
  public static void Main()
  {
    // Запросим тип, который позволит создать объект
    Type ShellType = Type.GetTypeFromProgID("Shell.Application.1");
    // Создадим объект на основе полученного только что типа
    Object shellObj = Activator.CreateInstance(ShellType);

    // Запросим необходимый для взаимодействия с COM-объектом интерфейс
    IShell iShell = (IShell)shellObj;
    // Вызовем метод ControlPanelItem
    iShell.Moe("desk.cpl");
  }
}
```

Как нетрудно догадаться, результат работы данного приложения не будет отличаться от предыдущих.

На этом мы закончим главу, посвященную взаимодействию с подсистемой COM из среды .NET. В ней были изложены лишь общие принципы взаимодействия. Мы не затрагивали самой сложной и важной темы — маршалингу данных. Она будет рассмотрена далее. Необходимо сказать, что если вы хотите писать управляемые приложения, взаимодействующие с COM, то информации, изложенной в данной главе, вам не хватит — придется читать книгу до конца.

Глава 13

Написание COMпонентов при помощи .NET

Нынче модно хвалить или, что чаще встречается, ругать технологии, созданные Microsoft. Я же пытаюсь избегать и того и другого, стараясь оценивать их объективно. Но на сей раз хочу отступить от правил и сказать, что поражен не столько реализацией, сколько существованием технологии, которая будет обсуждаться в данной главе. Если возможность взаимодействия с COM из управляемой среды воспринимается как должное, то об обратной возможности не приходилось и мечтать, просто не приходило в голову, что такое возможно.

Итак, в данной главе мы будем рассматривать технологию создания COM-компонентов при помощи сервисов среды .NET. Необходимо отметить, что эта технология серьезно расширяет сферу применения .NET. Так среда .NET может использоваться для создания частей обычных неуправляемых приложений. При этом взаимодействие с новыми модулями будет производиться при помощи привычных COM-сервисов.

13.1. Различие программных моделей COM и .NET

Прежде чем перейти к детальному описанию технологии, расскажем, в общих чертах, о программных пространствах .NET и COM, а также об их сходствах и различиях.

Обе модели построены на классах. На первый взгляд — очевидное сходство, однако — не все так просто. В среде .NET, так же как и в хорошо знакомом C++, классы служат для создания объектов и для их использования (рис. 13.1). Дополнительные программные элементы не требуются.

Рассмотрим абстрактный пример:

```
// Создаем экземпляр класса
Object obj = new Object();
```

```
// Используем его
obj.GetHashCode();
```

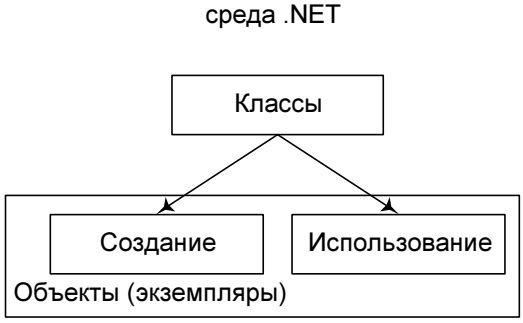


Рис. 13.1. В .NET классы служат как для создания объектов, так и для их использования

Как нетрудно заметить, в данном коде нет никаких иных программных элементов, кроме классов. Возможно, знающий читатель упрекнет меня в том, что я намеренно опускаю описание возможности работы .NET с интерфейсами. Но напомним, что интерфейсы являются дополнительной возможностью и используются по желанию программиста. В общем случае можно обойтись и без них.

В COM же дела обстоят несколько иначе. Для использования объектов в данной среде необходимо применение двух программных сущностей, а именно, классов и интерфейсов. Классы служат для создания объектов, а интерфейсы — для взаимодействия с ними (рис. 13.2).

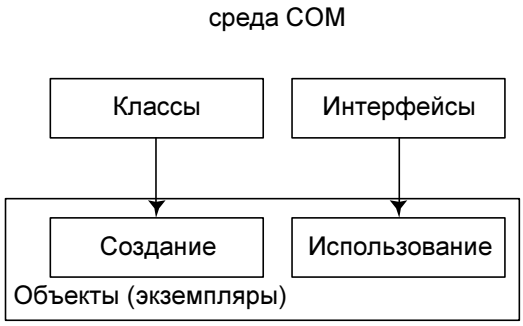


Рис. 13.2. Для использования классов в COM необходимы две программные сущности — классы и интерфейсы

В среде COM классы упрощены до простых идентификаторов, позволяющих запросить создание нужного объекта и не более того. Сам же объект скрыт

от пользователя в недрах компонента и никакого представления о его внутреннем устройстве получить нельзя. Для наглядности приведем пример использования некоторого гипотетического COM-объекта. Обратите внимание на обращение к функции `CoCreateInstance`, именно она производит создание COM-объекта (листинг 13.1).

Листинг 13.1. Идеализированный пример использования COM-объекта

```
// Заголовок намерено пропущен, чтобы не отвлекать вас
// от основного кода
...
void main()
{
  // Инициализируем подсистему COM
  CoInitialize(NULL);

  // Это интерфейс, который используется для доступа к объекту
  // Доступ к объекту может производиться только с его помощью
  // COM-класс в общении с объектом не участвует
  HelloWnd::ICrazyWorld* pICrazyWorld;

  // Вызов данной функции представляет собой создание COM-объекта
  HRESULT hr = CoCreateInstance(
                    // Обратите внимание на данный параметр,
                    // это и есть COM-класс, он передается
                    // для того, чтобы указать, какой именно класс
                    // требуется создать.
                    // Здесь класс — всего лишь имя-идентификатор
            __uuidof(HelloWnd::CrazyWorld),
                 NULL,
                    // Параметры создания компонента
                 CLSCTX_INPROC_SERVER,
                    // А это мы запрашиваем интерфейс, при помощи
                    // которого мы будет взаимодействовать с объектом
                    // мы передаем его идентификатор
                 __uuidof(HelloWnd::ICrazyWorld),
                    // А здесь нам вернут интерфейс
                   reinterpret_cast<void**>(&pICrazyWorld));

  // Проверяем, удалось ли нам создание класса
```

```
  // Об этом мы узнаем от сервиса создания классов CoCreateInstance.
  // Обратите внимание, в данной операции сам класс (его идентификатор)
  // участия не принимает
  if ( !SUCCEEDED( hr) )
  {
    printf("Hm \n");
        // По-видимому, не удалось создать класс
    return;
  }

  // А теперь при помощи интерфейса обращаемся к созданному при помощи
  // класса объекту
  pICrazyWorld->DoIt();

  // Уничтожаем COM-объект, это опять же делается через
  // интерфейс без участия класса (его идентификатора)
  pICrazyWorld->Release();

  // Завершаем работы подсистемы COM
  CoUninitialize();
}
```

Для непосвященного подход, реализующий взаимодействие с COM, может показаться странным. Почему нельзя было сделать все "как у людей" — использовать классы и для создания, и для использования объектов? Ответ прост: данная модель работает только при наличии полноценной информации о типах, связанных с классом. В COM же информация о типах не представлена в необходимом объеме.

В .NET информация о типах представлена так подробно, как никогда ранее. Она базируется на технологии метаданных и подкрепляется технологий отражения. Последняя позволяет производить с типами практически любые операции, начиная от обычного запроса информации и заканчивая динамическим созданием новых типов прямо по ходу исполнения программы.

В COM технология управления типами развита куда меньше. Это можно объяснить тем, что COM разрабатывалась как компонентная надстройка для существовавших в то время языков и компиляторов. Тогда у разработчиков не было возможности создать встроенные средства поддержки информации о типах. Конечно, можно сказать, что уже существовала технология RTTI (Run-Time Type Information, информация о типах времени исполнения), которую можно было развить и включить в COM.

Примечание

RTTI являлась технологией, которая давала возможность получать информацию о типах во время исполнения программы. При классическом подходе к компиляции данная информация полностью терялась в конечной программе. Компиляторы, поддерживающие RTTI, встраивают в приложение дополнительный код, позволяющий получать информацию о типах. Но данной технологии очень далеко до .NET.

Но при разработке технологии COM ставилось жесткое требование полной независимости от языка и платформы. Поэтому использовать встроенные средства какого-либо компилятора разработчики просто не имели права.

Именно поэтому в COM для поддержки информации о типах используется внешний довесок, в виде библиотек типов. Это утверждение может смутить не только начинающего, но даже опытного программиста. Как бы странно это не звучало, но это действительно так. Информация о типах нужна, только для поддержки технологии автоматизации и маршалинга, сама же COM, в пределах одного процесса, может прекрасно работать и без нее. Более того, если поддержку автоматизации и маршалинга писать вручную, то можно вообще отказаться от использования информации о типах. Правда, такой подход, в подавляющем большинстве случаев, не будет оправданным.

Итак, мы разобрались с тем, как устроены объектные среды .NET и COM. Теперь настало время наладить взаимодействие между ними. Для этого мы создадим объект COM при помощи технологии .NET. На уровне .NET будут располагаться классы, а для взаимодействия с ними — использоваться идентификаторы классов и интерфейсы среды COM (рис. 13.3).

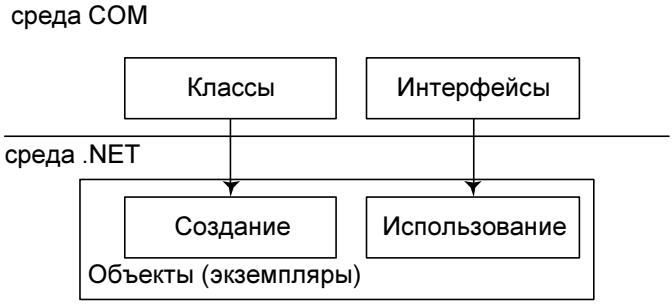


Рис. 13.3. Механизм взаимодействия с .NET-объектами при помощи сервисов среды COM

Для того чтобы организовать подобную модель, каждому .NET-классу необходимо сопоставить COM-класс и интерфейс. О чем, кстати говоря, нам не придется беспокоиться — среда исполнения это сделает за нас автоматически.

13.2. Создаем первый COMпонент при помощи .NET

Создадим простейший компонент, который будет выводить на экран пользователя диалоговое окно, с заданным сообщением. Код компонента вы можете найти ниже (листинг 13.2).

Листинг 13.2. Простейший COMпонент, созданный при помощи технологий .NET

```
/*
  Листинг 13.2
  File:   FunnyWorld.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за предоставление
// пользовательского интерфейса
using System.Windows.Forms;

// Подключим пространство имен, отвечающее за поддержку
// сервисов отражения
using System.Reflection;

// Укажем файл, где должны располагаться ключи, необходимые для
// подписывания нашей сборки
[assembly: AssemblyKeyFile("Keys.snk")]

// Это класс нашего компонента, он обязательно должен быть
// общедоступным, поскольку будет использоваться извне сборки,
// то есть из подсистемы COM, которая уж точно располагается
// за пределами данной сборки
public class FunnyWorld
{
  // Это единственный метод нашего компонента, он будет
  // вызываться из COM
  public void SayHello(String param)
```

```
  {
    // А для того чтобы информировать пользователя о том, что метод
    // был действительно вызван, выведем на экран диалоговое окно
    MessageBox.Show(param,"FunnyWorld::SayHello metnod was invoked");
  }
};
```

Из данного исходного кода нам необходимо сделать сборку, причем она обязательно должна быть строго именованной. Это требуется для того, чтобы поместить ее в GAC. Подробнее об этом будет рассказано далее.

Соответственно, необходимо создать файл с парой криптографических ключей. Это можно сделать при помощи команды:

```
sk -k Keys.snk
```

Теперь можно компилировать исходный код сборки, не забыв указать параметр `/target:library`

```
csc /target:library FunnyWorld.cs
```

Результатом компиляции должен явиться файл `FunnyWorld.dll`. Это строго именованная сборка, которую необходимо поместить в GAC:

```
gacutil /i FunnyWorld.dll
```

После выполнения данной команды утилита должна сообщить об успешном добавлении сборки в глобальное хранилище сборок.

```
Assembly successfully added to the cache
```

Регистрация сборки в GAC обязательна. Ведь сборка, грубо говоря, будет представлять собой COM-компонент, который должен быть доступен любому приложению в системе. Более подробно об участии сборки в процессе использования COM-компонента вы узнаете несколько позже.

А сейчас нам предстоит самый ответственный момент — регистрация компонента в среде COM. Для автоматизации данного процесса создана утилита `regasm`, которая позволяет осуществить задуманное при помощи единственного вызова

```
regasm FunnyWorld.dll
```

Если все было сделано правильно и сборка действительно является строго именованной, то утилита сообщит об успехе:

```
Types registered successfully
```

Вот и все. Наш компонент создан, зарегистрирован в системе COM и готов к работе.

Но как же им пользоваться, мы же не знаем `CLSID`, его единственного класса `FunnyWorld`. Оказывается, он нам и не понадобится в явном виде, так как

утилита `regasm` зарегистрировала для него в реестре `ProgID`. По умолчанию он равен имени класса, в нашем случае, `FunnyWorld`. Теперь, когда мы знаем `ProgID` класса, у нас есть вся необходима информация для создания приложения. Оно будет использовать наш COMпонент, созданный при помощи .NET. Осталось только выбрать, на каком языке писать приложение. Для того чтобы разнообразить меню используемых нами языковых средств, воспользуемся встроенными скриптовыми средствами DHTML (Dynamic HTML, динамический язык разметки гипертекста).

Ниже приведена Web-страница, которая обращается к нашему компоненту, написанная при помощи DHTML с использованием двух языков: JScript и VBScript (листинг 13.3).

Листинг 13.3. Использование .NET COMпонента при помощи средств DHTML

```
<!-
  Листинг 13.3
  File:   Some.html
  Author: Дубовцев Алексей
-->
<html>
  <body>
   <!-
     Этот тег отделяет код, написанный на JScript
   -->
  <script language="JScript">
    function OnJClick()
    {
      // Создадим наш объект, по его ProgID
      x = new ActiveXObject("FunnyWorld");
      // Обратимся к его единственному методу
      x.SayHello("Hello World from DHTML web page");
    }
  </script>
  <!-
    Этот тег отделяет код, написанный на JScript
  -->
  <script language="vbscript">
    Sub OnVBClick()
      Dim x
      ' Создадим наш объект, по его ProgID
```

```
      Set x = CreateObject("FunnyWorld")
      ' Обратимся к его единственному методу
      x.SayHello("Hello World from DHTML web page")
    End Sub
  </script>

   <!-  Отцентрируем страничку -->
   <center>
      <!- Поясним, для чего нужна эта страничка, не забыв выделить
     текст большим шрифтом -->
      <h1>This application test interoperability
       from COM to .NET</h1><br>
      <!- Создадим кнопку, нажатие которой приводит к вызову
          функции, написанной на JScript -->
      <input type="button" onClick="OnJClick()"↳
 value="Click here to test JScript"/>

      <!- Создадим кнопку, нажатие которой приводит к вызову
          функции, написанной на VBScript -->
      <input type="button" onClick="OnVBClick()"↳
 value="Click here to test VBScript"/>
      <!- Теперь осталось только закрыть теги -->
    </center>
  </body>
</html>
```

На Web-странице, создаваемой данным файлом, будут располагаться две кнопки, при нажатии любой из которых браузер выдаст предупреждение о попытке использования ActiveX-компонента (ActiveX- и COM-компоненты в данном случае — одно и то же) (рис. 13.4).

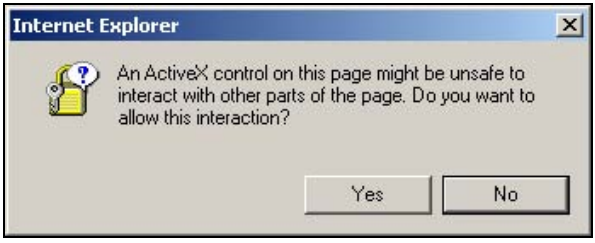


Рис. 13.4. Предупреждение о попытке взаимодействия DHML-кода с внешним компонентом

Для того чтобы разрешить работу нашего скрипта, необходимо нажать кнопку **Yes**. На экране появится диалоговое окно, выводимое нашим, недавно написанным компонентом (рис. 13.5).

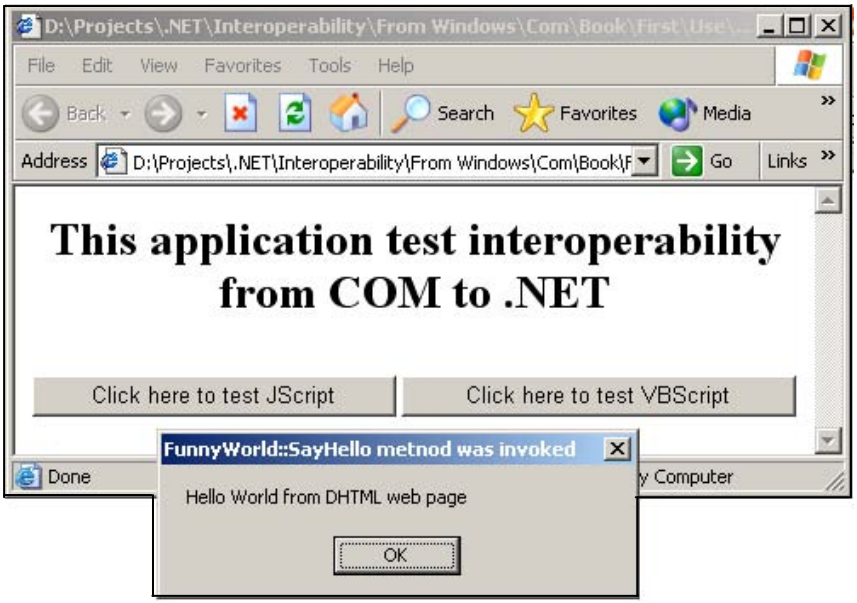


Рис. 13.5. Результат работы .NET COMпонента, вызванного из DHTML-кода

Итак, наш компонент работает. Теперь предстоит разобраться в том, как он устроен, а также рассмотреть более сложные способы работы с ним.

Нет никакого сомнения, что для понимания того, как действует компонент, необходимо, прежде всего, очень хорошо разбираться в идеологии и принципах технологии COM. Рассмотрим среду COM в подробностях.

Начнем с `ProgID`. Он использовался при создании нашего компонента

```
x = new ActiveXObject("FunnyWorld");
```

Здесь `FunnyWorld` — это и есть `ProgID` класса `FunnyWorld`. Каким же образом происходит создание столь сложного компонента всего лишь из одной строки кода? Для того чтобы ответить на вопрос, необходимо открыть раздел реестра `HKEY_CLASSES_ROOT\CLSID`, где найти `ProgID` нашего компонента (рис. 13.6).

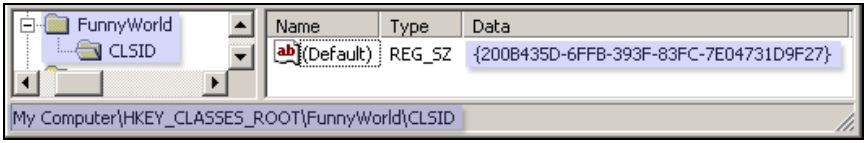


Рис. 13.6. Значение `ProgID` компонента `FunnyWorld`, указанное в реестре

Оказывается, `ProgID "FunnyWorld"` — это всего лишь ассоциативная ссылка на идентификатор класса `CLSID`. Он был сгенерирован автоматически утилитой `regasm` во время регистрации сборки в среде COM. Соответственно, для данного `CLSID` в реестре должна быть информация. Попробуем ее найти в том же подразделе (рис. 13.7).

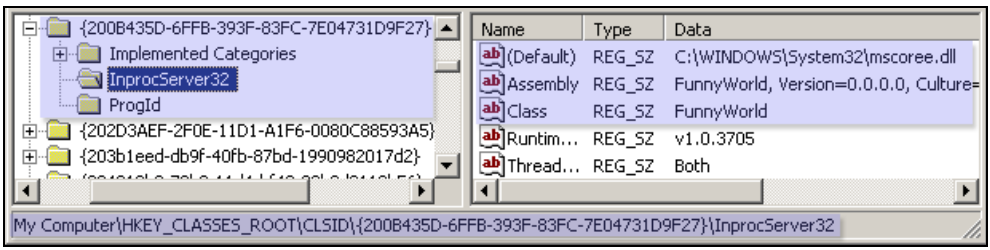


Рис. 13.7. Информация по компоненту `FunnyWorld`, указанная в реестре

Данный раздел содержит информацию, которую использует подсистема COM при создании компонента. Среди его подразделов наиболее интересным является `InprocServer32`. Именно в нем указано имя динамической библиотеки, в которой располагается COM-компонент, предоставляющий соответствующий класс. Эта динамическя библиотека будет загружена подсистемой COM в контекст приложения при обращении к функции `CoCreateInstance`, создающей COM-классы. В предыдущем примере обращение к данной функции было завуалировано конструкцией `new ActiveXObject`. Но что же мы видим — вместо сборки `FunnyWorld`, в данном подразделе указана библиотека `mscoree.dll`. Напомню, что в данной библиотеке расположен загрузчик виртуальной машины среды исполнения. Оказывается, что `mscoree.dll` экспортирует все четыре функции, обязательные для каждого COM-компонента. Они перечислены ниже:

```
File Type: DLL

  Section contains the following exports for mscoree.dll
    ordinal hint RVA      name
      ...
        52   13 0000B2AB DllCanUnloadNow
        53   14 00007F2A DllGetClassObject
        54   15 00011678 DllRegisterServer
        55   16 00010BE9 DllUnregisterServer
             ...
```

Подсистема COM, ничего не подозревая, загружает библиотеку `mscoree.dll` в адресное пространство приложения, запросившего создание нашего ком-

понента. В нашем случае это был Microsoft Internet Explorer. Затем подсистема COM обращается к функции `DllGetClassObject`, экспортируемой библиотекой.

```
// Данная функция позволят создать и получить в распоряжение
// объект (класс), расположенный в библиотеке

STDAPI DllGetClassObject(
  // Идентификатор объекта, который мы хотим создать
  const CLSID & rclsid,

  // Интерфейс, который мы хотим получить для доступа к данному
  // объекту, априори он должен быть IClassFactory или IClassFactory2
  const IID & riid,
  // А через данный параметр нам вернут указатель на интерфейс
  void ** ppv
);
```

Данная функция создает не сам класс, а его специальный служебный компонент, называемый фабрикой классов. Данный компонент предназначен для создания классов заданного типа. В нашем случае у `DllGetClassObject` запрашивается экземпляр фабрики классов для класса `FunnyWorld`. Соответственно, фабрика классов, которая будет возвращена, будет уметь создавать только классы типа `FunnyWorld`.

Для чего нужны фабрики классов

До этого мы рассматривали процессы, происходящие вне среды .NET. Но как только подсистема COM загружает библиотеку `mscoree.dll` и обращается к функции `DllGetClassObject`, мы попадаем во власть виртуальной машины .NET. Какие же шаги она предпринимает для создания компонента? Сначала среде исполнения необходимо выяснить — в какой сборке находится компонент, создание которого было у нее запрошено? Для этого она сканирует подраздел `(HKEY_CLASSES_ROOL\CLSID\[rclsid]\InprocServer32)` реестра для поиска `CLSID`, переданного в качестве первого параметра функции `DllGetClassObject`. Если в среду COM экспортируется действительно комплексный .NET-компонент, то в данном разделе будут располагаться подключи `Assembly` и `Class`. В нашем случае их значения таковы:

- `InprocServer32\Assembly - FunnyWorld, Version=0.0.0.0, Culture=neutral, PublicKeyToken=9efab99be5760e0a`
- `InprocServer32\Class - FunnyWorld`

В ключе `Assembly` указано полное строгое имя сборки, в которой расположен необходимый компонент. А в `Class` указан класс, который представляет экспортируемый COM-компонент.

Теперь картина создания COM-компонента полностью ясна. Представим ее в виде схемы (рис. 13.8).

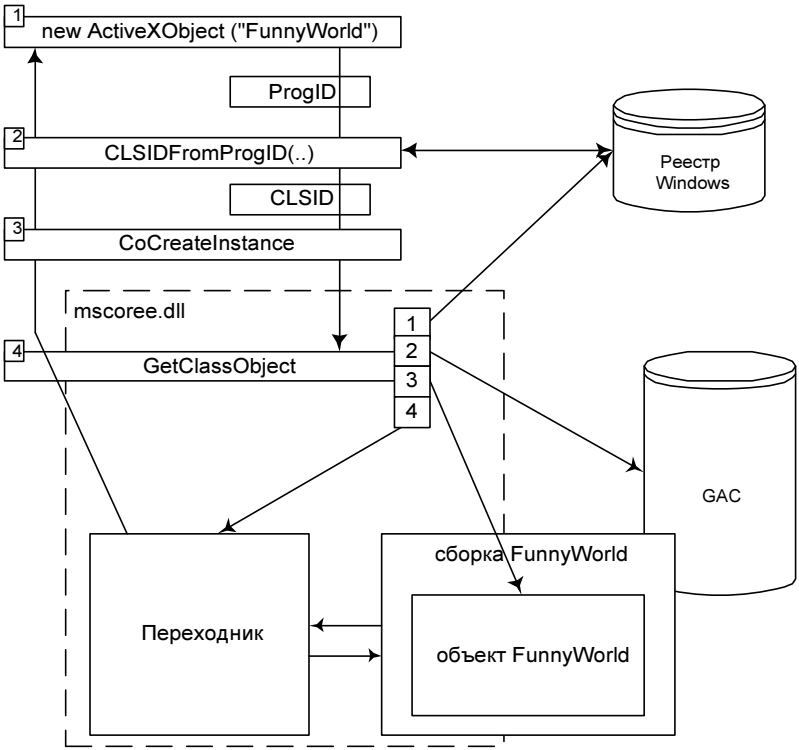


Рис. 13.8. Действие, происходящее во время загрузки .NET COMпонента

Приведем описание этапов загрузки .NET COMпонента:

1. Создание .NET COMпонента с верхнего уровня при помощи `ProgID`.
2. Получение настоящего идентификатора `CLSID` класса по `ProgID`, через косвенное обращение к реестру при помощи функции `CLSIDFromProgID`.
3. Создание компонента на основе реального идентификатора необходимого класса.
4. Попадаем внутрь среды исполнения через функцию `GetClassObject`, экспортируемую библиотекой `mscoree.dll`.
   4.1. Среда исполнения обращается к реестру и проверяет, является ли запрошенный класс .NET-компонентом. Если да — то загружает из

реестра все необходимые параметры: строгое имя сборки и имя класса, представляющего COM-компонент.

4.2. Среда исполнения загружает сборку из GAC.

4.3. Среда исполнения создает экземпляр класса, указанного в ключе `Class`, подраздела `InprocServer32`.

4.4. Создается специальный переходник для класса, отвечающий за взаимодействие с ним из среды COM.

Хотя данная схема является несколько упрощенной, все-таки она позволяет лучше понять процесс загрузки .NET COMпонента. А тогда становится ясным и назначение шагов, которые мы предпринимали при создании компонента.

Интерфейсы для COMпонента

В самом начале данной главы мы говорили, что для взаимодействия с любым COM-объектом обязательно должен использоваться интерфейс. Но как же так? В нашем предыдущем примере мы не объявляли ни одного интерфейса, и все прекрасно работало. Да и в скрипте, встроенном в Web-страницу, интерфейсы не применялись, по крайней мере, в явном виде. На самом деле интерфейсы присутствовали, просто их использование было скрыто от нас: со стороны компонента — средой исполнения, а со стороны Web-страницы — скриптовой машиной. Взаимодействие с нашим компонентом происходило при помощи сервисов автоматизации, а точнее интерфейса `IDispatch`. Это очевидно, хотя бы исходя из того, что скриптовая машина по-другому с COM-объектами общаться попросту не умеет. Для того чтобы окончательно развеять сомнения, обратимся к библиотеке типов COM нашего компонента. По умолчанию, она естественно не предоставляется, ее необходимо будет создать самостоятельно, при помощи специально предназначенной для этого утилиты `tlbexp`:

```
tlbexp FunnyWorld.dll
```

В результате мы должны получить в том же каталоге файл `FunnyWorld.tlb`, который и является библиотекой типов нашего компонента. С первого взгляда создание библиотеки типов COM для .NET-компонента может показаться странноватым. Но если учесть, что библиотека типов является всего лишь хранилищем информации о типах, то становится ясным, что делает утилита `tlbexp`. Она всего лишь преобразует метаданные в формат библиотек типов COM.

Чтобы просмотреть содержимое полученной библиотеки типов, необходимо воспользоваться утилитой Ole/Com Object Viewer (oleview.exe), выбрав пункт меню **File->ViewTypeLib**. У меня получился следующий результат (листинг 13.4). Должен сказать, что у вас точно такого же не получится, поскольку идентификаторы `CLSID` и `GUID` генерируются автоматически и совпадение маловероятно.

Листинг 13.4. Описание интерфейсов .NET COMпонента FunnyWorld на языке IDL

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: <could not determine filename>

[
  uuid(8012A2D0-C98F-30FC-ADAE-24598CBEF2F1),
  version(1.0),
  custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, FunnyWorld,
 Version=0.0.0.0, Culture=neutral, PublicKeyToken=9efab99be5760e0a)

]
library FunnyWorld
{
  // TLib :
  // TLib : Common Language Runtime Library : {BED7F4EA-1A96-11D2-8F08-
00A0C9A6186D}

  // Библиотека типов общей библиотеки классов среды исполнения
  importlib("mscorlib.tlb");
  // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
  importlib("stdole2.tlb");

  // Forward declare all types defined in this typelib
  interface _FunnyWorld;
  [
    uuid(200B435D-6FFB-393F-83FC-7E04731D9F27),
    version(1.0),
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, FunnyWorld)
  ]
  // Вот это, собственно, и есть определение нашего компонента
  coclass FunnyWorld {
    // Основной интерфейс данного класса
    [default] interface _FunnyWorld;
    // Также поддерживается специальный интерфейс для
    // доступа к членам класса предка Object
    interface _Object;
  };
```

```
  [
    odl,
    uuid(85876DCB-9342-34F1-A832-695F928F55B3),
    hidden,
    dual,
    oleautomation,
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, FunnyWorld)
  ]
  // А это определение нашего интерфейса, как видно по его предку
  // IDispatch, наш класс поддерживает автоматизацию
  interface _FunnyWorld : IDispatch
  {
  };
};
```

Прежде всего, в данном листинге стоит обратить внимание на одну мало заметную строчку, расположенную в самом его начале:

```
importlib("mscorlib.tlb");
```

Данная строка подключает определения типов из библиотеки типов `mscorlib.tlb`. В ней описаны типы, входящие в общую библиотеку классов COM. В нашем случае из нее используется интерфейс `_Object`, который предоставляет доступ к членам `Object` класса `FunnyWorld`, экспортируемого в среду COM. Данный класс, как, впрочем, и все классы в среде .NET, неявно является потомком от `System.Object`. То, что `FunnyWorld` экспортируется в COM, не отменяет его наследственные связи с `Object`. Следовательно, с ним должна быть связь! И она существует в лице интерфейса `_Object`. Вы можете запросить его у нашего класса и пользоваться его членами. Определение данного интерфейса для COM находится в библиотеке `mscorlib.tlb` и выглядит оно так:

```
[
  uuid(65074F7F-63C0-304E-AF0A-D51741CB4A8D),
  hidden,
  dual,
  nonextensible,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, System.Object)

]
dispinterface _Object {
```

```
  properties:
  methods:
   [id(00000000), propget,
    custom(54FC8F55-38DE-4703-9C4E-250351302B1C, 1)]
   BSTR ToString();
   [id(0x60020001)]
   VARIANT_BOOL Equals([in] VARIANT obj);
   [id(0x60020002)]
   long GetHashCode();
   [id(0x60020003)]
   _Type* GetType();
};
```

Как видите, заметно явное сходство с определением класса `Object` в управляемом коде.

Но вернемся к нашему компоненту: помимо интерфейса `_Object`, он поддерживает и `_FunnyWorld`. Это служебный, автоматически сгенерированный интерфейс, который необходим исключительно для работы сервисов автоматизации. Подобные интерфейсы генерируются для всех классов, экспортируемых в среду COM. Их имена всегда начинаются с символа подчеркивания. По умолчанию данный интерфейс не имеет собственных членов. Он использует только методы для работы с сервисами автоматизации (`IDispatch`).

Хотя, в нашем случае, для доступа к объекту автоматизация и используется, но к `_FunnyWorld` обращения не происходит. Данный интерфейс не имеет никакого отношения к тому `IDispatch`, который использовала скриптовая машина. Она запрашивала `IDispatch` непосредственно при создании компонента, примерно таким образом:

```
// Приблизительный вид кода, отвечающего за создание компонента,
// внутри виртуальной скриптовой машины
CoCreateInstance(
  // Идентификатор класса FunnyWorld
  CLSID_FunnyWorld,
  // Это агрегация, она скорее всего не использовалась
  NULL,
  // Использовать компонент в адресном пространстве родного процесса
  CLSCTX_INPROC_SERVER,
  // Передадим идентификатор интерфейса IDispatch, мы ведь
  // собираемся работать с компонентом через сервисы автоматизации
  __uuidof(IDispatch),
  reinterpret_cast<void**>(&pIDispatch));
```

При этом машина ничего не знала об интерфейсе _FunnyWorld. Таким образом, нам всегда гарантировано, что с компонентом можно будет связаться при помощи интерфейсов автоматизации.

Но, как мы уже выяснили ранее, это не самый быстрый способ работы с COM-компонентами. На радость нам, среда исполнения позволяет создавать настоящие интерфейсы, а также при желании отказаться от использования сервисов автоматизации. Для достижения этих целей предназначен атрибут ClassInterfaceAttribute.

```
public ClassInterfaceAttribute(
   ClassInterfaceType classInterfaceType
);
```

Его конструктор в качестве единственного параметра принимает член перечисления ClassInterfaceType, который определяет тип предоставляемого интерфейса. Члены перечисления описаны далее (табл. 13.1).

Таблица 13.1. *Описание членов перечисления System.Runtime.InteropServices.ClassInterfaceType, определяющий тип генерируемого интерфейса*

Член перечисления (Флаг)	Описание
AutoDispatch (Используется по умолчанию)	Предписывает предоставлять для данного класса доступ лишь через интерфейс Idispatch, а также служебный интерфейс _ИмяКласса. Члены класса через данный интерфейс предоставляться не будут, но будет предоставлен интерфейс _Object
AutoDual	Для класса будет сгенерировано оба интерфейса, IDispatch, а также служебный интерфейс, содержащий настоящие методы класса. Плюс, конечно же, интерфейс _Object
None	Предписывает подавлять генерирование всех служебных и специальных интерфейсов кроме тех, которые явно указаны в списке наследования

Продемонстрируем использование второго флага на нашем примере. Измененный исходный код приведен ниже (листинг 13.5).

Листинг 13.5. Пример компонента, поддерживающего прямой доступ к компоненту

```
/*
  Листинг 13.5
  File:   Some.cs
  Author: Дубовцев Алексей
```

```
*/

// Подключим основное пространство имен общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за предоставление
// пользовательского интерфейса
using System.Windows.Forms;

// Подключим пространство имен, отвечающее за динамическое управление типами
using System.Reflection;

// Укажем файл, где должны располагаться ключи, необходимые для
// подписывания нашей сборки
[assembly: AssemblyKeyFile("Keys.snk")]

// Это класс нашего компонента, он обязательно должен быть
// общедоступным, поскольку будет использоваться извне сборки,
// то есть из подсистемы COM, которая уж точно располагается
// за пределами данной сборки
[
  // Укажем, что для класса должны быть сгенерированы оба типа
  // интерфейсов, как для доступа при помощи сервисов автоматизации,
  // так и для прямого доступа
  ClassInterface(ClassInterfaceType.AutoDual)
]
public class FunnyWorld
{
  // Это единственный метод нашего компонента, экспортируемый
  // в среду COM
  public void SayHello(String param)
  {
    MessageBox.Show(param,"FunnyWorld::SayHello metnod was invoked");
  }
};
```

Теперь библиотека типов нашего компонента будет выглядеть совершенно по-другому. В целях экономии места, не будем полностью приводить ее код, а лишь рассмотрим изменения, относящиеся непосредственно к рассматриваемому вопросу.

```
// Само определение класса осталось без изменений
// он по-прежнему предоставляет два интерфейса
```

```
coclass FunnyWorld {
  [default] interface _FunnyWorld;
  interface _Object;
};

[
  odl,
  uuid(AE9CAA02-4373-39A2-B97C-BC83036FD2BC),
  hidden,
  dual,
  nonextensible,
  oleautomation,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, FunnyWorld)
]
// Данный интерфейс теперь явным образом поддерживает
// все члены, реализованные в данном классе, а также доступ
// к ним при помощи сервисов автоматизации
interface _FunnyWorld : IDispatch {
  // Сначала идут определения членов, пришедших из базовых
  // классов, в данном случае из Object
  [id(00000000), propget,
   custom(54FC8F55-38DE-4703-9C4E-250351302B1C, 1)]
  HRESULT ToString([out, retval] BSTR* pRetVal);
  [id(0x60020001)]
  HRESULT Equals([in] VARIANT obj,
                 [out, retval] VARIANT_BOOL* pRetVal);
  [id(0x60020002)]
  HRESULT GetHashCode([out, retval] long* pRetVal);
  [id(0x60020003)]
  HRESULT GetType([out, retval] _Type** pRetVal);

  // А здесь мы наконец-то добрались до членов, реализованных
  // самим классом FunnyWorld
 [id(0x60020004)]
 HRESULT SayHello([in] BSTR param);
};
```

Теперь можно работать с компонентом через настоящие интерфейсы, не прибегая к использованию интерфейсов автоматизации. Это должно несколько повысить скорость общения с компонентом.

Сделаем одно важное замечание по поводу атрибута `ClassInterfaceAttribute`. Действие данного атрибута, фактически, ограничивается только специальным интерфейсом (`_FunnyWorld`), если не учитывать базовую поддержку автоматизации, которую можно попросту отключить при помощи данного атрибута. На другие интерфейсы, определенные вручную, данный атрибут воздействия не оказывает.

Определение дополнительных пользовательских интерфейсов

При написании серьезных приложений, одних стандартных COM-интерфейсов, генерируемых автоматически при создании компонента, может не хватить. Наверняка потребуется вводить и использовать собственные интерфейсы. Следовательно, необходимо этому научиться. На самом деле, создавать и работать с такими интерфейсами даже проще, чем в настоящем COM. Для этого необходимо лишь ввести соответствующий управляемый интерфейс, а при создании объекта он будет автоматически преобразован в COM-интерфейс.

Для того чтобы продемонстрировать сказанное ранее, введем в наш пример дополнительный интерфейс `IFunnyWorld`. Он будет предоставлять единственный член `SayHello`, известный нам по предыдущей работе с компонентом. Таким образом, мы дадим пользователям возможность использования метода `SayHello` напрямую, через интерфейс `IFunnyWorld`. А если еще применить к классу атрибут `ClassInterface` (`ClassInterfaceType.AutoDual`), то интерфейсы автоматизации, впрочем, как и все служебные интерфейсы, будут полностью заблокированы и доступ к методу `SayHello` будет возможен только через `IFunnyWorld`.

Исходный код примера вы найдете ниже (листинг 13.6).

Листинг 13.6. Введение поддержки явного интерфейса

```
/*
  Листинг 13.6
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за предоставление
// пользовательского интерфейса
```

```
using System.Windows.Forms;

// Подключим пространство имен, отвечающее за динамическое управление типами
using System.Reflection;

// Укажем файл, где должны располагаться ключи, необходимые для
// подписывания нашей сборки
[assembly: AssemblyKeyFile("Keys.snk")]



// Введем дополнительный интерфейс, предоставляемый нашим
// компонентом, через который мы сможем использовать функцию
// SayHello
public interface IFunnyWorld
{
  void SayHello(String param);
};

// Это класс нашего компонента, он обязательно должен быть
// общедоступным, поскольку будет использоваться извне сборки,
// то есть из подсистемы COM, которая уж точно располагается
// за пределами данной сборки
[
  // Укажем, что для класса должны быть сгенерированы оба типа
  // интерфейсов, как для доступа при помощи сервисов автоматизации,
  // так и для прямого доступа
  ClassInterface(ClassInterfaceType.AutoDual)
]
// Добавим к предкам нашего класса интерфейс IFunnyWorld
public class FunnyWorld : IFunnyWorld
{
  // Это единственный метод нашего компонента
  public void SayHello(String param)
  {
    MessageBox.Show(param,"FunnyWorld::SayHello metnod was invoked");
  }
};
```

Скомпилируем данный пример. Затем получим из него библиотеку типов COM, которую изучим при помощи утилиты Ole/Com Object Viewer. Напомню последовательность команд, которые необходимо выполнить:

```
csc /target:library FunnyWorld.csc
tlbexp FunnyWorld.dll
oleview FunnyWorld.tlb
```

Ниже приводится исходный код библиотеки типов нашего компонента с комментариями (листинг 13.7).

Листинг 13.7. Исходный код библиотеки типов .NET COMпонента

```
/*
  Листинг 13.7
  File:   Some.idl
  Author: Дубовцев Алексей
*/

// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: FunnyWorld.tlb

[
  uuid(8012A2D0-C98F-30FC-ADAE-24598CBEF2F1),
  version(1.0),
  custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, FunnyWorld,↳
 Version=0.0.0.0, Culture=neutral, PublicKeyToken=9efab99be5760e0a)

]
library FunnyWorld
{
  // TLib :
  // TLib : Common Language Runtime Library :↳
{BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
  // Библиотека типов общей библиотеки классов среды исполнения
  importlib("mscorlib.tlb");
  // TLib : OLE Automation : {00020430-0000-0000-C000-000000000046}
  importlib("stdole2.tlb");

  // Forward declare all types defined in this typelib
```

```
interface IFunnyWorld;
interface _FunnyWorld;

[
  odl,
  uuid(52F18850-8756-3D52-8774-6D22B40819FC),
  version(1.0),
  dual,
  oleautomation,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, IFunnyWorld)

]
// Это введенный нами дополнительный интерфейс, при помощи
// которого мы можем получить доступ к интересующему нас методу
interface IFunnyWorld : IDispatch {
// Наш суперметод
  [id(0x60020000)]
  HRESULT SayHello([in] BSTR param);
};

[
  uuid(200B435D-6FFB-393F-83FC-7E04731D9F27),
  version(1.0),
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, FunnyWorld)
]

// Вот это, собственно, и есть определение нашего компонента
coclass FunnyWorld {
  // Служебный интерфейс
  [default] interface _FunnyWorld;
  // Интерфейс доступа к классу как к базовому объекту Object
  interface _Object;
  // Наш интерфейс
  interface IFunnyWorld;
};

[
  odl,
  uuid(AE9CAA02-4373-39A2-B97C-BC83036FD2BC),
```

```
    hidden,
    dual,
    nonextensible,
    oleautomation,
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, FunnyWorld)
   ]

   // Служебный интерфейс, который предоставляет как сервисы
   // автоматизации через методы интерфейса IDispatch, так и прямой
   // доступ ко всем методам нашего класса.
   interface _FunnyWorld : IDispatch {
    // Это функции основного базового класса Object
    [id(00000000), propget,
    custom(54FC8F55-38DE-4703-9C4E-250351302B1C, 1)]
    HRESULT ToString([out, retval] BSTR* pRetVal);
    [id(0x60020001)]
    HRESULT Equals([in] VARIANT obj,
                   [out, retval] VARIANT_BOOL* pRetVal);
    [id(0x60020002)]
    HRESULT GetHashCode([out, retval] long* pRetVal);
    [id(0x60020003)]
    HRESULT GetType([out, retval] _Type** pRetVal);

    // Наша функция.
    [id(0x60020004)]
    HRESULT SayHello([in] BSTR param);
  };
};
```

Теперь мы имеем совершенно полноценный COM-компонент, ничем не уступающий своим собратьям из настоящей среды COM.

В заключение данного раздела, дадим одну небольшую рекомендацию. Во избежание неожиданностей при создании собственных компонентов, перед каждым построением, проводите локальную инициализацию среды. Сделать это можно при помощи следующей последовательности команд, которые для удобства можно помесить в BAT-файл.

```
rem Удаляем регистрацию COM-объекта из реестра
regasm /u FunnyWorld.dll
rem Удаляем сборку из GAC
```

```
gacutil /u FunnyWorld.dll
rem Очищаем папку от мусора, образовавшегося в результате компиляции
rem нашего приложения (FunnyWorld.dll FunnyWorld.tlb)
nmake clean
rem Собираем заново наше приложение
nmake
rem Регистрируем в реестре общедоступные типы, расположенные в сборке
rem как COM-объекты
regasm FunnyWorld.dll
rem Помещаем нашу сборку в GAC
gacutil /i FunnyWorld.dll
rem Создаем из нашей сборки библиотеку типов
tlbexp FunnyWorld.dll
```

Если вы создаете свои приложения при помощи среды Visual Studio, то данные команды можно заставить выполняться каждый раз перед построением вашего приложения. Для этого их необходимо прописать в настройках Pre-Build Event вашего приложения (рис. 13.9).

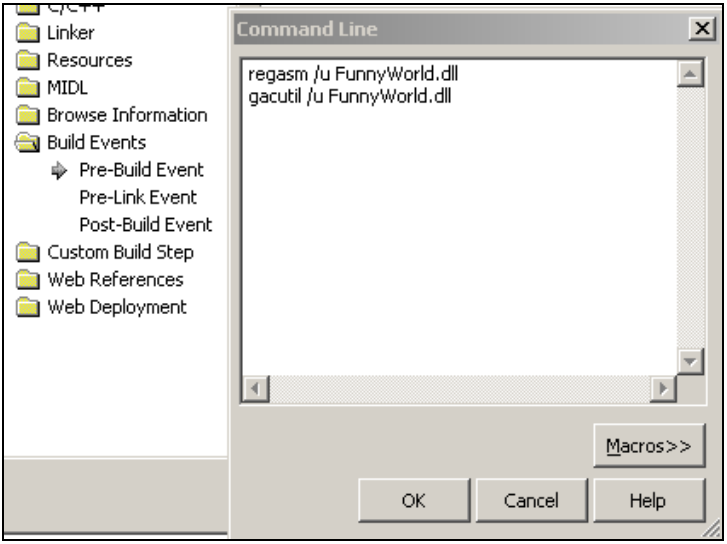


Рис. 13.9. Настройки для среды Visual Studio .NET, производящие автоматическую перерегистрацию разрабатываемого компонента

В большинстве случаев все будет прекрасно работать и без этого, но подобный подход застрахует вас от неприятных ситуаций. Например, вы, не заметив, что компонент не скомпилирован и соответственно обновлен, приметесь отлаживать приложение на старой версии.

Использование COMпонента напрямую

В данном разделе мы откажемся от простых способов использования COM-компонентов, как в примере со скриптом. И попробуем методы реализации на уровне API и некоторые другие.

Мы напишем несколько примеров использования компонентов, начиная с простых и заканчивая "очень" сложными.

Первый пример будет написан при помощи специальных классов поддержки взаимодействия с COM, введенными в компилятор. Дополнительно будет использована библиотека типов компонента. Для того чтобы подключить библиотеку к исходному коду, воспользуемся директивой `#import`, компилятора:

```
#import "FunnyWorld.tlb"
```

Помимо библиотеки типов самого компонента необходимо подключить библиотеку типов FCL среды исполнения .NET.

Полный код приложения представлен ниже (листинг 13.8).

Листинг 13.8. Пример взаимодействия с COMпонентом при помощи классов, входящих в поставку компилятора

```
/*
  Листинг 13.8
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Подключим Windows API
#include <windows.h>

// Подключим библиотеки типов, по которым будут построены классы,
// позволяющие облегчить работу с данными COM-объектами
#import "mscorlib.tlb"
// Конвертируем библиотеку типов в заголовки, понятные компилятору
#import "FunnyWorld.tlb"

// Точка входа в приложение
void main()
{
  // Инициализируем подсистему COM
  CoInitialize(NULL);
```

```
  // Опишем интерфейс, с которым будем работать, фактически
  // именно при исполнении данной строки будет создан наш объект
  // и запрошен соответствующий интерфейс

  // Используем специальный, интеллектуальный класс-указатель,
  // позволяющий легко работать с компонентом
  // данный класс был сгенерирован компилятором как следствие
  // применения директивы #import
  FunnyWorld::IFunnyWorldPtr↳
  pIFunnyWorld(__uuidof(FunnyWorld::FunnyWorld));

  // Вызовем метод нашего компонента
  pIFunnyWorld->SayHello(L"Hello World from C++");

  // Деинициализируем подсистему COM
  CoUninitialize();
};
```

В результате работы данного приложения на экране появится уже знакомое нам окно, но только с другим приветствием (рис. 13.10).

Рис. 13.10. Диалоговое окно с приветствием, выведенное на экран .NET COMпонентом

Приложение оказалось простым и красивым, если, конечно, не знать, что в реальности скрывается за его кодом. При работе компилятор генерирует груды ненужного обобщенного кода, годного на все случаи жизни. Для создания объекта, мы использовали класс `FunnyWorld::IFunnyWorldPtr`, который можно легко перепутать с определением самого интерфейса `IFunnyWorld`, если не заметить постфикса `Ptr`. Это класс интеллектуальной сборки, он автоматически генерируется компилятором при использовании директивы `#import`. В следующем примере мы сознательно откажемся от этой особенности компилятора при помощи модификатора `raw_interfaces_only`.

```
#import "FunnyWorld.tlb" raw_interfaces_only
```

Он позволяет отказаться от автоматической генерации интеллектуальных классов заглушек, предназначенных для облегчения взаимодействия с компонентом.

К тому же отпадет надобность в использовании библиотеки типов `mscorlib.tlb`. Она была необходима лишь для генерации вышеупомянутых интеллектуальных классов-переходников.

Код нового приложения выглядит так (листинг 13.9).

Листинг 13.9. Использование .NET COMпонента напрямую, без помощи интеллектуальных классов-переходников

```
/*
  Листинг 13.9
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Подключим стандартный заголовочный файл Windows API
#include <windows.h>

// Конвертируем библиотеку типов в заголовки, понятные компилятору,
// не забывая при этом указать ему, что нам нужны только
// определения интерфейсов и генерировать дополнительные
// классы не следует
#import "FunnyWorld.tlb" raw_interfaces_only

// Точка входа в приложение
void main()
{
  // Инициализируем подсистему COM
  CoInitialize(NULL);

  // При помощи данного интерфейса мы будем получать доступ к
  // нашему объекту
  // Это специальный интерфейс, введенный нами вручную
  FunnyWorld::IFunnyWorld* pIFunnyWorld;

  // Пришло время создать объект, первым параметром идет
  // идентификатор класса
```

```
  // По идее, мы не знаем идентификатор, потому что он автоматически
  // генерировался при создании компонента, но он указан в библиотеке
  // типов, ее услужливо разобрал компилятор и встроил в код
  // все нужные нам идентификаторы
  CoCreateInstance( __uuidof(FunnyWorld::FunnyWorld),
         // Данный параметр необходим для управления
         // агрегацией и в данном случае не нужен
         NULL,
         // Загружам компонент внутрь нашего процесса,
         // как dll
         CLSCTX_INPROC_SERVER,
         // Передаем идентификатор интерфейса, который
         // хотим получить для доступа к объекту
         __uuidof(FunnyWorld::IFunnyWorld),
         // А сюда нам вернут указатель на интерфейс
         reinterpret_cast<void**>(&pIFunnyWorld));


  // Наша функция в качестве параметра принимает строку типа BSTR,
  // не путать с Unicode!
  // Для того чтобы сконвертировать обычную Unicode-строку в BSTR,
  // можно воспользоваться классом, поставляемым вместе с
  // компилятором, в рамках поддержки технологии COM
  // Но в данном случае он использоваться не будет.
  //pIFunnyWorld->SayHello(*(new _bstr_t(L"Hello World from C++")));

  // Если же классы, поставляемые с компилятором, вам чем-либо
  // не приглянулись, можно то же самое сделать на чистом Windows API
  BSTR bstrParam = SysAllocString(L"Hello World from C++");
  pIFunnyWorld->SayHello(bstrParam);
  SysFreeString( bstrParam);

  // Освобождаем интерфейс, он нам больше не нужен
  // тем самым мы удалим объект
  pIFunnyWorld->Release();

  // Деинициализируем подсистему COM
  CoUninitialize();
}
```

Как оказалось, написание примера на чистом API ненамного сложнее. Нужно лишь аккуратно передать необходимые параметры и не перепутать идентификаторы необходимого нам класса и интерфейса.

Единственным неприятным моментом, при написании программ подобным способом, является подсчет ссылок и управление временем жизни интерфейсов. В нашем приложении все свелось лишь к единственному вызову:

```
pIFunnyWorld->Release();
```

В более серьезных приложениях придется подсчитывать ссылки при помощи `AddRef` и `Release` методов интерфейса `IUnknown`. Именно поэтому многие программисты предпочитают использовать классы поддержки COM, которые автоматически производят подсчет ссылок и самостоятельно управляют временем жизни интерфейсов. Такие классы устроены достаточно просто — все методы, приводящие к дублированию указателя на интерфейс, автоматически вызывают `AddRef`, а в деструкторе класса находится обращение к `Release()`. Так происходит управление временем жизни COM-классов, причем совершенно прозрачно для программиста. Но, для того чтобы грамотно пользоваться данными классами, необходимо иметь четкое представление об их внутренней организации и структуре. К примеру, можно совершенно уверенно заявить, что в первом примере есть ошибка. Попробуйте найти ее! Без знания внутренних механизмов работы классов поддержки COM вам вряд ли это удастся. А ошибка такая — после окончания работы с интерфейсом необходимо освободить его, обратившись к методу `Release`. Класс делает это из своего деструктора, который вызывается автоматически при выходе объекта из текущей области видимости, в нашем случае это произойдет на выходе из функции `main`. Таким образом, после того как отработает функция `main`, будет вызван деструктор класса, но к этому моменту подсистема COM будет уже деинициализирована. В последней строке функции `main` происходит обращение к функции `CoUninitialize`, которая отключает подсистему COM. В нашем случае, это не важно, поскольку приложение все равно завершает свою работу и компонент автоматически выгружается.

Данный пример особенно наглядно демонстрирует необходимость понимания внутреннего устройства и механизмов работы используемых классов.

Автоматизация вручную

Оба предыдущих примера использовали библиотеку типов нашего компонента, которую мы создавали сами. В то же время, при использовании компонента с Web-страницы при помощи DHTML никаких дополнительных библиотек, кроме самого компонента, явно не использовалось (на тот момент они еще не были созданы). Компонент использовался только при помощи сервисов автоматизации. Данный процесс был подробно описан ранее, здесь же мы попробуем воспроизвести его в программе, код которой вы найдете далее (листинг 13.10).

Листинг 13.10. Использование сервисов автоматизации в ручном режиме

```
/*
  Листинг 13.10
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Подключим необходимые заголовочные файлы
#include <windows.h>
#include <comdef.h>
#include <stdio.h>
#include <objbase.h>

// Точка входа в приложение
void main()
{
   // Инициализируем подсистему COM
   CoInitialize(NULL);

   // Это именно тот самый интерфейс, на котором держится
   // практически вся автоматизация COM
   IDispatch* pIDispatch;


  // В данной переменной мы будем хранить CLSID нашего компонента
  CLSID CLSID_FunnyWorld;
  // Получим идентификатор класса по его ProgID
  HRESULT hr = CLSIDFromProgID(L"FunnyWorld",&CLSID_FunnyWorld);

  // Проверим, удалось ли получить идентификатор, если нет, то
  // класс, скорее всего, не зарегистрирован
  if ( !SUCCEEDED(hr))
  {
    // Сообщим пользователю об ошибке
    printf("ProgID not registered");
    // Компонент не зарегестрирован, дальнейшая работа
    // приложения бессмысленна
    return;
```

```
}

// Создадим наш объект, запросив для взаимодействия с ним
// интерфейс IDispatch
hr = CoCreateInstance(
  // Идентификатор класса, который требуется
  // создать.
  CLSID_FunnyWorld,
  // Этот параметр используется для агрегирования, сегодня он
  // нам не понадобится
  NULL,
  // Динамическая библиотека компонента
  // будет загружена в адресное пространство
  // нашего процесса
  CLSCTX_INPROC_SERVER,
  // Идентификатор интерфейса, который нам
  // необходим для доступа к объекту
  IID_IDispatch,
  // А сюда мы получим указатель на
  // необходимый нам интерфейс
  reinterpret_cast<void**>(&pIDispatch));

// Проверим, получилось ли создать компонент
if ( !SUCCEEDED(hr) )
{
  // Сообщим пользователю, что не удалось создать комонент
  // или получить необходимый интерфейс
  printf("Can't get IDispatch interface or create class\n");
  // Компонент не зарегестрирован, дальнейшая работа
  // приложения бессмысленна
  return;
}

// Название метода, который мы хотим вызвать. В данном случае
// на его месте может быть любой член класса, к которому мы хотим
// получить доступ
// Напомню, что автоматизация позволяет организовать доступ
// к членам интерфейсов по их именам
OLECHAR* szMember = L"SayHello";
```

```
// А это идентификатор члена класса, при помощи которого
// производится доступ.
DISPID dispid;


// Запросим нужный нам идентификатор по имени члена
hr = pIDispatch->GetIDsOfNames(
  // Так надо
  IID_NULL,
  // Имя члена
  &szMember,
  // Запрос производится только
  / для одного члена
  1,
  // Стандартная региональная информация
  LOCALE_SYSTEM_DEFAULT,
  // Сюда мы получим запрошенный
  // идентификатор члена
  &dispid);

// Надо проверить — удался ли запрос идентификатора, если нет, то,
// вероятнее всего, такого члена не существует
if ( !SUCCEEDED(hr) )
{
  // Сообщим пользователю, что он пытался обратиться к
  // к несуществующему члену
  printf("Can't find member");
  // Компонент не зарегестрирован, дальнейшая работа
  // приложения бессмысленна
  return;
}

// Это еще понадобится как временная переменная
unsigned int uiSome;

// Сюда нам положат результат работы нашего метода
VARIANT varResult;
```

```
// Здесь будут храниться параметры, которые мы хотим передать
// методу
DISPPARAMS params;

// А это конкретный параметр, который будет передан — он у нашей
// функции один
VARIANTARG arg1;

// Преобразуем строку из Unicode в BSTR
BSTR bstrHello = SysAllocString(L"Hello World from IDispatch");


// Определим тип параметра
arg1.vt = VT_BYREF | VT_BSTR;
// Зададим значение параметра
arg1.pbstrVal = &bstrHello;

// Присоединим параметр к общему списку параметров
params.rgvarg = &arg1;
// У нас всего один позиционный параметр
params.cArgs = 1;
// И ни одного именованного
params.cNamedArgs = 0;

// А вот и он, метод Invoke в действии. Здесь будет
// совершен вызов метода нашего объекта
pIDispatch->Invoke(
  // Передадим идентификатор члена, к которому мы
  // хотели бы получить доступ
  dispid,
  // Так надо
  IID_NULL,
  // Региональная информация - стандартная
  LOCALE_SYSTEM_DEFAULT,
  // Указываем на то, что хотим вызвать метод
  DISPATCH_METHOD,
  // Это параметры, которые мы хотим передать
  // методу
  &params,
```

```
    // Это результат работы нашей функции
    &varResult,
    // Так надо.
    NULL,
    // И так тоже.
    &uiSome);


  // Удалим ранее преобразованную BSTR-строку
  SysFreeString(bstrHello);

  // Освободим не нужный более интерфейс
  pIDispatch->Release();

  // Деинициализируем подсистему COM.
  CoUninitialize();
}
```

В результате работы приложения, на экране появится знакомое диалоговое окно с сообщением: "Hello World from Idispatch".

Кто бы мог раньше подумать, что данный код по своей природе полностью идентичен следующему:

```
x = new ActiveXObject("FunnyWorld");
x.SayHello("Hello World from IDispatch");
```

Если не знать сути, то сходство между ними найти очень сложно.

Использование сервисов автоматизации прекрасно подходит для использования в скриптовых языках, для которых другой альтернативы не существует. Но для серьезных приложений такой подход не приемлем — лучше обращаться к компоненту напрямую, что будет работать на порядок быстрее. Для полного представления о преимуществах использования прямого вызова приведем реальный низкоуровневый код взаимодействия с COMпонентом (листинг 13.11).

Листинг 13.11. Реальный машинный код взаимодействия с COMпонентом, исполняющийся процессором

```
// В коде параметры функций интерфейсов передаются в обратном порядке

// Третий параметр функции
lea eax,pIHello
```

```
// Помещаем его в стек
push eax


// Вычисляем адрес второго параметра, функции он будет
// передаваться по ссылке
lea eax,iid
// Помещаем второй параметр в стек
push eax

// Помещаем в стек первый параметр
push DWORD ptr 0

// Помещаем в регистр указатель на интерфейс
mov eax,pClassFactory
// Сохраняем в стеке указатель на интерфейс, по которому будем
// обращаться к функции
push eax

// Вычисляем адрес таблицы интерфейса
mov eax,[eax]

// Производим вызов третьего метода интерфейса
// 0Ch = 12 байт, на каждый указатель в 32-битной среде 4 байта
// итого 12/4 = 3, три указателя
call [eax + 0Ch]

// Записываем в переменную значение, возвращенное функции
mov hr,eax
```

После компиляции данный код будет занимать 30 байт, которые процессор проглотит, даже не заметив. Очевидно, что при использовании автоматизации, объемы генерируемого кода будут несравнимо большими.

К данной информации необходимо относиться с умом, учитывая относительную важность для конкретного проекта производительности приложения и скорости его разработки. Приложения, создаваемые с помощью скриптовых средств, хотя и более медленные, зато сроки разработки короче. Хотя бывает, что профессионал, использующий C++, создает приложение быстрее, чем при использовании RAD (Rapid Application Development, Средства быстрой разработки приложений).

Работа с COMпонентом в обход подсистемы COM

Для того чтобы полностью осознать механизмы и принципы работы нашего компонента, откажемся от использования сервисов, предоставляемых подсистемой COM, и сделаем всю работу за нее вручную.

В данном разделе будет приведено два примера: первый — упрощенный, с использованием только стандартных языковых средств, во втором — применяются более сложные низкоуровневые технологии, вскрывающие самые темные недра технологии COM. Но по сути дела оба примера являются одинаковыми. Взаимодействие с компонентом ведется при помощи введенного нами дополнительного интерфейса `IFunnyWorld`. Компонент рассматривается не как некий COM-объект, а как обычная динамическая библиотека, и работа с ней ведется через обычный Windows API. Для большей наглядности можно было бы, конечно, загружать библиотеку вручную, без API. Но это увело бы нас в сторону от понимания цели приложения. Дополнительно используется библиотека типов компонента. Без нее также можно было бы обойтись, но тогда в приложение необходимо было бы ввести дополнительный код, считывающий необходимую информацию из реестра. Код первого приложения представлен ниже (листинг 13.12).

Листинг 13.12. Использование .NET COMпонента в обход подсистемы COM, но при использовании встроенных возможностей компилятора

```
/*
  Листинг 13.12
  File:   Some.cpp
  Author: Дубовцев Алексей
*/

// Подключаем стандартные заголовочные файлы
// обратите внимание, среди них нет ни одного
// из необходимых для работы с подсистемой COM
#include <windows.h>
#include <stdio.h>

// Это нам понадобится для получения значений идентификаторов
#import "FunnyWorld.tlb" raw_interfaces_only

// Это тип указателя на функцию, экспортируемую из
// динамической библиотеки нашего компонента
typedef HRESULT (__stdcall *typepDllGetClassObject) (
  const CLSID & rclsid,
```

```
  const IID & riid,
  void ** ppv
);

// А это сам указатель, который мы будет использовать для того,
// чтобы вызывать функцию DllGetClassObject
typepDllGetClassObject pDllGetClassObject;


// Точка входа в приложение
void main()
{
  // Описатель динамической библиотеки нашего компонента
  // и среды исполнения по совместительству
  HANDLE hLibrary;

  // Загружаем библиотеку, одновременно представляющую и загрузчик
  // виртуальной машины и динамическую библиотеку, в которой
  // как бы расположен компонент
  hLibrary = LoadLibrary("mscoree.dll");

  // Загрузим из динамической библиотеки функцию, которая отвечает
  // за предоставление фабрики классов компонента
  // На самом деле мы узнаем адрес необходимой нам функции
  pDllGetClassObject = (typepDllGetClassObject)GetProcAddress(↳
(HMODULE)hLibrary,"DllGetClassObject");

  // Указатель на интерфейс, предоставляющий доступ к
  // фабрике класса нашего компонента
  IClassFactory* pClassFactory;

  // Сюда мы будем получать результаты работы некоторых функций
  HRESULT hr;

  // Обратимся к динамической библиотеке, чтобы получить фабрику
  // классов нашего компонента
  hr = pDllGetClassObject(
        // Идентификатор компонента, для которого
```

```
      // требуется получить фабрику классов
      __uuidof(FunnyWorld::FunnyWorld),
      // Передадим идентификатор интерфейса
      // фабрики классов
      __uuidof(IClassFactory),
      // Сюда нам вернут указатель на интерфейс
      reinterpret_cast<void**>(&pClassFactory));

// Проверим, удался ли запрос фабрики классов
if ( !SUCCEEDED(hr) )
{
  // Сообщим пользователю о том, что фабрику
  // классов компонента получить не удалось
  printf("Hm can't get Class Factory interface");
  // Не удалось получить фабрику классов,
  // дальнейшая работа компонента бессмысленна
  return;
}

// При помощи данного интерфейса мы будет обращаться
// к нашему объекту
FunnyWorld::IFunnyWorld* pIFunnyWorld;

// Запросим у фабрики классов создание экземпляра компонента
hr = pClassFactory->CreateInstance(
  // Опять все та же агрегация, которой мы
  // пользоваться не будем
  NULL,
  // Это интерфейс, который мы хотим получить
  // от объекта
  __uuidof(FunnyWorld::IFunnyWorld),
  // А сюда нам вернут запрошенный интерфейс
  reinterpret_cast<void**>(&pIFunnyWorld));

// Проверим, удалось ли создание объекта и получение
// соответствующего интерфейса
if ( !SUCCEEDED(hr) )
{
```

```
    // Сообщим пользователю о том, что создание объекта не удалось
    printf("Hm can't get IFunnyWorld interface or create object");
    // Поскольку создание объекта не удалось,
    // дальнейшая работа приложения бессмысленна
    return;
  }

  // Преобразуем Unicode-строку в BSTR
  BSTR bstrParam = SysAllocString(L"Hello World from C++");

  // А это и есть обращение к методу нашего компонента
  pIFunnyWorld->SayHello(bstrParam);

  // Удалим созданную ранее строку
  SysFreeString( bstrParam);

  // Освободим не нужные более интерфейсы
  pIFunnyWorld->Release();
  pClassFactory->Release();

  // Выгрузим библиотеку вместе со средой исполнения.
  FreeLibrary(hLibrary);


  // Обратите внимание, привычной деинициализации подсистемы
  // COM нет, потому что она не использовалась
}
```

Поздравляю, теперь вы знаете, чем занимается функция `CoCreateInstance`. Как оказалось все просто и понятно, за исключением, пожалуй, момента с использованием фабрики класса. При первоначальном знакомстве с COM совершенно непонятно, для чего они введены и почему нельзя было создавать компоненты напрямую при помощи функции `DllGetClassObject`. Соглашусь, в данном примере использование фабрики классов избыточно, но при написании распределенных сетевых приложений фабрики классов незаменимы для поддержки возможности создания компонентов по сети.

Теперь приведем исходный код второго приложения, общающегося с компонентом на самом низком уровне (листинг 13.13).

Листинг 13.13. Использование .NET COMпонента в обход подсистемы COM, без использования встроенных возможностей компилятора

```
/*
  Листинг 13.13
  File:   Some.cpp
*/

// Подключаем стандартные заголовочные файлы
// обратите внимание, среди них нет ни одного
// из необходимых для работы с подсистемой COM
#include <windows.h>
#include <stdio.h>

// Это нам понадобится для получения значений идентификаторов
#import "FunnyWorld.tlb" raw_interfaces_only

// Точка входа в приложение
void main()
{
  // Описатель динамической библиотеки нашего компонента
  HMODULE hLibrary;

  // Загружаем библиотеку, одновременно представляющую и загрузчик
  // виртуальной машины и динамическую библиотеку, в которой
  // как бы расположен компонент
  hLibrary = LoadLibrary("mscoree.dll");

  // Загружаем из динамической библиотеки функцию, которая отвечает
  // за предоставление фабрики классов компонента
  // На самом деле мы узнаем адрес необходимой нам функции
  PUINT dwDllGetClassObject = (PUINT)GetProcAddress(↳
hLibrary,"DllGetClassObject");

  // Указатель на интерфейс, предоставляющий доступ к
  // фабрике класса нашего компонента
  IClassFactory* pClassFactory;

  // Это идентификатор компонента, мы получим его по данным,
```

```
// указанным в библиотеке
CLSID CLSID_FunnyWorld = __uuidof(FunnyWorld::FunnyWorld);

// Результат работы функции
HRESULT hr;

// Обратимся к динамической библиотеке, чтобы получить фабрику
// классов нашего компонента
// Для того чтобы не вводить указатель на функцию,
// здесь используется ассемблерная вставка, мне кажется,
// что ее использование гораздо удобнее, чем возможность описывать
// тип указателя на функцию и вводить дополнительную
// переменную
// DllGetClassObject(
//      __uuidof(FunnyWorld::FunnyWorld),
//      __uuidof(IClassFactory),
//      reinterpret_cast<void**>(&pClassFactory));
__asm  {
   // Вычислим адрес переменной pClassFactory
   lea   eax,pClassFactory
   // Помещаем его в стек - третий параметр
   push  eax
   // Вычисляем адрес идентификатора интерфейса IClassFactory
   lea   eax,IID_IClassFactory
   // Помещаем его в стек - второй параметр
   push  eax
   // Вычисляем адрес идентификатора класса FunnyWorld
   lea   eax,CLSID_FunnyWorld
   // Помещаем его в стек - третий параметр
   push  eax
   // Обращаемся к функции
   call  dwDllGetClassObject
   // Сохраняем значение, возвращенное функцией
   mov   hr,eax
}

// Проверим, удался ли запрос фабрики классов
if ( !SUCCEEDED(hr) )
{
```

```
  // Сообщим пользователю, что создание фабрики классов
  // компонента не удалось
  printf("Hm can't get Class Factory interface");
  // Поскольку создание фабрики классов не удалось,
  // дальнейшая работа приложения бессмысленна
  return;
}

// При помощи данного интерфейса мы будет обращаться
// к нашему объекту
FunnyWorld::IFunnyWorld* pIFunnyWorld;

// Запросим у фабрики классов создание экземпляра компонента
hr = pClassFactory-> CreateInstance(
        // Опять все та же агрегация
        NULL,
        // Это интерфейс, который мы хотим получить
        // от объекта
        __uuidof(FunnyWorld::IFunnyWorld),
        // А сюда нам вернут запрошенный интерфейс
        reinterpret_cast<void**>(&pIFunnyWorld));

// Поверим, удалось ли создание объекта и получение
// соответствующего интерфейса
if ( !SUCCEEDED(hr) )
{
  // Сообщим пользователю о том, что создание объекта не удалось
  printf("Hm can't get IFunnyWorld interface");
  // Поскольку создание компонента не удалось,
  // дальнейшая работа приложения бессмысленна
  return;
}

// Преобразуем Unicode-строку в BSTR
BSTR bstrParam = SysAllocString(L"Hello World from C++");

// А это и есть обращение к методу нашего компонента
pIFunnyWorld->SayHello(bstrParam);

// Удалим ранее созданную строку
```

```
  SysFreeString(bstrParam);

  // Освободим не нужные более интерфейсы
  pIFunnyWorld->Release();
  pClassFactory->Release();

  // Выгрузим библиотеку вместе со средой исполнения.
  FreeLibrary(hLibrary);

  // Обратите внимание, привычной деинициализации подсистемы
  // COM нет, потому что она не использовалась
}
```

Фактически, данные примеры являются демонстрацией внутреннего устройства подсистемы COM, в упрощенном виде. Сама подсистема COM устроена, конечно, гораздо сложнее, поскольку ей необходимо учитывать общие случаи. Мы же рассматривали всего лишь отдельно взятый вариант.

Дополнительные сервисы, необходимые при взаимодействии с COM

Из предыдущих разделов вы узнали, как создавать собственные COMпоненты при помощи .NET. Но все эти компоненты были всего лишь тестовыми. Для того чтобы разрабатывать полноценные COMпоненты, вам понадобится знание механизмов и некоторых особенностей, которые описаны далее.

ProgID — зачем и как

В примерах, созданных ранее, `ProgId` генерировался автоматически при регистрации компонента. А это не совсем хорошо, если не сказать — плохо. При автоматической генерации вероятны коллизии имен `ProgId`. Например, вполне могут совпасть имена класса и соответствующего `ProgId`. Во избежание проблем, рекомендуется в `ProgId` включать имя приложения. В нашем случае можно сделать, например, так:

```
Sample.FunnyWorld
```

Для того чтобы самостоятельно задать `ProgId`, необходимо воспользоваться атрибутом `ProgIdAttribute`. Прототип его конструктора представлен ниже:

```
public ProgIdAttribute(
   string progId
);
```

При задании ProgId необходимо быть очень осторожным. Так, совпадение с уже существующим именем приведет к непоправимой коллизии: один из ProgId будет ссылаться на совершенно другой компонент. Для того чтобы полностью исключить совпадение ProgId, можно помимо имени приложения включать имя вашей компании:

```
[
        ProgId("VasiliyPupkinCoporation.Sample.FunnyWorld")
        ClassInterface(ClassInterfaceType.AutoDual)
]
public class FunnyWorld : IFunnyWorld
```

Видимость .NET-классов для COM

По умолчанию все общедоступные (public) программные элементы (на уровне всей сборки) экспортируются в среду COM. Это относится как к классам, так и к интерфейсам. Для того чтобы предотвратить экспортирование класса или интерфейса, но при этом сохранить его общедоступность на уровне сборки, необходимо воспользоваться атрибутом ComVisibleAttribute.

```
public ComVisibleAttribute(
   bool visibility
);
```

Если в качестве параметра передать false, то данный программный элемент будет невидим для COM.

Явное задание идентификатора COM для классов (*CLSID*) и интерфейсов (*IID*)

При создании компонента среда исполнения автоматически генерирует все необходимые идентификаторы как для классов, так и для интерфейсов. При их создании учитывается информация о версии сборки, экспортируемой в среду COM. Это позволяет создавать для идентичных версий сборок одинаковые идентификаторы. Таким образом, можно не опасаться, что после перекомпиляции приложения они будут изменены. Помимо этого, существует возможность сохранить идентификаторы неизменными независимо от версии сборки. Для этого предназначен атрибут ComCompatibleVersionAttribute. Ему лишь необходимо указать версию сборки, которая будет использоваться при генерации идентификаторов.

Автоматическая генерация — вещь, конечно, полезная. Но она будет неприемлема, если вам необходимо использовать собственные, заранее известные значения идентификаторов. Для этого придется воспользоваться атрибутом Guid.

```
public GuidAttribute(
   string guid
);
```

Самое интересное, что при помощи `Guid` вы можете задавать как идентификаторы классов (`CLSID`), так и идентификаторы интерфейсов (`IID`). После применения данного атрибута, автоматическая генерация идентификатора для него будет отключена.

То, что не вошло в эту книгу

Помимо описанных интерфейсов и атрибутов, среда .NET содержит огромное количество других, предназначенных для поддержки взаимодействия с COM. К сожалению, ограниченный объем книги никак не позволяет описать их все. Для этого пришлось бы писать отдельный том — только о взаимодействии с COM. Но чтобы иметь общее представление о сервисах, описание которых не вошло в книгу, здесь приводится их перечисление с краткими пояснениями (табл. 13.2).

Таблица 13.2. *Список интерфейсов, предназначенных для взаимодействия с COM, описание которых не вошло в книгу*

Имя класса	Пояснение
	Атрибуты
`AutomationProxyAttribute`	Позволяет выбирать между пользовательским и стандартным маршалером
`ComAliasNameAttribute`	Позволяет указать тип, который будет использоваться при создании библиотеки типов
`ComCompatibleVersionAttribute`	При генерации библиотек типов для компонентов утилита `tlbexp` вычисляет идентификаторы, учитывая версию сборки. Соответственно при изменении версии, идентификаторы также станут другими. Атрибут позволяет предотвратить изменение идентификаторов от версии к версии
`ComConversionLossAttribute`	Применяется утилитой `tlbimp` для указания на то, что при преобразовании библиотеки типов в сборку была утеряна некоторая информация
`ComRegisterFunctionAttribute`	Позволяет ввести пользовательскую функцию регистрации компонента в реестре. Обычно вызывается утилитой `Regasm` или программой установки
`ComSourceInterfacesAttribute`	Позволяет ввести пользовательский интерфейс для поддержки системы событий COM

Таблица 13.2 *(окончание)*

Имя класса	Пояснение
Атрибуты	
`ComUnregisterFunctionAttribute`	Позволяет ввести пользовательскую функцию, которая будет вызываться при удалении COMпонента. Обычно вызывается утилитой `Regasm` или программой деинсталляции
`ImportedFromTypeLibAttribute`	Указывает на то, что данный тип был получен путем преобразования библиотеки типов. Носит чисто информативный характер
`LCIDConversionAttribute`	Указывает номер параметра, который отвечает за определение региональной информации
`PrimaryInteropAssemblyAttribute`	Обычно применяется утилитой `tlbimp` для указания на то, что данная сборка используется для взаимодействия со средой COM
Классы и интерфейсы	
`ExtensibleClassFactory`	Позволяет контролировать процесс создания компонентов
`ICustomFactory`	Позволяет создавать собственные фабрики классов
`ICustomAdapter`	Позволяет клиентам получать доступ к настоящим объектам, а не к тем, что предоставляет стандартный маршалер
`ICustomMarshaler`	Введен для поддержки создания пользовательских маршалеров
`GCHandle`	Позволяет получить доступ к неуправляемой памяти из управляемой среды
`HandleRef`	Позволяет управлять описателями уровня операционной системы
`Marshal`	Мощный класс поддержки взаимодействия с неуправляемой средой
`RuntimeEnvironment`	Предоставляет информацию о среде исполнения

Помимо приведенных выше, существуют и другие классы, интерфейсы, атрибуты и типы, участвующие во взаимодействии с подсистемой COM. Они здесь не упоминаются по причине их малой актуальности или узкой специализации.

Подведем итоги

В завершение раздела практического взаимодействия с COM из среды .NET подведем итоги, сравнив обе технологии. Далее (табл. 13.3) вы найдете сводное сравнение по основным характеристикам, присущим технологиям разработки программ.

Таблица 13.3. *Сводное сравнение технологий COM и .NET*

Характеристика	Подход .NET	Подход COM
Программная модель	Классово ориентированная	Интерфейсно ориентированная
Идентификация типов	Строгие полные имена (общедоступный криптографический ключ)	128-битные идентификаторы в формате GUID
Механизм передачи и отлова ошибок	Механизм блоковой обработки исключений	Значения, возвращаемые функциями в формате HRESULT
Совместимость типов	На уровне метаданных	На уровне самих данных
Предоставление информации о типах	Метаданные	Библиотеки типов
Безопасность типов	Опционально	Не поддерживается
Управление версиями	На уровне сборок	На уровне интерфейсов

Многие из данных пунктов в деталях рассматривались ранее, но все-таки некоторые из них требуют дополнительного пояснения.

Идентификация типов

Сколько бы ни ругали архитекторов COM за тяжеловесность и неповоротливость технологии, но надо отдать им должное — они создали поистине интересную технологию. Вы никогда не задумывались над тем, что COM предоставляет всем программам, компонентам и классам единое пространство имен? Когда в своей программе вы создаете COM-объект, вам не нужно знать, в какой динамической библиотеке он расположен и вообще есть ли он на вашем компьютере. Для работы с ним вам необходим только `ProgId` или `CLSID`.

Примечание

Специальные настройки в реестре позволяют незаметно для программ заставить подсистему COM использовать компонент в удаленном режиме, когда его код будет исполняться на другом компьютере.

Именно поэтому создателям COM пришлось отказаться от идентификации типов при помощи символьных имен. Обеспечить их уникальность во всем глобальном пространстве имен было бы невозможно. А идентификаторы позволяли полностью гарантировать отсутствие коллизий.

Создатели .NET поступили несколько иначе — они ввели в пространство имен еще один дополнительный уровень, в виде сборок. Именно благодаря сборкам стала возможна идентификация типов при помощи их символьных имен. Таким образом, в .NET мы имеем двухуровневую идентификацию типов:

- уровень сборки — идентификация при помощи асимметричных криптографических алгоритмов и хешей;
- уровень программного кода — строковая идентификация типов.

Помимо того что система идентификации типов .NET куда сильнее, чем в COM, она еще и гораздо удобнее в использовании за счет своей двухуровневой организации. На уровне сборок применяются механизмы сверхнадежной идентификации при помощи криптографических алгоритмов. Это позволяет на программном уровне производить идентификацию при помощи обычных строк, что, естественно, очень удобно. Первый уровень идентификации абсолютно прозрачен для пользователя: сравнением хешей сборок и работой с криптографическими ключами занимается сама среда исполнения в автоматическом режиме.

Самое интересное, что подход .NET работает ничуть не медленнее, чем механизмы работы с объектами COM. Основные мощности будут тратиться на верхнем уровне, во время идентификации сборок при помощи криптографических алгоритмов. А на втором уровне будет производиться сравнение строк, причем не прямое, а при помощи таблиц перехода, основанных на контрольных суммах имен типов. Все это позволяет существенно ускорить работу приложений .NET.

Безопасность типов

В COM проверка типов полностью ложится на компилятор. Но зачастую, особенно при использовании автоматизации, компилятор не имеет необходимой для этого информации. И весьма вероятна ситуация, когда клиент передает один тип, а COM-сервером он будет рассматриваться как совершенно другой — налицо коллизия типов данных. В .NET данная ситуация полностью исключена, поскольку на всем протяжении жизни данных с ними неразрывно связывается информация о типах (при помощи метаданных), которая постоянно проверяется JIT. И если будет обнаружено несоответствие запрашиваемого типа с данными, то будет выброшено исключение `InvalidCastException`, что естественно позволяет избежать многих глупых ошибок.

Продемонстрируем вышесказанное на примере, попробовав использовать вместо одного класса другой (листинг 13.14).

Листинг 13.14. Демонстрация невозможности подмены типа при использовании .NET

```
/*
  Листинг 13.14
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство имен общей библиотеки типов
using System;

// Просто класс
class Fake
{
};

// Основной класс нашего приложения
class App
{
  // Обратите внимание на тип параметра данной функции
  public static void X(object param)
  {
    // Вызовем метод от объекта App, которого не будет
    // у объекта, переданного в функцию
    ((App)param).Zlo();
  }

  // Присутствие данного метода необходимо для того, чтобы
  // успокоить компилятор. Хотя этот метод и не будет никогда вызван
  public void Zlo()
  {
    Console.WriteLine("Zlo Invoked");
  }

  // Точка входа в приложение
  public static void Main()
  {
    // Передадим нашей функции левый класс
```

```
    X(new Fake());
  }
}
```

В результате запуска данного приложения, среда исполнения сообщит нам об ошибке преобразования типов (рис. 13.11).

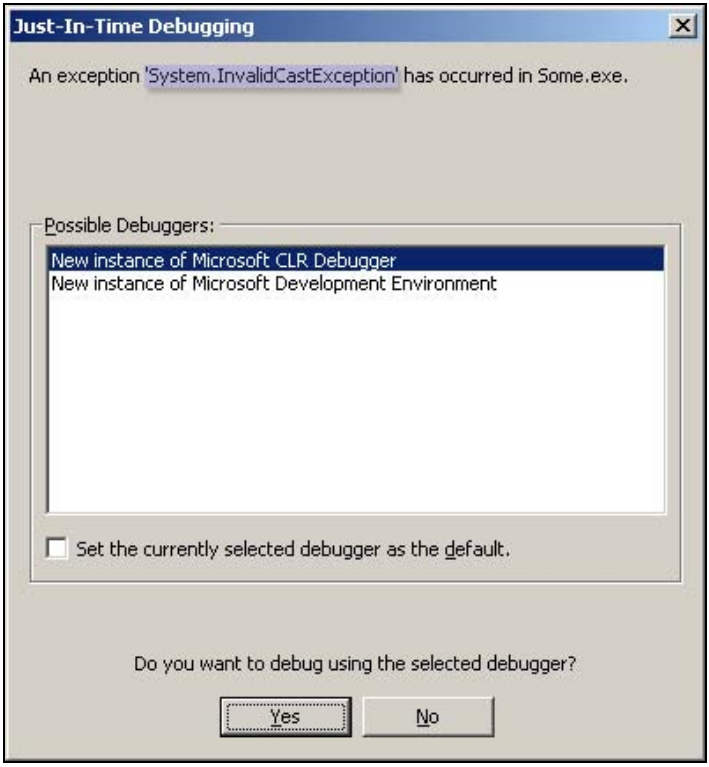


Рис. 13.11. Попытка неудачной подмены типов, ловко пресеченная средой исполнения .NET

Хотя данный пример несколько идеализирован и ошибка, намеренно допущенная в нем, очевидна, — суть от этого не меняется. В настоящих приложениях все будет точно так же, только кода будет побольше и ошибка будет менее очевидна при чтении листинга.

Таким образом, видно явное преимущество среды .NET в сфере безопасности типов и данных.

Глава 14

Тонкости взаимодействия с COM

В данной главе будут рассмотрены более глубокие вопросы взаимодействия между средами как теоретические, так и практические. В принципе, для написанных простых приложений можно ограничиться материалом, изложенным в первой части главы. Но для того чтобы создавать действительно сложные, производительные и устойчивые приложения, необходимо будет ознакомиться с остальным материалом, который по большей части носит фундаментальный характер.

Фактически, глава разбита на две части. В первой обсуждаются обертки, реализующие взаимодействие между средами, а во второй — маршализация данных между .NET и COM.

14.1. Обертки

В предыдущих главах мы уже успели убедиться в кардинальном различии между программными моделями .NET и COM. Тем не менее, взаимодействие между ними мы смогли наладить относительно легко. Самое сложное из того, с чем нам пришлось столкнуться на стороне .NET, были атрибуты. Взаимодействие же с .NET COMпонентами, как мы видели, строится так же, как и с обычными. Но за кулисами всей этой простоты скрываются достаточно неочевидные механизмы. Они представляют собой специальные обертки, обеспечивающие взаимодействие между средами. Всего существует два вида таких оберток:

- обертка для взаимодействия с COM из среды .NET (RCW, Runtime Callable Wrapper);
- обертка для взаимодействия с .NET из среды COM (CCW, Com Callable Wrapper).

Эти обертки позволяют нам легко создавать код, переходящий через границы двух сред (рис. 14.1).

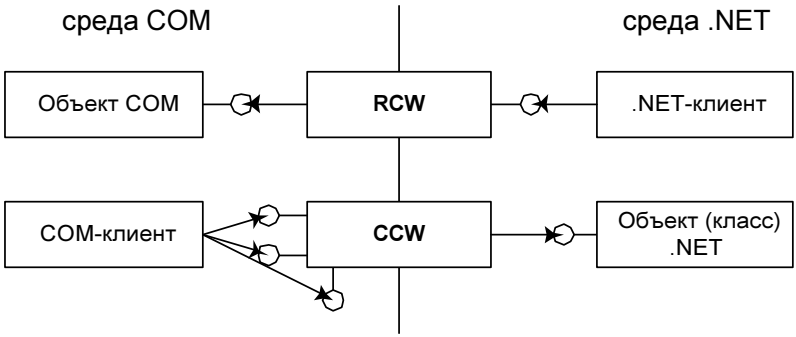


Рис. 14.1. Два вида оберток, используемых при взаимодействии между средами

Далее мы рассмотрим степень участия оберток в тех или иных операциях взаимодействия между средами, а также попытаемся ими поуправлять. С оберткой RCW все будет просто — настраивать ее работу можно при помощи атрибутов. А вот для управления CCW-обертками не предоставлено ни одного документированного сервиса. Для того чтобы изменить ее поведение, придется прибегнуть к нестандартным приемам.

RCW-обертки

При использовании COM-объектов из родной среды программисты, работающие с языками низкого уровня, вынуждены иметь дело с обязательными служебными интерфейсами (`IUnknown`, `IDispatch`, `ITypeInfo` и т. д). Но нам не приходилось с ними сталкиваться при взаимодействии с COM-объектами из среды .NET. Происходило это именно благодаря обертке RCW, которая инкапсулировала всю рутинную работу со служебными интерфейсами. Согласитесь, достаточно удобно не думать о таких низкоуровневых вещах, к тому же, уменьшается вероятность допустить ошибку. Ранее делались попытки реализовать подобную функциональность на C++ при помощи специальных интеллектуальных классов-оберток. В их задачу входило сокрытие от пользователей всей рутинной работы со служебными интерфейсами. Но из-за ограниченности возможностей языка C++ полнофункциональные обертки, подобные RCW, создать было невозможно. От программистов все равно требовалось понимание принципов работы внутренних механизмов COM-объектов.

Помимо осуществления стандартных рутинных запросов, обертка RCW отвечает за маршалинг данных между средами. Более подробно об этом будет рассказано далее.

Для того чтобы понять истинную сущность обертки RCW, проследим ее жизненный цикл. При создании объекта, через который предполагается

взаимодействие с подсистемой COM, среда исполнения предпринимает следующие шаги:

1. Создает сам управляемый объект, через который будет осуществляться взаимодействие с COMпонентом.
2. Создает RCW-обертку и присоединяет к ней объект, созданный на предыдущем шаге.
3. Создает искомый COMпонент и присоединяет его к RCW-обертке.

Все три объекта связываются таким образом, чтобы любые обращения к управляемому объекту переадресовывались RCW-обертке. Последняя, в свою очередь, использует нужный COMпонент (рис. 14.2).

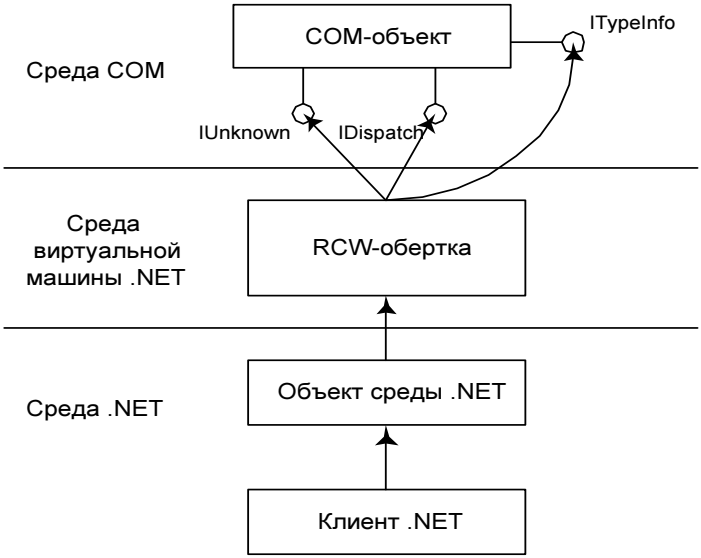


Рис. 14.2. Внутренне устройство механизма взаимодействия с COM-объектом из среды .NET

Здесь под клиентом .NET подразумевается любая функция или метод, обращающийся к объекту .NET. Дополнительный слой — "Среда виртуальной машины .NET" — представляет область виртуальной машины, которая, фактически, недоступна как из COM, так и из среды исполнения. Ранее, когда мы создавали приложения, взаимодействующие с COM, не упоминались ни среда виртуальной машины, ни тем более обертки. Мы лишь косвенно управляли ими при помощи атрибутов.

У каждого объекта среды .NET существует специальный внутренний флаг, определяющий используется ли данный объект для взаимодействия со средой COM. При любом обращении к объекту среда исполнения прове-

ряет — установлен ли данный флаг. Если да, то запрос перенаправляется к обертке RCW. Она интерпретирует запрос в соответствии с атрибутами, использовавшимися при описании типа объекта. Для того чтобы в деталях рассмотреть данный процесс, обратимся к одному из ранее приведенных примеров (листинг 14.1).

Листинг 14.1. Простой пример использования COMпонента из среды .NET

```
/*
  Листинг 14.1
  File:   Some.cs
  Author: Дубовцев Алексей
*/

// Подключим основное пространство общей библиотеки классов
using System;

// Подключим пространство имен, отвечающее за взаимодействие
// с операционной системой
using System.Runtime.InteropServices;

// Первым делом опишем интерфейс
[
  // Укажем, что интерфейс является заглушкой для связи с COM
  ComImport,
  // Укажем идентификатор интересующего нас интерфейса
  // Данный атрибут наиболее важный из используемых здесь
  Guid("D8F015C0-C278-11CE-A49E-444553540000"),
  // Укажем способ взаимодействия с интерфейсом
  InterfaceType(ComInterfaceType.InterfaceIsIDispatch),
]
interface IShell
{
  // Метод вызывает на экран диалоговое окно с предложением
  // запустить файл
  void FileRun();
  // Данный метод позволяет показать определенный апплет
  // панели управления
  void ControlPanelItem([MarshalAs(UnmanagedType.BStr)] String item);
};
```

```
// А теперь опишем класс, при помощи которого мы будем создавать
// наш COM-объект
[
  // Укажем на то, что класс используется для взаимодействия с COM
  ComImport,
  // Укажем идентификатор COM-класса, который соответствует данному
  Guid("13709620-C279-11CE-A49E-444553540000")
]
// Имя класса в данном случае не имеет никакого значения
class Shell
{
};

// Основной класс нашего приложения
class App
{
  // Точка входа в приложение
  public static void Main()
  {
    // Создадим COM-объект
    Shell ShellObj = new Shell();
    // Запросим у объекта необходимый нам для взаимодействия интерфейс
    IShell iShell = (IShell)ShellObj;

    // Это как и прежде работает, просто закомментировано
    //iShell.FileRun();

    // Вызовем на экран апплет панели управления, отвечающий за настройку
    // параметров дисплея и рабочего стола
    iShell.ControlPanelItem("desk.cpl");
  }
}
```

Код главной функции программы начинается с казалось бы безобидной конструкции:

```
Shell ShellObj = new Shell();
```

Она предписывает среде исполнения создать экземпляр объекта `Shell` и проинициализировать значение ссылки `ShellObj`. Во время создания объекта

среда исполнения, в первую очередь, проверит — был ли применен к классу модификатор `import`.

Примечание

Атрибут `ComImportAttribute` транслируется компилятором в модификатор `import`.

Если это так, то среда исполнения создаст RCW-переходник, передав ему тип объекта, который должен быть создан (в нашем случае, тип `Shell`). После этого управление передается переходнику, который уже непосредственно занимается созданием самого COM-объекта. При этом будет использоваться информация, закодированная в атрибутах, примененных к типу.

После завершения исполнения первой строки примера, все необходимые для взаимодействия с COM элементы уже созданы: управляемый объект, RCW-обертка, COM-объект. Из среды .NET будет виден только сам управляемый объект, о существовании двух других мы можем только догадываться.

Следующая строка программы куда более интересна, чем первая:

```
IShell iShell = (IShell)ShellObj;
```

С первого взгляда выглядит как простое приведение типов. Раньше за приведение типов отвечал компилятор, теперь же эта обязанность возложена на среду исполнения. Преобразование типов будет транслировано компилятором с помощью инструкции `castclass`. Для большей ясности обратимся к полному IL-коду функции `Main` (листинг 14.2).

Листинг 14.2. Полный IL-код функции `Main`

```
 .method public hidebysig static void  Main() cil managed
{
  // Указывает на то, что данный метод является точкой входа
  // в приложение
 .entrypoint
 // Code size       35 (0x23)
  // Указывает размер стека, необходимый для работы
  // данной функции
 .maxstack  2
  // Определение локальных переменных
 .locals init ( object V_0, class IShell V_1)

  // Создаем новый объект
  newobj     instance void Shell::.ctor()
```

```
  // Сохраним значение в локальной переменной V_0
  stloc.0
  // Загрузим значение локальной переменной V_0
  ldloc.0
  // Проводим преобразование интерфейсов
  castclass  IShell

  // Сохраним результат преобразования в локальной переменной
  // V_1
  stloc.1
  // Загрузим значение локальной переменной V_1
  ldloc.1
  // Загрузим ссылку на строку
  ldstr      "desk.cpl"
  // Вызовем метод через интерфейс
  callvirt   instance void IShell::ControlPanelItem(string)
  // Закончим исполнение метода
  ret
}
```

При выполнении инструкции `castclass` среда исполнения обнаружит, что объект `Shell`, от которого производится преобразование, является экспортируемым из среды COM. После этого управление будет передано RCW-обертке, которая рассматривает запрос на преобразование типов как запрос интерфейса. Она запросит атрибут `Guid` интерфейса. По нему будет вызван метод `QueryInterface`, через который запрашивается необходимый интерфейс. В случае если объект поддерживает интерфейс, то он вернет ссылку на него. Обертка RCW, в свою очередь, создаст для объекта специальный управляемый интерфейс, который будет перенаправлять вызовы в RCW.

Далее следует третий шаг — обращение к методу `ControlPanelItem`:

```
iShell.ControlPanelItem("desk.cpl");
```

На уровне IL-кода данный вызов ничем не отличается от обращения к обычному управляемому интерфейсу. Все самое интересное происходит в недрах среды исполнения, во время выполнения программы. Интерфейс, через который будет происходить вызов, создан таким образом, чтобы переадресовывать вызовы к RCW. А она уже, в свою очередь, произведет маршалинг данных и обратится к настоящему COM-интерфейсу.

В пределах одного .NET-процесса для каждого импортируемого объекта создается персональная обертка RCW, которая контролирует все обращения к нему (рис. 14.3).

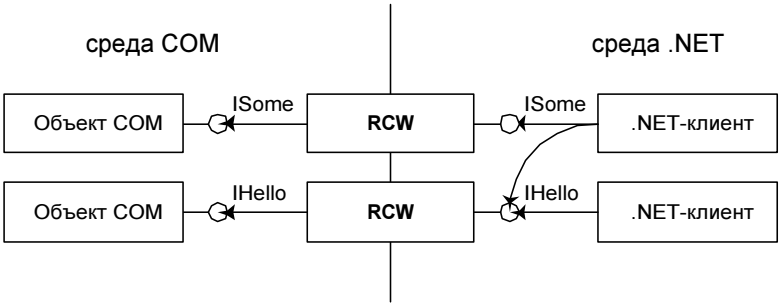


Рис. 14.3. Устройство внутреннего механизма обращения к COM объекту через обертку RCW

Доступ к COM-объекту осуществляется только через его RCW-обертку, независимо от того, какой .NET-клиент будет его использовать. Причем обойти данную обертку (документированными способами) совершенно невозможно, она будет неусыпно контролировать все ваши действия. С одной стороны, такой тотальный контроль некоторым может показаться избыточным. Но это позволяет предупредить глупые ошибки, которые можно совершить по неосторожности, а также избавиться от множества низкоуровневых деталей.

Кстати о деталях, как уже было сказано, обертка RCW самостоятельно взаимодействует со всеми служебными интерфейсами. Далее (табл. 14.1) перечислены все интерфейсы, с которыми взаимодействует RCW, а также их краткое описание.

Таблица 14.1. *Список интерфейсов, с которыми взаимодействует обертка RCW*

Интерфейс	Описание
`IDispatch`	Данный интерфейс используется для позднего связывания, через механизмы отражения (reflection). А также в случаях, когда явно указано, что взаимодействие должно производиться с его помощью
`IErrorInfo`	При возникновении ошибки в среде COM данный интерфейс используется для получения более точной и детальной информации, которая впоследствии перейдет в объект классического .NET-исключения, выброшенного оберткой RCW
`IProvideClassInfo`	Данный интерфейс используется оберткой RCW для получения более точной и полной информации о типах, если не хватает информации, указанной в метаданных

Таблица 14.1 *(окончание)*

Интерфейс	Описание
`IUnknown`	Данный интерфейс используется в трех случаях: • для идентификации COM-объектов — среда исполнения сравнивает указатели на интерфейсы `IUnknown`, предоставляемые COMпонентами, для того чтобы определить принадлежность интерфейса к компоненту; • для запроса интерфейсов посредством приведения типов — при запросе интерфейса обертка RCW обращается к методу `QueryInterface`, который, в случае успеха, вернет указатель на необходимый интерфейс; • для управления временем жизни COM-объекта — обертка RCW прозрачно для пользователя подсчитывает ссылки на COM-объект при помощи методов `AddRef` и `Release`, тем самым управляя временем его жизни
`IConnectionPoint` и `IConnectionPointContainer`	Используется оберткой RCW для регистрации событий COM, которые впоследствии будут перенаправляться к соответствующим управляемым событиям и делегатам
`IDispatchEx`	Если объект поддерживает данный интерфейс, то обертка RCW автоматически создает управляемый интерфейс `IExpando`, который может быть использован для доступа к его методам. Данный интерфейс является расширенной версией `IDispatch` и, в отличие от него, умеет перечислять, добавлять, удалять и обращаться к членам без учета регистра букв. Данный интерфейс незаменим при работе со скриптовыми движками, например, Windows Script Host
`IEnumVARIANT`	Данный интерфейс используется RCW для предоставления доступа к COM-коллекциям через обычные коллекции .NET

Может показаться, что интерфейсов, обращения к которым инкапсулирует обертка RCW, не так уж и много. Но поверьте, обертка проделывает за нас громадную работу, освобождая нас от написания многих километров рутинного кода.

CCW-обертки

При использовании COMпонента .NET работа с ним из среды COM никоим образом не отличается от взаимодействия с обычными компонентами, несмотря на то, что это всего лишь умелая эмуляция. Точно так же, как

и для обычных компонентов, мы имеем возможность обращаться к специальным служебным интерфейсам, невзирая на то, что сам управляемый класс напрямую их не поддерживает. Это возможно благодаря обертке CCW. Она эмулирует несуществующие в управляемом коде служебные интерфейсы среды COM, а также отвечает за обработку их вызовов (рис. 14.4).

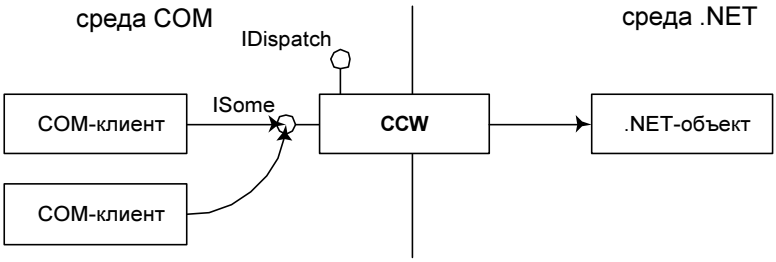


Рис. 14.4. Внутреннее устройство механизма взаимодействия с .NET COMпонентом

Как можно видеть, клиенты COM на самом деле, прежде всего, взаимодействуют не с объектом из среды .NET, а с оберткой CCW, о которой раньше мы даже не подозревали. Если сравнивать обертки RCW и CCW, то получается, что первая скрывает интерфейсы от пользователя, а вторая эмулирует. Забавно, не правда ли? Но, учитывая, что задачи у них диаметрально противоположные, все становится на свои места.

Как и в случае с RCW-оберткой, среда исполнения для каждого .NET-объекта, запрошенного из среды COM, создает персональную CCW-обертку (рис. 14.5).

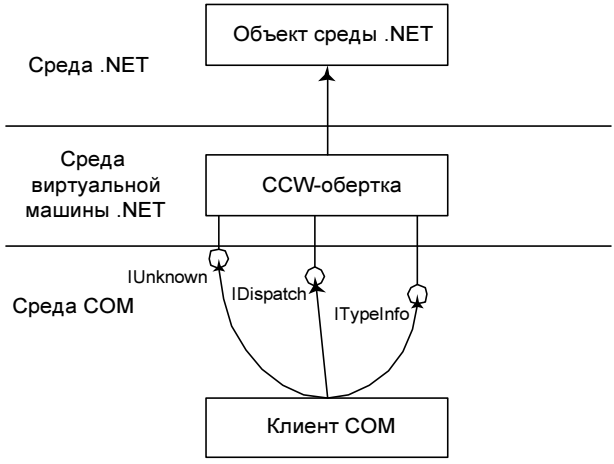


Рис. 14.5. Эмуляция интерфейсов CCW-оберткой

Набор интерфейсов, которые эмулирует обертка CCW для среды COM, будет очень схож с набором интерфейсов, скрываемых RCW. Но поскольку задачи у них разные, описывать их придется отдельно (табл. 14.2).

Таблица 14.2. *Набор интерфейсов, эмулируемых оберткой RCW*

Интерфейс	Описание
`IDispatch`	Предоставляет механизмы автоматизации для позднего связывания, работающие на уровне .NET через сервисы отражения
`IErrorInfo`	Позволяет получить полную информацию об ошибке, указанной в объекте выброшенного управляемого исключения
`IProvideClassInfo`	Предоставляет более полную информацию о типах, которую сама среда при помощи технологии отражения берет из метаданных
`ISupportErrorInfo`	Сигнализирует COM-клиентам о том, что данный объект поддерживает интерфейс `IErrorInfo`, позволяющий получить описание произошедшей ошибки
`ITypeInfo`	Предоставляет информацию о типах для указанного .NET-объекта. Данный сервис внутри среды .NET работает через технологию отражения
`IUnknown`	Предоставляет сервисы по управлению временем жизни управляемого объекта, а также запроса других, поддерживаемых им интерфейсов
`_ИмяКласса`	Предоставляет доступ к членам управляемого класса либо через механизмы автоматизации (отражения), либо напрямую
`IConnectionPoint` и `IConnectionPointContainer`	Предоставляют доступ к делегатам и событиям .NET-объектов, обычным для COM способом
`IDispatchEx`	Будет предоставлен оберткой CCW только в случае, если управляемый объект реализует интерфейс `IExpando`
`IEnumVARIANT`	Предоставляет доступ к .NET-коллекциям, через COM-коллекции

В результате, становится ясно, что все интерфейсы, предоставляемые .NET-компонентами, по сути дела, являются всего лишь переходниками, перенаправляющими вызовы к служебным сервисам среды исполнения.

Способы управления обертками

Прямых сервисов управления обертками не существует, более того — сами обертки вообще не представлены в программной модели общей библиотеки типов .NET. Есть только косвенные механизмы, позволяющие воздействовать на них. Методы, о которых здесь будет идти речь, являются "полулегальными", поскольку, с одной стороны, о них все-таки упоминает Microsoft, а с другой стороны, они совершенно не документированы.

Управление RCW-обертками

Во время работы обертка RCW использует следующие четыре источника информации.

1. Атрибуты.
2. Информация о типах, расположенная в метаданных.
3. Информация о типах, расположенная в библиотеках типов COM.
4. Информация о типах, предоставляемая системой автоматизации через служебные интерфейсы.

Здесь, пожалуй, стоит оговориться, — по большому счету, первые два пункта идентичны, поскольку атрибуты являются частью информации о типах. Но в данном случае, мы будем рассматривать их отдельно. Ниже (рис. 14.6) представлены все источники информации, которыми пользуется RCW, а также их взаимосвязи.

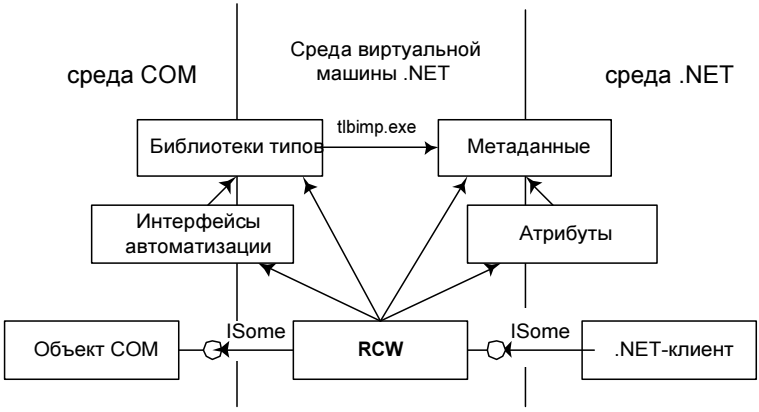


Рис. 14.6. Источники информации, используемые оберткой RCW

Стоит особо обратить внимание на связь интерфейсов автоматизации с библиотеками типов, а атрибутов с метаданными. Связь атрибутов с метаданными очевидна, поскольку они составляют общую систему информации

о типах. С интерфейсами автоматизации дело обстоит гораздо интереснее. Чаще всего интерфейсы автоматизации предоставляют информацию о типах, загружая ее из библиотек типов. Такой подход куда более прост в реализации, чем передача необходимых данных в ручном режиме. Последний применяют в редких случаях: например, в скриптовых средах, где библиотеки типов попросту отсутствуют.

Таким образом, для того чтобы повлиять на работу обертки RCW, мы можем использовать три подхода: применить необходимые атрибуты, внести изменения в метаданные или изменить содержание библиотеки типов.

В принципе, конечно, можно еще прибегнуть к перехвату интерфейсов автоматизации, но это слишком сложная тема, выходящая за рамки данной книги.

Применение атрибутов

Конечно же, самым простым вариантом является применение соответствующих атрибутов. В подавляющем большинстве случаев, их функциональности хватает за глаза.

На протяжении нескольких последних глав мы различными способами применяли атрибуты, так что повторное обсуждение способов их использования будет избыточным.

Изменение библиотек типов

Внося изменения в библиотеку типов компонента, можно преследовать две цели. Во-первых, изменить поведение интерфейсов автоматизации. Во-вторых, скорректировать сборку, которая впоследствии будет создана из данной библиотеки типов при помощи утилиты `tlbimp.exe`. Но какова бы ни была конечная цель, для начала все-таки необходимо изменить саму библиотеку типов.

Прежде всего нам понадобится ее исходный код. Если он не поставляется вместе с компонентом или в SDK, тогда можно получить его, декомпилируя библиотеку при помощи утилиты COM/Ole Object Viewer (oleview.exe).

После того как исходный код библиотеки типов окажется в нашем распоряжении, необходимо решить, какие параметры необходимо изменять. Можно ограничиться только изменением типов, которые прямым образом влияют на поведение системы маршализации данных между средами. Или изменить названия методов, интерфейсов, классов или других программных элементов. При этом необходимо быть предельно осторожным — в дальнейшем такую библиотеку можно будет гарантированно использовать только для создания сборки (`tlbimp.exe`). Оставлять такую библиотеку для работы с приложениями крайне не рекомендуется, поскольку представленная в ней информация о типах не будет соответствовать внутреннему устройству компонента. Соответственно, использование такой библиотеки сервисами автоматизации, вероятнее всего, приведет к неверной работе приложения.

Кроме того, существует возможность введения в библиотеку типов, специальных пользовательских атрибутов, незаметных для стандартной системы автоматизации, но которые позволяют влиять на код сборки генерируемой утилитой `tlbimp.exe`.

Язык ODL (Object Description Language, Язык описания объектов), используемый для написания библиотек типов, поддерживает введение пользовательских атрибутов. Конечно, данные атрибуты по степени информативности сильно уступают атрибутам .NET, но, тем не менее, позволяют расширить пользовательскую информацию о типах.

Атрибуты в ODL задаются при помощи ключевого слова `custom`

```
[
        ...
        custom(
                // Идентификатор атрибута
                XXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX,
                // Данные данного атрибута
                "hello")
        ...
]
```

Первым параметром должен быть 128-битный `GUID`-идентификатор, а вторым — произвольное значение одного из COM-типов `Variant`. Наиболее интересные из пользовательских атрибутов, используемых утилитой `tlbimp.exe`, описаны далее (табл. 14.3).

Таблица 14.3. *Пользовательские атрибуты, используемые утилитой tlbimp.exe*

Идентификатор атрибута	Описание
90883F05-3D28-11D2-8F17-00A0C9A6186D	Позволяет задать строгое имя сборки компонента, интерфейсы которого представлены в данной библиотеке типов
0F21F359-AB84-41E8-9A78-36D110E6D2F9	Позволяет определить имя программного элемента при экспорте в среду .NET

Первый атрибут не представляет для нас особого интереса — в основном он используется самой утилитой `tlbexp`, при создании библиотек типов. Пример использования данного атрибута в заголовке библиотеки типов, представляющей общую библиотеку классов среды .NET:

```
[
  uuid(BED7F4EA-1A96-11D2-8F08-00A0C9A6186D),
  version(1.10),
```

```
  helpstring("Common Language Runtime Library"),
  custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, mscorlib,↳
Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089)


]
library mscorlib
{
       ...
```

Второй атрибут гораздо интереснее и, главное, полезнее, чем первый, — он позволяет определить имя программного элемента при экспорте в среду .NET. К примеру, при помощи данного атрибута можно изменить имя интерфейса в среде .NET, не меняя при этом его имени внутри самой библиотеки типов:

```
[
  object,
  uuid(79E75D05-2AFA-4D39-96F5-77EFF6003459),
  dual,
  helpstring("IHello Interface"),
  pointer_default(unique),
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "INewName"),
]
interface IHello : IDispatch {
```

Если такую библиотеку преобразовать в сборку при помощи утилиты `tlbimp`, то данный интерфейс будет именоваться в управляемом коде `INewName`, а не `IHello`. При этом в библиотеке типов он сохранит имя `IHello`, что позволит избежать коллизий при обращении из компонента.

Основное достоинство данного атрибута состоит в возможности определения с его помощью перегруженных функций.

Примечание

Перегруженными называются функции с одинаковыми именами, но разными прототипами. К примеру, функции `SayHello()` и `SayHello(String str)` являются перегруженными.

Как известно, COM в силу ограничений со стороны системы предоставления информации о типах не поддерживает перегрузку функций. В среде .NET такого ограничения нет. Чтобы использовать из COM функции, перегруженные в среде .NET, придется воспользоваться вышеупомянутым атрибутом. Для этого мы просто создадим две функции с различными именами и пометим их атрибутом таким образом, чтобы они экспортировались

в среду .NET под одним именем. Пример, демонстрирующий экспортирование перегруженных функций в среду COM:

```
...

[
        id(1),
        helpstring("method SayHello"),
        // Объявим для этого метода имя SayHello в среде .NET
        custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "SayHello"),
]
HRESULT SayHello( [in] BSTR message);

[
        id(2),
        // И для этого метода мы тоже объявим имя SayHello в среде .NET,
        // хотя реальное имя метода расширено постфиксом _2
        custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9, "SayHello"),
]
HRESULT SayHello_2( [in] BSTR messag, [in] int SomeData);

                ...
```

Кстати говоря, при конвертировании сборок, имеющих перегруженные функции, в библиотеки типов COM утилита `tlbexp` поступает точно таким же образом. Она модифицирует имя метода, добавляя к нему число, как и было сделано в примере.

Внесение изменения в метаданные

Для внесения изменений в метаданные существует два доступных пути. Во-первых, можно просто декомпилировать сборку, внести изменения в IL-код, а затем собрать ее заново. Во-вторых, можно загрузить сборку в память, затем при помощи технологии отражения внести в нее соответствующие изменения. Безусловно, первый способ гораздо проще, чем первый.

Управление CCW-обертками

Способов управления CCW-обертками не так уж и много, так как они используют только информацию о типах, закодированную в метаданных. Соответственно, можно изменять типы и применять атрибуты. Кроме того, можно реализовывать некоторые стандартные интерфейсы самостоятельно. Но здесь данные технологии, в силу их узкой специализации и сложности, рассматриваться не будут.

14.2. Несколько дополнительных технических моментов

В данном разделе будут рассмотрены несколько технических моментов, которые по различным причинам не были описаны в других разделах.

Обработка ошибок

.NET и COM имеют различные технологии обработки ошибок. В .NET она строится на исключениях и объектах, их описывающих, а в COM — на кодах ошибок. При взаимодействии между средами обе обертки, RCW и CCW, преобразуют исключения в коды ошибок и наоборот. Ниже (табл. 14.4) приведены коды ошибок и соответствующие типы исключений.

Таблица 14.4. *Соответствие кодов ошибок и типов исключений в среде .NET*

Идентификатор COM-ошибки	Исключение .NET
COR_E_APPDOMAINUNLOADED	AppDomainUnloadedException
COR_E_APPLICATION	ApplicationException
COR_E_ARGUMENT или E_INVALIDARG	ArgumentException
COR_E_ARGUMENTOUTOFRANGE	ArgumentOutOfRangeException
COR_E_ARITHMETIC или ERROR_ARITHMETIC_OVERFLOW	ArithmeticException
COR_E_ARRAYTYPEMISMATCH	ArrayTypeMismatchException
COR_E_BADIMAGEFORMAT или ERROR_BAD_FORMAT	BadImageFormatException
COR_E_COMEMULATE_ERROR	COMEmulateException
COR_E_CONTEXTMARSHAL	ContextMarshalException
COR_E_CORE	CoreException
NTE_FAIL	CryptographicException
COR_E_DIRECTORYNOTFOUND или ERROR_PATH_NOT_FOUND	DirectoryNotFoundException
COR_E_DIVIDEBYZERO	DivideByZeroException
COR_E_DUPLICATEWAITOBJECT	DuplicateWaitObjectException
COR_E_ENDOFSTREAM	EndOfStreamException
COR_E_TYPELOAD	EntryPointNotFoundException
COR_E_EXCEPTION	Exception

Таблица 14.4 *(продолжение)*

Идентификатор COM-ошибки	Исключение .NET
COR_E_EXECUTIONENGINE	ExecutionEngineException
COR_E_FIELDACCESS	FieldAccessException
COR_E_FILENOTFOUND или ERROR_FILE_NOT_FOUND	FileNotFoundException
COR_E_FORMAT	FormatException
COR_E_INDEXOUTOFRANGE	IndexOutOfRangeException
COR_E_INVALIDCAST или E_NOINTERFACE	InvalidCastException
COR_E_INVALIDCOMOBJECT	InvalidComObjectException
COR_E_INVALIDFILTERCRITERIA	InvalidFilterCriteriaException
COR_E_INVALIDOLEVARIANTTYPE	InvalidOleVariantTypeException
COR_E_INVALIDOPERATION	InvalidOperationException
COR_E_IO	IOException
COR_E_MEMBERACCESS	AccessException
COR_E_METHODACCESS	MethodAccessException
COR_E_MISSINGFIELD	MissingFieldException
COR_E_MISSINGMANIFESTRESOURCE	MissingManifestResourceException
COR_E_MISSINGMEMBER	MissingMemberException
COR_E_MISSINGMETHOD	MissingMethodException
COR_E_MULTICASTNOTSUPPORTED	MulticastNotSupportedException
COR_E_NOTFINITENUMBER	NotFiniteNumberException
E_NOTIMPL	NotImplementedException
COR_E_NOTSUPPORTED	NotSupportedException
COR_E_NULLREFERENCE или E_POINTER	NullReferenceException
COR_E_OUTOFMEMORY или E_OUTOFMEMORY	OutOfMemoryException
COR_E_OVERFLOW	OverflowException
COR_E_PATHTOOLONG или ERROR_FILENAME_EXCED_RANGE	PathTooLongException
COR_E_RANK	RankException
COR_E_REFLECTIONTYPELOAD	ReflectionTypeLoadException

Таблица 14.4 *(окончание)*

Идентификатор COM-ошибки	Исключение .NET
COR_E_REMOTING	RemotingException
COR_E_SAFEARRAYTYPEMISMATCH	SafeArrayTypeMismatchException
COR_E_SECURITY	SecurityException
COR_E_SERIALIZATION	SerializationException
COR_E_STACKOVERFLOW или ERROR_STACK_OVERFLOW	StackOverflowException
COR_E_SYNCHRONIZATIONLOCK	SynchronizationLockException
COR_E_SYSTEM	SystemException
COR_E_TARGET	TargetException
COR_E_TARGETINVOCATION	TargetInvocationException
COR_E_TARGETPARAMCOUNT	TargetParameterCountException
COR_E_THREADABORTED	ThreadAbortException
COR_E_THREADINTERRUPTED	ThreadInterruptedException
COR_E_THREADSTATE	ThreadStateException
COR_E_THREADSTOP	ThreadStopException
COR_E_TYPELOAD	TypeLoadException
COR_E_TYPEINITIALIZATION	TypeInitializationException
COR_E_VERIFICATION	VerificationException
COR_E_WEAKREFERENCE	WeakReferenceException
COR_E_VTABLECALLSNOTSUPPORTED	VTableCallsNotSupportedException
Все остальные идентификаторы (HRESULT)	COMException

Потоковые модели

Те люди, которые хотя бы раз использовали или разрабатывали компоненты, предназначенные для работы в многопоточном режиме, наверняка знакомы с понятием потоковых моделей. Существует два типа компонентов: поддерживающие и не поддерживающие многопоточность. Один тип может работать только в том потоке, в котором он был создан, а ко второму типу компонентов можно обращаться из любого потока приложения. Внутренние механизмы подсистемы COM требуют, чтобы в самом начале работы приложения была выбрана потоковая модель, которая будет использоваться для работы с компонентами. Делается это при помощи функции `CoIntitializeEx`. Среда

исполнения, начиная исполнение управляемого приложения, также обращается к функции CoInitializeEx. При этом она руководствуется значением свойства Thread.ApartmentState объекта, представляющего текущий поток. При каждой установке данного свойства среда исполнения обращается к функции CoInitializeEx, передавая ей соответствующее значение. Надо отметить, что у данной функции существует одна особенность — она может установить значение потоковой модели приложения только при первом своем вызове, повторные обращения к ней никакой роли не играют. Соответственно, потоковую модель можно выбрать только один раз и при этом навсегда.

Изменить значение свойства, а соответственно и потоковой модели приложения, можно двумя способами:

- явным образом, установив значение поля Thread.ApartmentState равным одному из членов перечисления ApartmentState;
- применив один из атрибутов STAThread или MTAThread к функции, являющейся точкой входа в приложение.

Соответствие потоковых моделей, атрибутов и значений членов перечисления ApartmentState приведены далее (табл. 14.5).

Таблица 14.5. *Соответствие потоковых моделей и значений перечисления* `ApartmentState`

Атрибут .NET	Вызов функции подсистемы COM	Значение свойства Thread.ApartmentState
MTAThread	CoInitializeEx(NULL, COINIT_MULTITHREADED)	ApartmentState.MTA
STAThread	CoIntializeEx(NULL, COINIT_APARTMENTTHREADED)	ApartmentState.STA
Не установлен	CoInitializeEx(NULL, COINIT_MULTITHREADED)	ApartmentState.Unknow

Применение атрибута MTAThread, устанавливающего потоковую модель, будет выглядеть так (листинг 14.3).

Листинг 14.3. Пример установки многопоточной модели для управляемого приложения

```
/*
  Листинг 14.3
  File:   Some.cs
  Author: Дубовцев Алексей
*/
```

```
// Подключим основное пространство имен общей библиотеки классов
using System;

// Основной класс приложения
class App
{
  // Устанавливаем многопоточную модель для нашего приложения
  [MTAThread]
  // Точка входа в приложение
  public static void Main()
  {
    Console.WriteLine("Hello World");
  }
}
```

В результате применения данного атрибута, среда исполнения переведет подсистему COM в многопоточный режим еще до непосредственного обращения к функции `Main`.

14.3. Маршализация данных

В данном разделе будет обсуждаться одна из самых интересных и сложных тем взаимодействия двух сред — маршализация данных. Первым делом, мы рассмотрим, что вообще представляет собой маршализация данных и для чего она нужна. В связи с этим необходимо будет рассмотреть вопросы, связанные с управлением памятью в среде .NET.

В обычном управляемом приложении представлено две области памяти: управляемая и неуправляемая. Первой занимается менеджер памяти .NET, который строго контролирует все операции с управляемой памятью. Данную область памяти может использовать только управляемый код, да и то не напрямую, а косвенно, через стандартные операции выделения объектов и массивов. При этом даже все косвенные операции будут четко контролироваться — менеджер памяти .NET будет подсчитывать ссылки на все используемые объекты, следить за перемещением объектов в памяти, проводить сборку мусора (удалять неиспользуемые объекты).

Неуправляемая область памяти используется неуправляемым кодом, а также самой средой исполнения в служебных целях. В этой области памяти, в отличие от управляемой, можно проводить любые операции, разрешенные менеджером памяти операционной системы. Правда, для этого приложению

необходимо получить специальную привилегию, которая выдается администратором системы исходя из соображений безопасности.

Далее (табл. 14.6) приводится краткое сравнение способов работы с памятью в неуправляемой и управляемой средах.

Таблица 14.6. *Сравнение подходов к управлению памятью в управляемой и неуправляемой средах*

Операция	Неуправляемый код (Windows API)	Управляемая среда (IL-код, а также сервисы общей библиотеки классов)
Выделение памяти	Ручное (`HeapAlloc`)	Автоматическое (ручное не поддерживается)
Освобождение памяти	Ручное (`HeapFree`)	Автоматическое (ручное не поддерживается)
Доступ к произвольным участкам памяти	При помощи указателей и специальных сервисов	Не поддерживается
Информация о карте памяти приложения	При помощи специальных сервисов (`VirtualQuery` и т. п.)	Не поддерживается

Описание, приведенное в таблице, относится к управляемому коду с минимальным уровнем привилегий, не допускающим использование дополнительных операций. Но все же видно, что среда управления .NET ревностно оберегает управляемую область памяти, не допуская туда никого, кроме собственного менеджера. Разработчики .NET решили, что пора отстранить обычного программиста от такой важной задачи, как управление памятью. Такой подход вполне оправдан, поскольку большинство ошибок в неуправляемых программах появляется именно из-за неправильной работы с памятью.

Архитектура памяти в управляемых приложениях, в графическом виде (рис. 14.7), представлена далее.

Для доступа к каждой из областей памяти используется свой персональный менеджер. Но если взглянуть на ситуацию еще более пристально, оказывается, что все происходит несколько иначе. Фактически, среда исполнения является обычным приложением, работающим под управлением операционной системы. Следовательно, для взаимодействия с памятью она просто обязана использовать сервисы, предоставляемые менеджером памяти операционной системы. Другого пути нет. Работать с памятью в обход операционной системы в принципе, конечно, возможно, но это бессмысленно, и среда исполнения, естественно, не делает этого.

Таким образом, реальная картина будет выглядеть следующим образом (рис. 14.8).

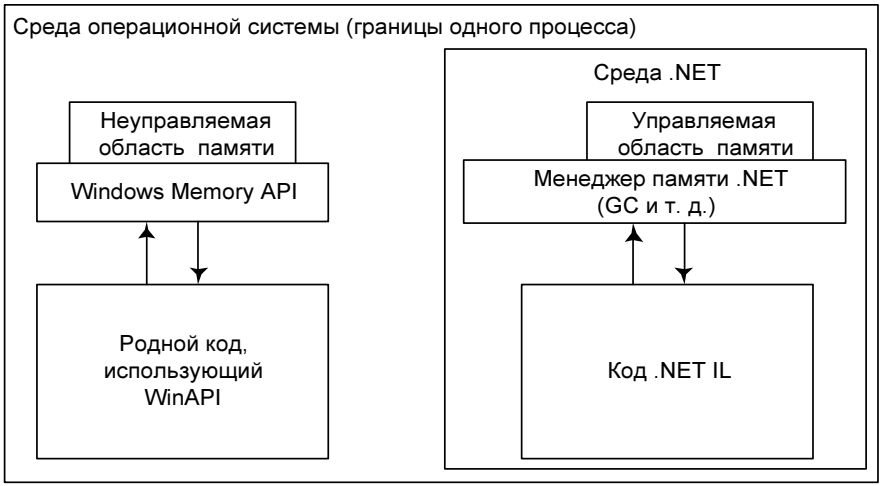


Рис. 14.7. Архитектура памяти в управляемых приложениях

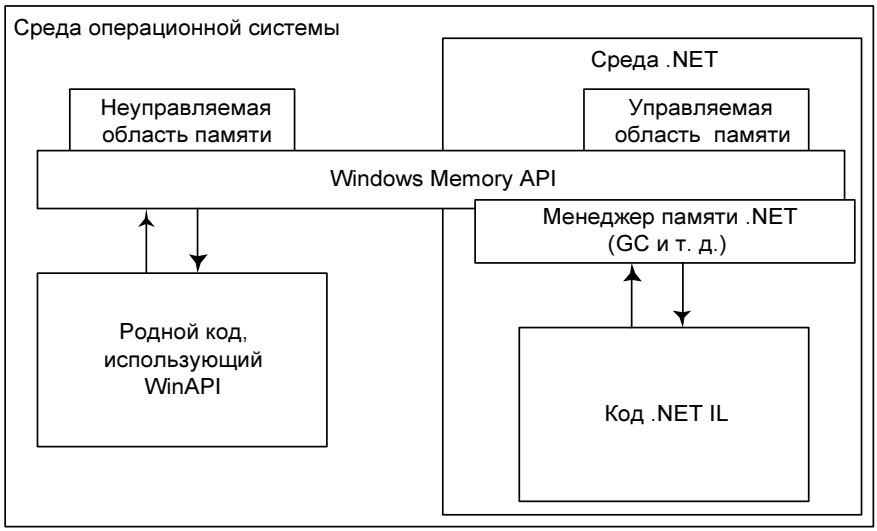


Рис. 14.8. Реальное устройство системы управления памятью внутри среды исполнения

Из этого плавно вытекает, что с помощью Windows Memory Management API (Сервисы управления памятью Windows) можно получить доступ к области памяти .NET. Отметим, что это мало что даст, поскольку структура управляемой области в полной мере не документирована, а просто так копаться в груде данных не имеет смысла. Помимо этого, существует еще один неприятный момент — никто не даст гарантии, что данные останутся на своем месте на всем протяжении работы приложения. Напомню, что во время

операций сборки мусора менеджер памяти среды .NET имеет обыкновение передвигать некоторые участки памяти. Отследить данный процесс извне практически невозможно.

Итак, мы совершенно незаметно подобрались к процессу маршализации данных.

Примечание

Маршалинг от анлийского marshaling — сортировка, расположение в определенном порядке.

Процесс маршализации данных заключается в переносе данных из одной области памяти в другую. К этому моменту нашего повествования необходимость этого должна быть предельно ясна. Схема маршализации данных при использовании классических динамических библиотек представлена далее (рис. 14.9).

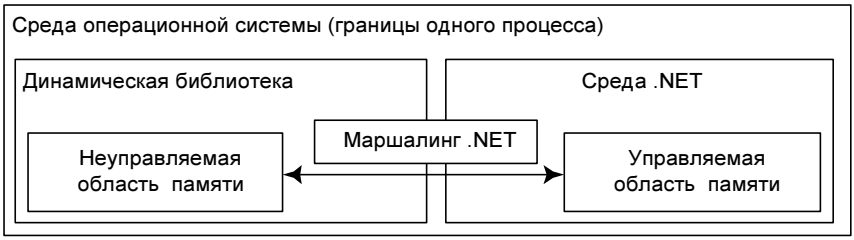


Рис. 14.9. Схема маршализации данных при взаимодействии с классическими динамическими библиотеками

В отличие от динамических библиотек, взаимодействие с COM-компонентами выглядит несколько сложнее. Подсистема COM дополнительно вводит собственную систему маршализации. Она подключается при вызове компонентов, расположенных в EXE-серверах, а также в некоторых случаях для компонентов, вызываемых из других потоков. Кроме этого, COM использует свои границы разделения данных, называемые апартаментами (COM Apartments), перенос данных между которыми осуществляется обязательно при участии сервисов маршализации. В итоге, схема маршализации данных при работе с COM из управляемого кода будет выглядеть следующим образом (рис. 14.10).

Безусловно, необходимо упомянуть о возможности избежать необходимости использования COM-маршалинга. Для этого можно применить модель свободных потоков, присоединив атрибут `MTAThread` к функции точке входа в управляемое приложение.

Маршалинг COM и маршалинг .NET — различные по своей природе механизмы. Первый необходим лишь для переноса данных в самой среде COM

(между ее апартаментами) и сводится к простому копированию данных. В подавляющем большинстве случаев маршализация COM происходит автоматически — либо при помощи заглушки, создаваемой компилятором прозрачно для программиста, либо при помощи сервисов стандартного маршалера. Последний получает необходимую информацию из библиотек типов компонента, но только в том случае, если взаимодействие происходит через механизмы автоматизации.

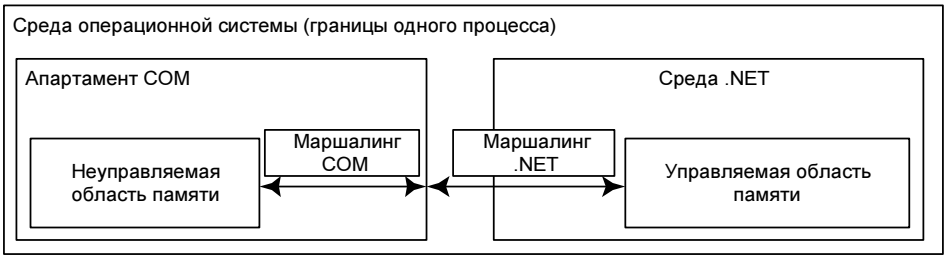


Рис. 14.10. Общая схема маршализации данных при взаимодействии с COM из управляемого кода

Маршалинг .NET несколько сложнее, поскольку в его задачу входит не только копирование данных между различными средами. Также проводится преобразование типов, внутреннее устройство которых различается в этих средах. К примеру, для представления строк в среде .NET используется класс `System.String`, который инкапсулирует строку в формате UNICODE. В то же время, в среде COM стандартом для представления строк является формат BSTR, а в операционной системе вообще могут использоваться ANSI-строки. Таким образом, при передаче данных из одной среды в другую маршалеру .NET придется не только скопировать данные, но еще и преобразовать их.

Итак, механизмы маршалинга используются при обращении из одной среды в другую. При этом данные сервисы отвечают за преобразование параметров функций, а также их копирование из одной среды в другую. На втором пункте мы остановимся более подробно.

У каждого приложения .NET существуют два стека, в которые помещаются параметры при обращении к функциям, первый стек — для вызова управляемых функций, а второй — для обращения к неуправляемым. Из этого следует, что при вызове функций из другой среды необходимо перенести параметры из одного стека в другой (рис. 14.11).

Все казалось бы предельно просто, если бы не одно *но*. Все типы .NET подразделяются на два вида: передаваемые по ссылке и по значению. С данными, передаваемыми по значению, все действительно просто: они полностью хранятся в управляемом стеке приложения, а их перенос осуществляется копированием из стека в стек. А вот с данными, передаваемыми по ссылке,

дело обстоит гораздо сложнее. В стеке хранятся не сами данные, а лишь ссылки на них (рис. 14.12)

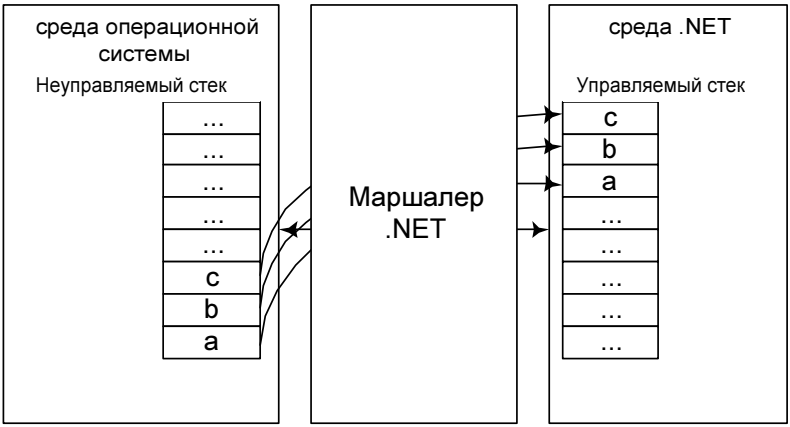


Рис. 14.11. Маршалинг параметров (данных) между управляемым и неуправляемым стеками

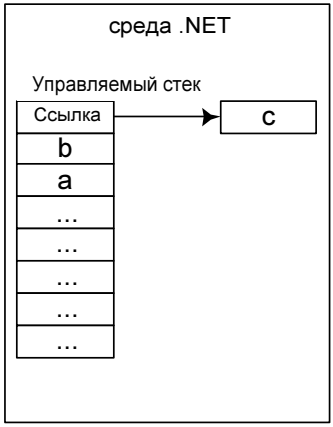


Рис. 14.12. Данные, передаваемые по ссылке

Причем эти ссылки указывают в управляемую область памяти. Таким образом, маршалер не может просто так перекинуть ссылку в другой стек. Придется еще скопировать данные из управляемой области в неуправляемую или обратно (рис. 14.13).

На самом деле, из соображений оптимизации по скорости, данные копируются далеко не всегда. При большом размере передаваемых данных маршалер .NET блокирует данные в управляемой области и передает указатель прямо на них. Помимо этого, в маршалер встроено еще несколько оптими-

зирующих трюков, позволяющих отказаться от копирования данных или любых других действий, замедляющих работу приложений.

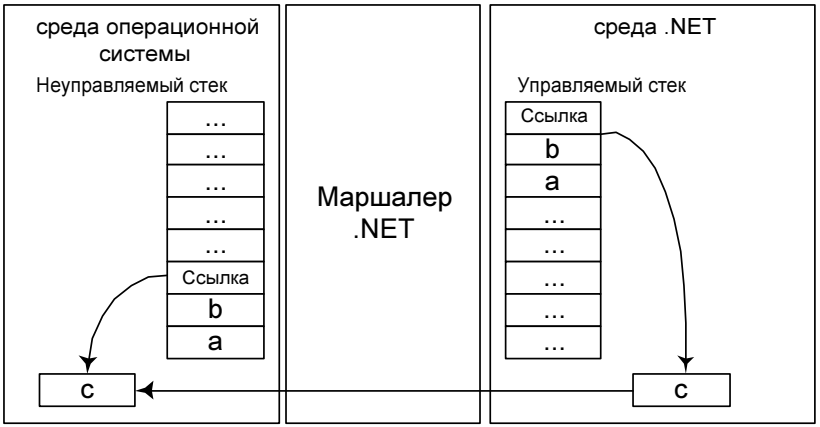


Рис. 14.13. Передача параметров по ссылке

Типы и маршалер

Некоторые типы имеют различное представление в .NET и COM, а также в среде операционной системы.

Примечание

Среда COM изначально разрабатывалась как независимая от аппаратной и программной платформы, на которых она исполняется. Впоследствии это позволило осуществить ее реализацию на различных платформах начиная от Unix и кончая Mac. Соответственно, многие из типов, используемых в среде COM, кардинально отличаются от типов конкретной операционной системы.

Следовательно, при их переносе из одной среды в другую необходимо проводить соответствующие преобразования. Однако, к всеобщей радости, все же существует группа достаточно часто используемых типов, не требующих преобразования:

- `System.Byte`
- `System.SByte`
- `System.Int16`
- `System.UInt16`
- `System.Int32`
- `System.UInt32`

- `System.Int64`
- `System.IntPtr`
- `System.UIntPtr`

Поскольку при передаче данных типов преобразования не нужны, рекомендуется использовать именно их. Это позволит немного увеличить производительность ваших приложений.

Далее (табл. 14.7) представлена группа типов, при передаче которых требуется проводить преобразования.

Таблица 14.7. *Типы, для передачи которых требуется выполнить операцию преобразования*

Тип .NET	Может быть преобразован в ...
`System.Array`	обычный массив в стиле C или в COM SAFEARRAY
`System.Boolean`	одно-, двух-, трех- или четырехбайтовые переменные со значениями 1,0 и 1
`System.Char`	однобайтовый ANSI-символ или двухбайтовый UNICODE символ
`System.Class`	интерфейс для доступа к классу (виртуальная таблица функций)
`System.Object`	либо в `Variant`, либо в интерфейс
`System.String`	одну из строк формата операционной системы в заданной кодировке (ANSI, UNICODE, BSTR)
`System.Valuetype`	структуру, члены которой непрерывно располагаются в памяти

Из приведенных правил есть два исключения, они касаются массивов и структур:

- если массив одномерный и состоит только из совместимых типов, то при его переносе не будет требоваться дополнительного преобразования. Это верно только в случае, если члены массива будут располагаться последовательно, в привычном для среды языка C стиле;
- если все типы членов структуры являются совместимыми, а также при определении структуры был использован атрибут `[StructLayout(LayoutKind.Sequential)]`, тогда для переноса структуры никаких преобразований не требуется.

В остальных случаях процесс преобразования типов будет обязательно происходить.

Копирование и блокирование данных

Если стандартные сервисы маршалинга .NET решают, что копировать данные нецелесообразно, тогда они блокируются в памяти и, вместо копирования, в неуправляемую среду передается указатель прямо на эти данные (рис. 14.14).

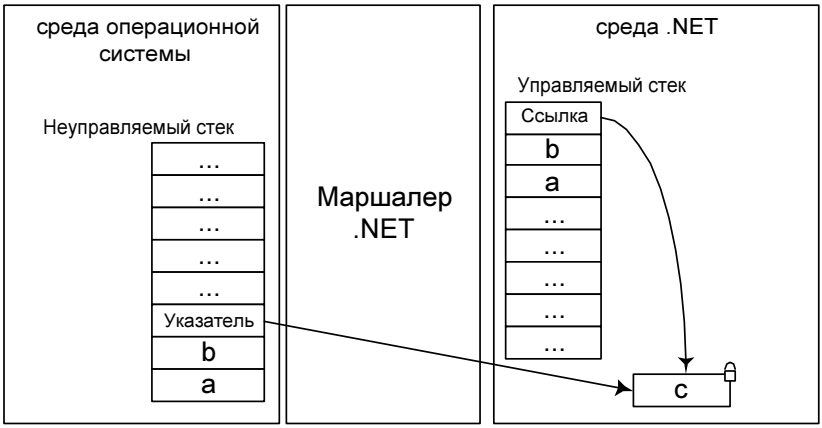


Рис. 14.14. Блокирование данных в управляемой области при маршалинге

При этом тип обязательно должен быть совместимым и передаваться по ссылке. В подавляющем большинстве случаев, маршалер прибегает к использованию данного механизма только при работе с большими объемами данных, например, массивами или структурами. Блокировать данные размером в четыре байта попросту не имеет смысла, поскольку на это может уйти больше тактов процессора, чем на их перенос. Впрочем, пользователю предоставляется возможность самостоятельно блокировать данные — именно для этого в язык Managed Extensions for C++ введено специальное ключевое слово `__pin` (или `pinned` для IL).

Приведем пример, демонстрирующий применение данной техники (листинг 14.4).

Листинг 14.4. Демонстрация блокирования данных в управляемой области памяти

```
/*
  Листинг 14.4
  File:   Some.cpp
  Author: Дубовцев Алексей
*/
```

```
// Подключим общую библиотеку классов
#using <mscorlib.dll>

// Подключим заголовок из стандартной библиотеки
#include <stdio.h>

// Введем управляемый класс
__gc class TestClass
{
public:
   // Конструктор класса проинициализирует значение
   // переменной нулем
   TestClass() {iSomeValue = 0;};

   // Опишем переменную, тип которой будет совместим
   // с неуправляемой средой
   int iSomeValue;
};

// Укажем компилятору на необходимость генерации
// неуправляемого кода
#pragma unmanaged
// Данная функция, получив указатель на данные типа
// int, изменяет их в памяти, а затем выводит на консоль их значение
void Inc(int * i)
{
  // Увеличим значение переменной на 1
  (*i)++;
  // Выведем ее значение на консоль
  printf("Unmanaged code iSomeValue = %d\n",*i);
};

// Сообщим компилятору, что с данного момента необходимо
// генерировать управляемый код
#pragma managed

// Точка входа в приложение
int main()
{
```

```
    // Создадим экземпляр управляемого класса,
   // указав, что его необходимо закрепить в памяти
    TestClass __pin * pTestClass = new TestClass;

    // Передадим адрес члена класса неуправляемому коду
    Inc(& pTestClass->iSomeValue);

    // Покажем пользователю, что значение члена класса изменилось
    // и что это видно из управляемой части кода
    System::Console::WriteLine("Managed code iSomeValue = {0}",↳
__box(pTestClass->iSomeValue));
}
```

В результате работы данного приложения на консоль будут выведены следующие строки.

```
Unmanaged code iSomeValue = 1
Managed code iSomeValue = 1
```

Таким образом, видно, что существует возможность доступа к управляемой области памяти из неуправляемого кода. Это должно быть предусмотрено заранее и поддержано со стороны управляемой части приложения.

Использование параметров, возвращающих значение

Функции могут возвращать результаты своей работы не только при помощи единственного возвращаемого значения, но также и через специальные ссылочные параметры. Данные параметры отличаются от обычных, передаваемых по ссылке, тем, что значение переменных, переданных в функцию, будет обновлено после ее вызова. Следовательно, маршалеру необходимо знать, через какие параметры функция будет возвращать результаты своей работы. Это нужно для того, чтобы скопировать их обратно после завершения работы функции. Для реализации задуманного предназначены два атрибута:

- `InAttribute` — параметр является только входным.
- `OutAttribute` — через данный параметр функция может возвращать результаты своей работы.

Данные атрибуты могут быть применены как по отдельности, так и вместе. К слову можно упомянуть, что они также представлены в языке ILD.

Помимо данных атрибутов существуют специальные ключевые слова языка, которые влияют на передачу параметров внутри самой среды исполнения и не имеют никакого отношения к маршалингу данных.

Ниже (табл. 14.8) приведены сводные правила использования данных атрибутов и ключевых слов.

Таблица 14.8. *Использование атрибутов и ключевых слов для указания способа передачи параметров в функции*

Описание типа параметра	C#	VB.NET	IDL
Входной по значению	По умолчанию, ключевого слова не требуется	`ByVal`	`[in]`
Через параметр и передается некоторое значение, и возвращается результат работы функции	`ref`	`ByRef`	`[in/out]`
Выходной	`out`		`[out]`

Неочевидность использования описанных выше механизмов в том, что, если функция имеет параметр, определенный как `out`, для правильной передачи параметра за пределы управляемой среды необходимо дополнительно пометить его атрибутом `[Out]`.

```
...
  void SomeMethod([Out] out int ParamOne);
...
```

При работе с данными атрибутами необходимо быть очень внимательными. К примеру, если вы используете ключевое слово языка `out`, но не зададите соответствующий атрибут, то вызов функции через границу сред будет работать неправильно.

Стандартные правила преобразования типов, используемые при маршализации

По умолчанию, при маршализации данных используется специальный, жестко закодированный в среде .NET набор правил. По большому счету, данный набор даже не один — их два. Один используется при взаимодействии со средой операционной системы (Dll), второй — при взаимодействии с COM. Ниже приведены оба набора правил.

Правила преобразования типов, являющихся "родными" для среды операционной системы

Приведенные далее соответствия справедливы только для 32-битных систем, в 64-битных — разрядность всех указателей, а также некоторых типов вырастет с 32 до 64 бит (табл. 14.9).

Таблица 14.9. *Набор правил, используемых средой исполнения при взаимодействии с динамическими библиотеками*

Тип WinAPI	Тип языка C(++)	Класс общей библиотеки классов	Дополнительное описание
HANDLE	void *	System.IntPtr	32 бита
BYTE	unsigned char	System.Byte	8 бит
SHORT	Short	System.Int16	16 бит
WORD	unsigned short	System.UInt32	16 бит
INT	Int	System.Int32	32 бита
UINT	unsigned int	System.UInt32	32 бита
LONG	Long	System.Int32	32 бита
BOOL	Long	System.Int32	32 бита
DWORD	unsigned long	System.UInt32	32 бита
ULONG	unsigned long	System.UInt32	32 бита
CHAR	Char	System.Char	8 бит (символ ANSI)
LPSTR	char *	System.String или System.StringBuilder	Строка ANSI
LPCSTR	const char *	System.String или System.StringBuilder	Строка ANSI
LPWSTR	wchar_t *	System.String или System.StringBuilder	Строка UNICODE
LPCWSTR	const wchar_t *	System.String или System.StringBuilder	Строка UNICODE
FLOAT	Float	System.Single	32 бита
DOUBLE	Double	System.Double	64 бита

Правила преобразования типов для среды COM

Ниже вы найдете свод правил, используемых стандартным маршалером при взаимодействии со средой COM (табл. 14.10).

Таблица 14.10. *Набор привил, используемых средой исполнения при взаимодействии со средой COM*

Тип по значению	Тип по ссылке	Класс-аналог .NET
Bool	bool *	System.Int32
char, small	char *, small *	System.Sbyte

Таблица 14.10 *(окончание)*

Тип по значению	Тип по ссылке	Класс-аналог .NET
Short	short *	System.Int16
long, int	long *, int *	System.Int32
Hyper	hyper *	System.Int64
unsigned char, byte	unsigned char *, byte *	System.Byte
wchar_t,	wchar_t *,	System.UInt16
unsigned short	unsigned short *	
unsigned long, unsigned int	unsigned long *, unsigned int *	System.UInt32
unsigned hyper	unsigned hyper *	System.UInt64
Float	float *	System.Single
Double	double *	System.Double
VARIANT_BOOL	VARIANT_BOOL *	System.Boolean
void*	void **	System.IntPrt
HRESULT	HRESULT *	System.Int16 или System.IntPtr
SCODE	SCODE *	System.Int32
BSTR	BSTR *	System.String
LPSTR или [string,...] char *	LPSTR *	System.String
LPWSTR или [string,...] wchar_t *	LPWSTR *	System.String
VARIANT	VARIANT *	System.Object
DECIMAL	DECIMAL *	System.Decimal
DATE	DATE *	System.DataTime
GUID	GUID *	System.Guid
CURRENCY	CURRENCY *	System.Decimal
IUnknown *	IUnknown **	System.Object
IDispatch *	IDispatch **	System.Object
SAFEARRAY(тип)	SAFEARRAY(тип) *	тип[]

Единственный, возможно, непонятный момент здесь — преобразование `IDispatch` в класс `Object`. Попробую удивить вас, заявив, что на самом деле мы встречались с подобной ситуацией, когда использовали метод `Activator.CreateInstance` для динамического создания COM-объекта.

Тогда, для взаимодействия с внешним объектом, мы использовали технологию отражения, которая и использовала данное преобразование.

Изменение правил преобразования типов

Программная среда .NET является крайне гибкой, и дополнительным доказательством этого служит возможность отступления от стандартных правил преобразования типов. Для этого создатели среды дали нам в руки атрибут `MarshalAs`, расположенный в пространстве имен `System.Runtime.InteropServices`.

```
public MarshalAsAttribute(
   UnmanagedType unmanagedType
);
```

В качестве обязательного позиционного параметра его конструктора должен быть задан один из членов перечисления `UnmanagedType`. Он определит тип, который будет использоваться для преобразования во время маршализации данных. Члены данного перечисления описаны ниже (табл. 14.11).

Таблица 14.11. *Описание членов перечисления* `System.Runtime.InteropServices.UnmanagedType`

Член перечисления (Флаг)		Описание
✓	`AnsiBStr`	Строка в формате ANSI, первый байт которой указывает ее длину (строки в стиле Pascal)
✓	`AsAny`	Определяет тип объекта во время исполнения программы и производит соответствующее преобразование типов
✓	`Bool`	Четырехбайтовое булево значение, где `true != 0, false = 0`
✓	`BStr`	Строка в формате BSTR, первые два байта которой указывают ее длину (стандарт хранения строк для среды COM)
✓	`ByValArray`	Используется для массивов элементов, которые необходимо передавать по значению. Размер массива указывается при помощи параметра `SizeConst`, атрибута `MarshalAs`
✓	`ByValTStr`	Непрерывный массив символов. При этом размер символов определяется значением параметра `CharSet`, атрибута `StructLayoutAttribute`
✓	`Currency`	Формат представления денег в .NET System.Decimal
✓	`CustomMarshaler`	Позволяет использовать пользовательский маршалер

Таблица 14.11 *(продолжение)*

Член перечисления (Флаг)	Описание
Error	Ошибка, в формате I4 или U4. Преобразуется в HRESULT
FunctionPtr	Указатель на функцию
I1	Однобайтовое знаковое целое
I2	Двухбайтовое знаковое целое
I4	Четырехбайтовое знаковое целое
I8	Восьмибайтовое знаковое целое
IDispatch	Указатель на интерфейс IDispatch
Interface	Указатель на COM-интерфейс, идентификатор которого определяется при помощи атрибута Guid
IUnknown	Указатель на интерфейс IUnknown
LPArray	Массив в стиле C
LPStr	Однобайтовая строка в формате ANSI, заканчивающаяся нулем
LPStruct	Указатель на структуру, оформленную в формате C
LPTStr	Строка, формат которой зависит от платформы, на которой исполняется в данный момент среда .NET. В операционных системах линейки 9x — это строки ANSI-формата, на системах NT — это UNICODE-строки
LPWStr	Двухбайтовая строка в формате UNICODE, завершающаяся двумя нулями
R4	Четырехбайтовый тип с плавающей точкой
R8	Восьмибайтовый тип с плавающей точкой
SafeArray	Массив в стиле COM (SAFEARRAY)
Struct	Структура в стиле C
SysInt	Знаковый целый тип, формат которого зависит от платформы. На 32-битных платформах — 4 байта, на 64-битных платформах — 8 байт
SysUInt	Беззнаковый целый тип, формат которого не зависит от платформы
TBStr	Стока, формат символов которой зависит от платформы. Первый(е) байт указывает ее длину (9x - ANSI, NT - UNICODE)
U1	Однобайтовое беззнаковое целое

Таблица 14.11 (окончание)

Член перечисления (Флаг)	Описание
U2	Двухбайтовое беззнаковое целое
U4	Четырехбайтовое беззнаковое целое
U8	Восьмибайтовое беззнаковое целое
VariantBool	Двухбайтовое булевое значение в стиле OLE (`true = -1. false = 0`)
VBByRefStr	Строка, передаваемая по ссылке. Позволяет изменять буфер строки, находящийся в управляемой области памяти, из неуправляемого кода

Приведем определение атрибута `MarshalAs`:

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Parameter
   | AttributeTargets.ReturnValue)]
public sealed class MarshalAsAttribute : Attribute
```

Из него видно, что он может применяться к полям, параметрам функций, а также значениям, возвращаемым функциями. По поводу последней возможности хотелось бы поговорить отдельно, поскольку ее использование не является очевидным. Для этого необходимо воспользоваться довольно хитрым трюком — при определении атрибута добавить ключевое слово `return`. Выглядеть это будет так:

```
[return : MarshalAs( UnmanagedType.Interface )]
      Object SomeMethod();
```

В результате, при маршалинге объект, возвращаемый функцией, будет передаваться в виде COM-интерфейса.

Кроме обязательного позиционного параметра данный атрибут имеет шесть именованных. Они описаны далее (табл. 14.12).

Таблица 14.12. *Именованные параметры атрибута* `System.Runtime.InteropServices`

Поле атрибута (именованный параметр)	Описание
ArraySubType	Тип элементов в неуправляемом массиве
MarshalCookie	Позволяет передать дополнительную информацию пользовательскому маршалеру
MarshalType	Задает имя типа пользовательского маршалера

Таблица 14.12 *(окончание)*

Поле атрибута (именованный параметр)	Описание
MarshalTypeRef	Задает в явном виде тип пользовательского маршалера
SafeArraySubType	Тип элементов SAFEARRAY массива
SafeArrayUserDefinedSubType	Позволяет задать пользовательский тип элементов SAFEARRAY массива
SizeConst	Определяет количество элементов при маршалинге массивов
SizeParamIndex	Определяет, в каком из параметров функции будет передан размер массива (size_is в COM)

Атрибут MarshalType позволяет определить пользовательский маршалер, который здесь рассматриваться не будет. Возможностей стандартного маршалера .NET в подавляющем большинстве случаев хватает с головой.

Отдельно пояснения требует поле SafeArraySubType, которое позволяет задать один из типов Variant, указывающих тип элементов массива. Прототип поля представлен ниже:

```
public VarEnum SafeArraySubType;
```

Данное поле может принимать значения флагов, входящих в состав перечисления VarEnum. Описание полей перечисления представлено далее (табл. 14.13).

Таблица 14.13. *Описание членов перечисления* *System.Runtime.InteropServices.VarEnum*

Член перечисления (Флаг)	Описание
VT_ARRAY	Указатель на SAFEARRAY
VT_BLOB	Binary Large Object (BLOB, Большие данные в двоичном формате)
VT_BLOB_OBJECT	Указывает на то, что BLOB содержит объект
VT_BOOL	Булево значение
VT_BSTR	Строка в формате BSTR
VT_BYREF	Указывает на то, что значение передается по ссылке
VT_CARRAY	Массив в стиле языка C
VT_CF	Формат данных буфера обмена

Таблица 14.13 *(продолжение)*

Член перечисления (Флаг)	Описание
VT_CLSID	Идентификатор класса COM (CLSID)
VT_CY	Денежное значение
VT_DATE	Дата в формате DATE
VT_DECIMAL	Тип данных decimal
VT_DISPATCH	Указатель на IDispatch
VT_EMPTY	Используется, для того чтобы указать, что значение не задано
VT_ERROR	Ошибка в формате SCODE
VT_FILETIME	Структура FILETIME
VT_HRESULT	Результат работы COM-функции в формате HRESULT
VT_I1	Число в формате char
VT_I2	Число в формате short
VT_I4	Число в формате long
VT_I8	64-битное число
VT_INT	Целое значение, размер которого зависит от разрядности платформы, для которой компилировалось данное приложение
VT_LPSTR	Указатель на однобайтовую ANSI-строку, завершающуюся нулем
VT_LPWSTR	Указатель на двухбайтовую UNICODE-строку, завершающуюся двумя нулями
VT_NULL	Нулевая ссылка
VT_PTR	Данные являются указателем
VT_R4	Число в формате float
VT_R8	Число в формате double
VT_RECORD	Данные в пользовательском формате (типе)
VT_SAFEARRAY	Массив в стиле COM
VT_STORAGE	Имя хранилища
VT_STORED_OBJECT	Определяет то, что хранилище содержит объект
VT_STREAM	Имя потока данных
VT_STREAMED_OBJECT	Определяет то, то поток содержит объект

Таблица 14.13 *(окончание)*

Член перечисления (Флаг)	Описание
VT_UI1	Число в формате byte
VT_UI2	Число в формате unsigned short
VT_UI4	Число в формате unsigned long
VT_UI8	64-битное беззнаковое число
VT_UINT	Целое беззнаковое число, размер которого зависит от разрядности платформы, для которой компилировалось данное приложение
VT_UNKNOWN	Указатель на IDispatch
VT_USERDEFINED	Пользовательский тип
VT_VARIANT	Указатель на тип данных VARIANT
VT_VECTOR	Простой линейный массив
VT_VOID	Тип void в стиле языка C

При работе с сервисами автоматизации COM, вам частенько придется сталкиваться с типом VARIANT, а соответственно и обращаться к данному перечислению.

Краткое описание дополнительных сервисов взаимодействия с неуправляемым кодом

Осветив достаточно большое количество тем, относящихся к взаимодействию с неуправляемым кодом, я понял, что полностью описать данную тему просто невозможно. В настоящем разделе будет подведен итог, где вы найдете краткое описание сервисов, рассмотрение которых не вошло в книгу. Просто знание об их существовании может вам очень пригодиться (табл. 14.14).

Таблица 14.14. *Краткое описание сервисов взаимодействия с неуправляемым кодом, описание которых не вошло в книгу. Все они являются членами класса* `System.Runtime.InteropServices.Marshal`

Метод класса	Описание
AddRef	Увеличивает счетчик ссылок интерфейса
AllocCoTaskMem	Выделяет блок памяти при помощи сервиса COM CoTaskMemAlloc

Таблица 14.14 *(продолжение)*

Метод класса	Описание
AllocHGlobal	Выделяет блок памяти при помощи GlobalAlloc
BindToMoniker	Позволяет получить указатель на интерфейс при помощи моникера
Copy	Позволяет скопировать данные из .NET-массива в неуправляемую область памяти
CreateWrapperOfType	Создает для COM-объекта обертку заданного типа
DestroyStructure	Освобождает память, занятую членами структуры (даже если они являются указателями)
FreeBSTR	Освобождает память, занятую строкой в формате BSTR при помощи SysFreeString
FreeCoTaskMem	Освобождает блок памяти при помощи API-функции CoTaskMemFree подсистемы COM
FreeHGlobal	Освобождает блок памяти при помощи функции GlobalFree
GenerateGuidForType	Генерирует GUID для указанного типа, который по алгоритму использует утилита tlbexp.exe
GenerateProgIdForType	Возвращает ProgId для указанного типа
GetComInterfaceForObject	Возвращает определенный интерфейс для заданного COM-объекта
GetComObjectData	Позволяет получить данные по определенному ключу, сопоставленные с данным COM-объектом, при помощи метода SetComObjectData
GetComSlotForMethodInfo	Позволяет узнать номер в таблице виртуальных функций для определенного метода
GetEndComSlot	Позволяет получить индекс последней ячейки в виртуальной таблице интерфейса
GetExceptionCode	Позволяет узнать низкоуровневый код произошедшего исключения

Таблица 14.14 *(продолжение)*

Метод класса	Описание
 GetExceptionPointers	Возвращает указатель на структуру EXCEPTION_POINTERS, описывающую произошедшее исключение
 GetHINSTANCE	Позволяет получить описатель заданного модуля
 GetHRForException	Позволяет получить значение поля HResult заданного объекта Exception или производного от него
 GetHRForLastWin32Error	Позволяет преобразовать последний код ошибки Win32 в HRESULT
 GetIDispatchForObject	Возвращает указатель на интерфейс IDispatch для определенного COM-объекта
 GetITypeInfoForType	Позволяет получить интерфейс ITypeInfo для определенного типа
 GetIUnknownForObject	Возвращает указатель на интерфейс IUnknown для заданного COM-объекта
 GetLastWin32Error	Возвращает код последней произошедшей ошибки
 GetManagedThunkFor UnmanagedMethodPtr	Создает управляемую заглушку для вызова неуправляемого метода
 GetMethodInfoForComSlot	Позволяет получить описание метода по индексу в таблице виртуальных функций
 GetNativeVariantForObject	Позволяет получить из Object тип VARIANT
 GetObjectForIUnknown	Создает объект Object, который инкапсулирует указатель на интерфейс IUnknown
 GetObjectForNativeVariant	Создает объект для заданного типа VARIANT
 GetObjectsForNativeVariants	Аналогичен предыдущему методу, только преобразование проводится для группы объектов
 GetStartComSlot	Возвращает индекс первого элемента в виртуальной таблице методов
 GetThreadFromFiberCookie	Позволяет получить объект потока по специальному служебному идентификатору

Таблица 14.14 (продолжение)

Метод класса	Описание
GetTypedObjectForIUnknown	Возвращает объект заданного типа, который инкапсулирует интерфейс IUnknown
GetTypeForITypeInfo	Возвращает тип, представляющий доступ к определенному интерфейсу ITypeInfo
GetTypeInfoName	Возвращает имя типа, представленного при помощи заданного интерфейса ITypeInfo
GetTypeLibGuid	Возвращает идентификатор COM библиотеки типов
GetTypeLibGuidForAssembly	Возвращает идентификатор библиотеки типов, преобразованной в сборку
GetTypeLibLcid	Возвращает региональную информацию о библиотеке типов
GetTypeLibName	Возвращает имя библиотеки типов
GetUnmanagedThunkFor ManagedMethodPtr	Создает заглушку для вызова управляемой функции из неуправляемого кода
IsComObject	Позволяет узнать, является ли объект импортированным из среды COM
IsTypeVisibleFromCom	Позволяет узнать, экспортируется ли объект в среду COM
NumParamBytes	Вычисляет количество памяти в байтах, необходимое для удержания параметров заданного метода
OffsetOf	Позволяет узнать адрес, по которому хранится заданное поле определенного класса
Prelink	Подготавливает метод к работе. Обычно это происходит автоматически при первом вызове метода
PrelinkAll	Подготавливает к работе все методы определенного класса
PtrToStringAnsi	Преобразует строку в формате ANSI, которая задается при помощи указателя, в объект типа System.String

Таблица 14.14 *(продолжение)*

Метод класса	Описание
 PtrToStringAuto	Преобразует строку, заданную при помощи указателя, в объект типа System.String, при этом формат строки подбирается автоматически
 PtrToStringBSTR	Преобразует строку в формате BSTR, которая задается при помощи указателя, в объект типа System.String
 PtrToStringUni	Преобразует строку в формате Unicode, которая задается при помощи указателя, в объект типа System.String
 PtrToStructure	Создает из данных, переданных по указателю, объект определенного типа
 QueryInterface	Позволяет запросить необходимый интерфейс, через функцию QueryInterface
 ReadByte	Позволяет прочитать данные размером в байт по определенному заданному адресу
 ReadInt16	Позволяет прочитать два байта по заданному адресу
 ReadInt32	Позволяет прочитать четыре байта по заданному адресу
 ReadInt64	Позволяет прочитать восемь байт по заданному адресу
 ReadIntPtr	Позволяет прочитать указатель (размер которого варьируется от платформы) по определенному адресу
 ReAllocCoTaskMem	Позволяет произвести повторное резервирование памяти при помощи CoTaskMemRealloc
 ReAllocHGlobal	Позволяет произвести повторное резервирование памяти при помощи GlobalReAlloc
 Release	Уменьшает счетчик ссылок определенного интерфейса
 ReleaseComObject	Уменьшает счетчик ссылок, хранящийся внутри обертки, предоставляющей доступ к некоторому интерфейсу

Таблица 14.14 *(продолжение)*

Метод класса	Описание
ReleaseThreadCache	Сбрасывает кэш текущего потока (используется для внутренних нужд среды исполнения)
SetComObjectData	Связывает с COM-объектом любые данные при помощи заданного ключа
SizeOf	Позволяет узнать размер определенного типа или объекта в байтах
StringToBSTR	Преобразует объект String в строку формата BSTR
StringToCoTaskMemAnsi	Преобразует объект String в строку формата ANSI. При этом память резервируется при помощи сервисов подсистемы COM
StringToCoTaskMemAuto	Преобразует строку в формат, выбираемый автоматически, на основе атрибутов, примененных к данному объекту или в зависимости от текущей операционной системы. При этом память резервируется при помощи сервисов подсистемы COM
StringToCoTaskMemUni	Преобразует объект String в строку формата Unicode. При этом память резервируется при помощи сервисов операционной системы
StringToHGlobalAnsi	Преобразует объект String в строку формата ANSI. При этом память резервируется при помощи сервисов операционной системы
StringToHGlobalAuto	Преобразует строку в формат, определяемый автоматически, на основе атрибутов, примененных к данному объекту или в зависимости от текущей операционной системы. При этом память резервируется при помощи сервисов операционной системы
StringToHGlobalUni	Преобразует объект String в строку формата UNICODE. При этом память резервируется при помощи сервисов операционной системы
StructureToPtr	Копирует данные из структуры в единый блок памяти, на которые возвращает указатель

Таблица 14.14 *(окончание)*

Метод класса	Описание
ThrowExceptionForHR	Выбрасывает исключение с заданным значением Hresult
UnsafeAddrOfPinnedArrayElement	Возвращает адрес определенного элемента в массиве
WriteByte	Позволяет записать данные размером в байт по определенному заданному адресу
WriteInt16	Позволяет записать два байта по заданному адресу
WriteInt32	Позволяет записать четыре байта по заданному адресу
WriteInt64	Позволяет записать восемь байт по заданному адресу
WriteIntPtr	Позволяет записать указатель (размер которого варьируется от платформы) по определенному адресу

Конечно, хотелось бы описать каждый из данных методов более подробно. Но они обязательно потащат за собой еще что-нибудь, а главу необходимо заканчивать, поскольку она и так уже сильно разрослась.

14.4. Заключение

В данной главе было рассмотрено огромное количество материала как прикладного, так и фундаментального характера. Даже если вы разберетесь хотя бы в половине написанного, проблем при программировании взаимодействия с операционной системой из управляемого кода у вас не будет.

АМИНЬ

Описание компакт-диска

На компакт-диске приведены листинги и примеры из книги. Вся информация разбита по главам. Названия каталогов соответствуют главам в книге:

- Chapter 1 — листинги и примеры к главе "Общеязыковая среда исполнения"
- Chapter 2 — листинги и примеры к главе "Метаданные"
- Chapter 3 — листинги и примеры к главе "Общая система типов"
- Chapter 4 — листинги и примеры к главе "Сборки"
- Chapter 5 — листинги и примеры к главе "Атрибуты"
- Chapter 6 — листинги и примеры к главе "Делегаты и события"
- Chapter 7 — листинги и примеры к главе "Обработка исключений"
- Chapter 8 — листинги и примеры к главе "Управление памятью"
- Chapter 9 — листинги и примеры к главе "Потоки"
- Chapter 10 — листинги и примеры к главе "Архитектура доменов"
- Chapter 11 — листинги и примеры к главе "Введение во взаимодействие с операционной системой"
- Chapter 12 — листинги и примеры к главе "Использование COMпонентов при помощи .NET"
- Chapter 13 — листинги и примеры к главе "Написание COMпонентов при помощи .NET"
- Chapter 14 — листинги и примеры к главе "Тонкости взаимодействия с COM"

Предметный указатель

Б

В

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ч

Э