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

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

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

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

Задачи развертывания в .NET Framework

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

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

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

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

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

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

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

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

Компоновка типов в модуль

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Файл параметров

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

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

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

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

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

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

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

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

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

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

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

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

ПРИМЕЧАНИЕ

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

Рабочий каталог.

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

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

Все каталоги, указанные с использованием параметра /lib компилятора.

Все каталоги, указанные в переменной окружения LIB.

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

Несколько слов о метаданных

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

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

Таблица 2.1. Общие таблицы определений, входящих в метаданные

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

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

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

Таблица 2.2. Общие таблицы ссылок, входящие в метаданные

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

продолжение ⇩

Таблица 2.2 *(продолжение)*

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

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

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

```
ILDasm Program.exe
```

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

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

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

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

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

TypeRef #2 (01000002)
---------------------------
Token:            0x01000002
ResolutionScope:  0x23000001
```

продолжение ⇩