Глава 1. Модель выполнения кода в среде CLR

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

Компиляция исходного кода в управляемые модули

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

Теперь надо выбрать язык программирования. И это непростая задача — ведь у разных языков имеются разные возможности. Например, с одной стороны, «неуправляемый код» C/C++ дает доступ к системе на низком уровне. Вы вправе распоряжаться памятью по своему усмотрению, при необходимости создавать программные потоки и т. д. С другой стороны, Microsoft Visual Basic 6.0 позволяет очень быстро строить пользовательские интерфейсы и легко управлять COM-объектами и базами данных.

Название среды — *общеязыковая среда выполнения* (Common Language Runtime, CLR) — говорит само за себя: это среда выполнения, которая подходит для разных языков программирования. Основные возможности CLR (управление памятью, загрузка сборок, безопасность, обработка исключений, синхронизация) доступны в любых языках программирования, использующих эту среду. Например, при обработке ошибок среда выполнения опирается на исключения, а значит, во всех языках программирования, использующих эту среду выполнения, сообщения об ошибках передаются при помощи механизма исключений. Или, например, среда выполнения позволяет создавать программные потоки, а значит, во всех языках программирования, использующих эту среду, тоже могут создаваться потоки.

Фактически во время выполнения программы в среде CLR неизвестно, на каком языке программирования разработчик написал исходный код. А это значит, что можно выбрать любой язык программирования, который позволяет проще всего решить конкретную задачу. Разрабатывать программное обеспечение можно на любом языке программирования, если только используемый компилятор этого языка поддерживает CLR.

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

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

Рисунок 1.1 иллюстрирует процесс компиляции файлов с исходным кодом. Как видно из рисунка, исходный код программы может быть написан на любом языке, поддерживающем среду выполнения CLR. Затем соответствующий компилятор проверяет синтаксис и анализирует исходный код программы. Вне зависимости от типа используемого компилятора результатом компиляции будет являться *управляемый модуль* (managed module) — стандартный переносимый исполняемый (portable executable, PE) файл 32-разрядной (PE32) или 64-разрядной Windows (PE32+), который требует для своего выполнения CLR. Кстати, управляемые сборки всегда используют преимущества функции безопасности «предотвращения выполнения данных» (DEP, Data Execution Prevention) и технологию ASLR (Address Space Layout Optimization), применение этих технологий повышает информационную безопасность всей системы.

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

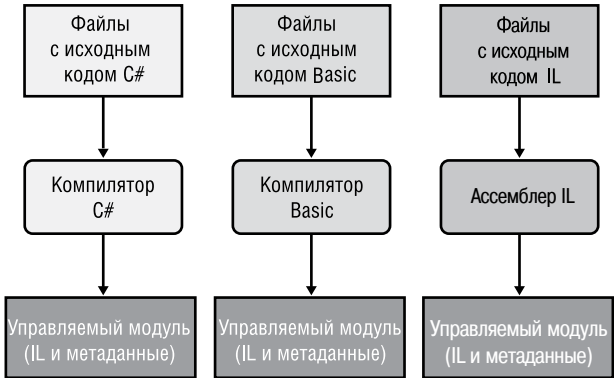

Рис. 1.1. Компиляция исходного кода в управляемые модули

В табл. 1.1 описаны составные части управляемого модуля.

Таблица 1.1. Части управляемого модуля

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

Каждый компилятор, предназначенный для CLR, помимо генерирования IL-кода, должен также создавать полные *метаданные* (metadata) для каждого управляемого модуля. Проще говоря, метаданные — это набор таблиц данных, описывающих то, что определено в модуле, например типы и их члены. В метаданных также есть таблицы, указывающие, на что ссылается управляемый модуль, например на импортируемые типы и их члены. Метаданные расширяют возможности таких старых технологий, как библиотеки типов COM и файлы IDL (Interface Definition Language, язык описания интерфейсов). Важно отметить, что метаданные CLR содержат куда более полную информацию. И, в отличие от библиотек типов и IDL-файлов, они всегда связаны с файлом, содержащим IL-код. Фактически метаданные всегда встроены в тот же EXE- или DLL-файл, что и код, так что их нельзя разделить. А поскольку компилятор генерирует метаданные и код одновременно и привязывает их к конечному управляемому модулю, возможность рассинхронизации метаданных и описываемого ими IL-кода исключена.

Метаданные находят много применений. Перечислим лишь некоторые из них.

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

В главе 2 метаданные описаны более подробно.

Языки программирования C#, Visual Basic, F# и IL-ассемблер всегда создают модули, содержащие управляемый код (IL) и управляемые данные (данные, поддерживающие сборку мусора). Для выполнения любого управляемого модуля на машине конечного пользователя должна быть установлена среда CLR (в составе .NET Framework) — так же, как для выполнения приложений MFC или Visual Basic 6.0 должны быть установлены библиотека классов Microsoft Foundation Class (MFC) или DLL-библиотеки Visual Basic.

По умолчанию компилятор Microsoft C++ создает EXE- и DLL-файлы, которые содержат неуправляемый код и неуправляемые данные. Для их выполнения CLR не требуется. Однако если вызвать компилятор C++ с параметром `/CLR` в командной строке, он создаст управляемые модули (и конечно, для работы этих модулей должна быть установлена среда CLR). Компилятор C++ стоит особняком среди всех упомянутых компиляторов производства Microsoft — только он позволяет разработчикам писать как управляемый, так и неуправляемый код и встраивать его в единый модуль. Это также единственный компилятор Microsoft, разрешающий программистам определять в исходном коде как управляемые, так и неуправляемые типы данных. Компилятор Microsoft предоставляет разработчику непревзойденную гибкость, позволяя использовать существующий неуправляемый код на C/C++ из управляемого кода и постепенно, по мере необходимости, переходить на управляемые типы.

Объединение управляемых модулей в сборку

На самом деле среда CLR работает не с модулями, а со сборками. *Сборка* (assembly) — это абстрактное понятие, понять смысл которого на первых порах бывает нелегко. Во-первых, сборка обеспечивает логическую группировку одного или нескольких управляемых модулей или файлов ресурсов. Во-вторых, это наименьшая единица многократного использования, безопасности и управления версиями. Сборка может состоять из одного или нескольких файлов — все зависит от выбранных средств и компиляторов. В контексте среды CLR сборкой называется то, что мы обычно называем *компонентом*.

О сборках довольно подробно рассказано в главе 2, а здесь достаточно подчеркнуть, что это концептуальное понятие обозначает способ объединения группы файлов в единую сущность.

Рисунок 1.2 поможет понять суть сборки. На этом рисунке изображены некоторые управляемые модули и файлы ресурсов (или данных), с которыми работает некоторая программа. Эта программа создает единственный файл PE32(+), который обеспечивает логическую группировку файлов. При этом в файл PE32(+) включаетсяч блок данных, называемый *манифестом* (manifest). Манифест представляет собой обычный набор таблиц метаданных. Эти таблицы описывают файлы, которые входят в сборку, общедоступные экспортируемые типы, реализованные в файлах сборки, а также относящиеся к сборке файлы ресурсов или данных.

По умолчанию компиляторы сами выполняют работу по преобразованию созданного управляемого модуля в сборку, то есть компилятор C# создает управляемый модуль с манифестом, указывающим, что сборка состоит только из одного файла. Таким образом, в проектах, где есть только один управляемый модуль и нет файлов

ресурсов (или файлов данных), сборка и является управляемым модулем, поэтому выполнять дополнительные действия по компоновке приложения не нужно. В случае если необходимо сгруппировать несколько файлов в сборку, потребуются дополнительные инструменты (например, компоновщик сборок AL.exe) со своими параметрами командной строки. О них подробно рассказано в главе 2.

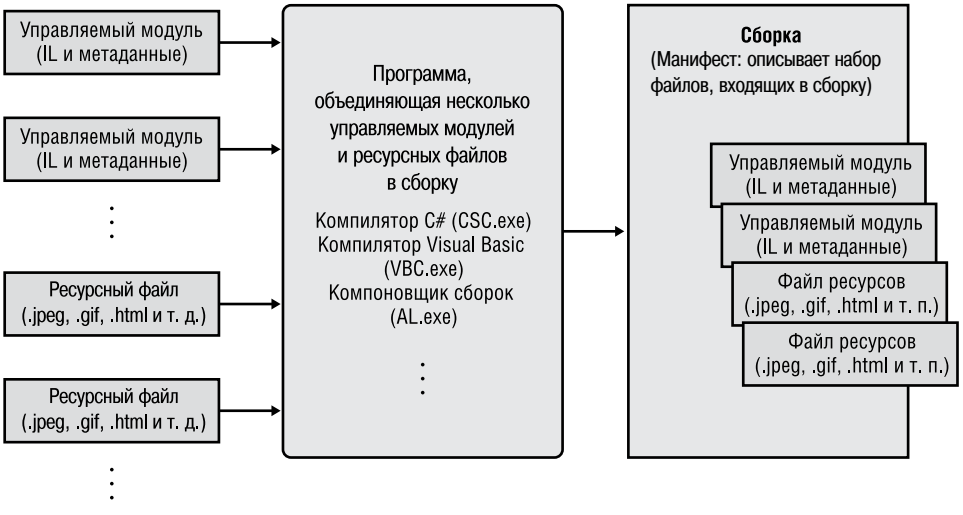

Рис. 1.2. Объединение управляемых модулей в сборку

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

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

Загрузка CLR

Каждая создаваемая сборка представляет собой либо исполняемое приложение, либо библиотеку DLL, содержащую набор типов для использования в исполняемом приложении. Разумеется, среда CLR отвечает за управление исполнением кода. Это значит, что на компьютере, выполняющем данное приложение, должна быть установлена платформа .NET Framework. В компании Microsoft был создан дистрибутивный пакет .NET Framework для свободного распространения, который вы можете бесплатно поставлять своим клиентам. Некоторые версии операционной системы семейства Windows поставляются с уже установленной платформой .NET Framework.

Для того чтобы понять, установлена ли платформа .NET Framework на компьютере, попробуйте найти файл MSCorEE.dll в каталоге %SystemRoot%\system32. Если он есть, то платформа .NET Framework установлена. Однако на одном компьютере может быть установлено одновременно несколько версий .NET Framework. Чтобы определить, какие именно версии установлены, проверьте содержимое следующих подкаталогов:

```
%SystemRoot%\Microsoft.NET\Framework
%SystemRoot%\Microsoft.NET\Framework64
```

Компания Microsoft включила в .NET Framework SDK утилиту командной строки CLRVer.exe, которая выводит список всех версий CLR, установленных на машине, а также сообщает, какая именно версия среды CLR используется текущими процессами. Для этого нужно указать параметр `-all` или идентификатор интересующего процесса.

Прежде чем переходить к загрузке среды CLR, поговорим поподробнее об особенностях 32- и 64-разрядных версий операционной системы Windows. Если сборка содержит только управляемый код с контролем типов, она должна одинаково хорошо работать на обеих версиях системы. Дополнительной модификации исходного кода не требуется. Более того, созданный компилятором готовый EXE- или DLL-файл будет правильно выполняться в Windows версий x86 и x64, а библиотеки классов и приложения Windows Store будут работать на машинах с Windows RT (использующих процессор ARM). Другими словами, один и тот же файл будет работать на любом компьютере с установленной платформой .NET Framework.

В исключительно редких случаях разработчикам приходится писать код, совместимый только с какой-то конкретной версией Windows. Обычно это требуется при работе с небезопасным кодом (unsafe code) или для взаимодействия с неуправляемым кодом, ориентированным на конкретную процессорную архитектуру. Для таких случаев у компилятора C# предусмотрен параметр командной строки `/platform`. Этот параметр позволяет указать конкретную версию целевой платформы, на которой планируется работа данной сборки: архитектуру x86, использующую только 32-разрядную систему Windows, архитектуру x64, использующую только 64-разрядную операционную систему Windows, или архитектуру ARM, на которой работает только

32-разрядная Windows RT. Если платформа не указана, компилятор задействует значение по умолчанию `anycpu`, которое означает, что сборка может выполняться в любой версии Windows. Пользователи Visual Studio могут указать целевую платформу в списке Platform Target на вкладке Build окна свойств проекта (рис. 1.3).

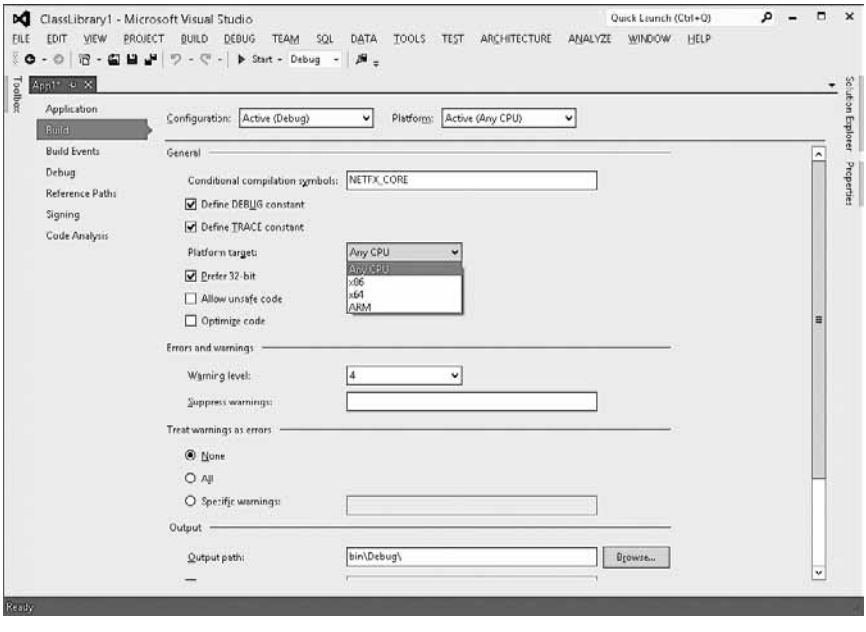

Рис. 1.3. Определение целевой платформы средствами Visual Studio

На рис. 1.3 обратите внимание на флажок Prefer 32-Bit. Он доступен только в том случае, когда в списке Platform Target выбрана строка Any CPU, а для выбранного типа проекта создается исполняемый файл. Если установить флажок Prefer 32-Bit, то Visual Studio запускает компилятор C# с параметром командной строки `/platform: anycpu32bitpreferred`. Этот параметр указывает, что исполняемый файл должен выполняться как 32-разрядный даже на 64-разрядных машинах. Если вашему приложению не нужна дополнительная память, доступная для 64-разрядных процессов, обычно стоит выбрать именно этот режим, потому что Visual Studio не поддерживает функцию «Изменить и продолжить» (Edit-and-Continue) для приложений x64. Кроме того, 32-разрядные приложения могут взаимодействовать с 32-разрядными библиотеками DLL и компонентами COM, если этого потребует ваше приложение.

В зависимости от указанной целевой платформы C# генерирует заголовок — PE32 или PE32+, а также включает в него требуемую процессорную архитектуру (или признак независимости от архитектуры). Для анализа заголовочной информации, вставленной компилятором в управляемый модуль, Microsoft предоставляет две утилиты — DumpBin.exe и CorFlags.exe.

При запуске исполняемого файла Windows анализирует заголовок EXE-файла для определения того, какое именно адресное пространство необходимо для его работы — 32- или 64-разрядное. Файл с заголовком PE32 может выполняться в адресном пространстве любого из указанных двух типов, а файлу с заголовком PE32+ требуется 64-разрядное пространство. Windows также проверяет информацию о процессорной архитектуре на совместимость с заданной конфигурацией. Наконец, 64-разрядные версии Windows поддерживают технологию выполнения 32-разрядных приложений в 64-разрядной среде, которая называется *WoW64* (Windows on Windows64).

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

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

Значение параметра /platform	Тип выходного управляемого модуля	x86 Windows	x64 Windows	ARM Windows RT
anycpu (по умолчанию)	PE32/независимый от платформы	Выполняется как 32-разрядное приложение	Выполняется как 64-разрядное приложение	Выполняется как 32-разрядное приложение
anycpu32bit-preferred	PE32/независимый от платформы	Выполняется как 32-разрядное приложение	Выполняется как WoW64-приложение	Выполняется как 32-разрядное приложение
x86	PE32/x86	Выполняется как 32-разрядное приложение	Выполняется как WoW64-приложение	Не выполняется
x64	PE32+/x64	Не выполняется	Выполняется как 64-разрядное приложение	Не выполняется
ARM	PE32+/Itanium	Не выполняется	Не выполняется	Выполняется как 32-разрядное приложение

После анализа заголовка EXE-файла для выяснения того, какой процесс необходимо запустить — 32- или 64-разрядный, — Windows загружает в адресное

пространство процесса соответствующую версию библиотеки MSCorEE.dll (x86, x64 или ARM). В системах Windows семейств x86 и ARM 32-разрядная версия MSCorEE.dll хранится в каталоге %SystemRoot%\System32. В системах x64 версия x86 библиотеки находится в каталоге %SystemRoot%\SysWow64, а 64-разрядная версия MSCorEE.dll размещается в каталоге %SystemRoot%\System32 (это сделано из соображений обратной совместимости). Далее основной поток вызывает определенный в библиотеке MSCorEE.dll метод, который инициализирует CLR, загружает сборку EXE, а затем вызывает ее метод `Main`, в котором содержится точка входа. На этом процедура запуска управляемого приложения считается завершенной[1].

ПРИМЕЧАНИЕ

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

Когда неуправляемое приложение вызывает функцию Win32 `LoadLibrary` для загрузки управляемой сборки, Windows автоматически загружает и инициализирует CLR (если это еще не сделано) для обработки содержащегося в сборке кода. Ясно, что в такой ситуации предполагается, что процесс запущен и работает, и это сокращает область применимости сборки. В частности, управляемая сборка, скомпилированная с параметром `/platform:x86`, не сможет загрузиться в 64-разрядном процессе, а исполняемый файл с таким же параметром загрузится в режиме WoW64 на компьютере с 64-разрядной Windows.

Исполнение кода сборки

Как говорилось ранее, управляемые модули содержат метаданные и программный код IL. Это не зависящий от процессора машинный язык, разработанный компанией Microsoft после консультаций с несколькими коммерческими и академическими организациями, специализирующимися на разработке языков и компиляторов. IL — язык более высокого уровня по сравнению с большинством других машинных языков. Он позволяет работать с объектами и имеет команды для создания

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

и инициализации объектов, вызова виртуальных методов и непосредственного манипулирования элементами массивов. В нем даже есть команды инициирования и перехвата исключений для обработки ошибок. IL можно рассматривать как объектно-ориентированный машинный язык.

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

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

Узнать о возможностях CLR, доступных при использовании конкретного языка, можно только при изучении соответствующей документации. В этой книге сделан акцент на возможностях среды CLR и на том, какие из этих возможностей доступны при программировании на C#. Подозреваю, что в других книгах и статьях среда CLR рассматривается с точки зрения других языков и разработчики получают представление лишь о тех ее функциях, которые доступны при использовании описанных там языков. Впрочем, если выбранный язык решает поставленные задачи, такой подход не так уж плох.

ВНИМАНИЕ

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

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

На рис. 1.4 показано, что происходит при первом вызове метода.

Непосредственно перед исполнением метода `Main` среда CLR находит все типы данных, на которые ссылается программный код метода `Main`. При этом CLR выделяет внутренние структуры данных, используемые для управления доступом к типам, на которые есть ссылки. На рис. 1.4 метод `Main` ссылается на единственный тип — `Console`, и среда CLR выделяет единственную внутреннюю структуру. Эта внутренняя структура данных содержит по одной записи для каждого метода,

определенного в типе `Console`. Каждая запись содержит адрес, по которому можно найти реализацию метода. При инициализации этой структуры CLR заносит в каждую запись адрес внутренней недокументированной функции, содержащейся в самой среде CLR. Я обозначаю эту функцию `JITCompiler`.

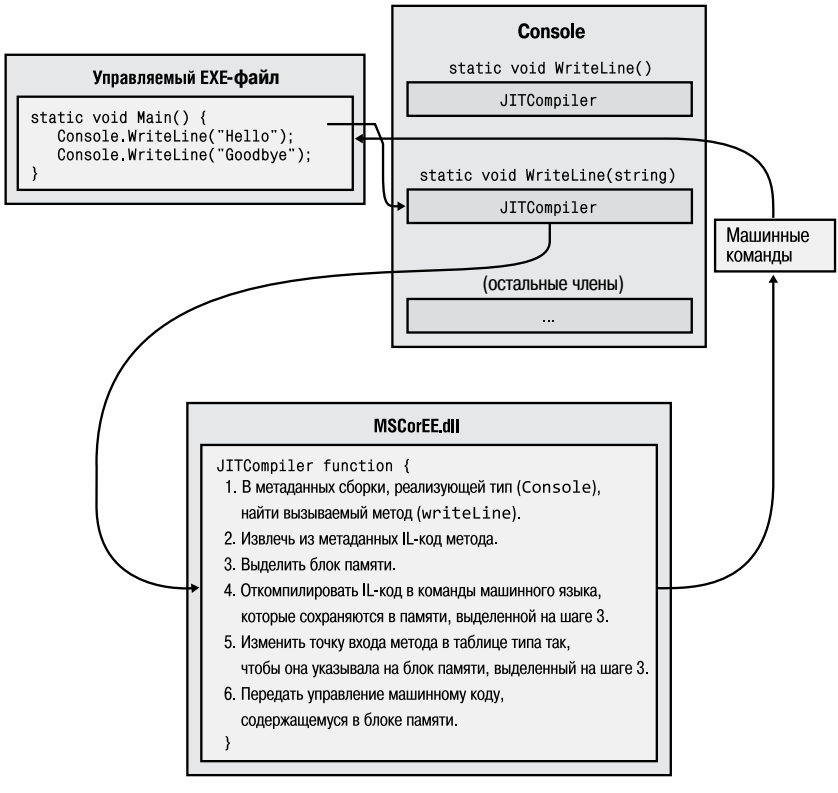

Рис. 1.4. Первый вызов метода

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

ПРИМЕЧАНИЕ

Если приложение исполняется в x86 версии Windows или в режиме WoW64, JIT-компилятор генерирует команды для архитектуры x86. Для приложений, выполняемых как 64-разрядные в версии x64 Windows, JIT-компилятор генерирует команды x64. Наконец, если приложение выполняется в ARM-версии Windows, JIT-компилятор генерирует инструкции ARM.