第3章

托管执行环境

计算机本身只能理解0、1的二进制信息，所以高级编程语言需要被编译成二进制代码才能由计算机执行。那么，.NET Framework程序是如何被计算机理解和运行的呢？本章将详细介绍托管代码的编译和执行原理。

3.1 概　　述

图3.1简单描述了.NET Framework中托管代码编译和执行的流程。C#和VB.NET的代码首先被编译为微软中间语言(Microsoft Intermediate Language，MSIL)并存储在本地。当需要运行这些托管代码时，CLR又对MSIL进行第二次编译(JIT编译)，将MSIL代码转换成本机代码(本机代码是针对当前CPU的可执行二进制代码)。

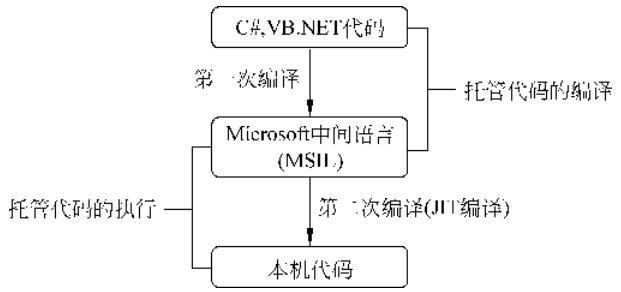

图3.1　编译和执行

从高级语言编译为公共中间语言的概念是现代编译器技术的主要特征，早在Visual Studio 6.0中，编译器就已经能够将各种编程语言转换为相同的中间语言，然后使用公共后端将其编译为二进制代码。这种二进制代码针对特定的CPU，只能在兼容的CPU上运行。在.NET Framework出现之前，这些二进制代码就是安装在用户计算机上的程序。

对于.NET Framework应用程序，安装在用户计算机上的程序并不是针对特定的CPU二进制代码，而是此应用程序的MSIL，即一种类似于以前隐藏在编译器内部的中间语言的代码。为何要进行这种变化，将代码作为MSIL进行分发有何好处？

很明显，这么做可以实现可移植性。.NET Framework的核心并不需要在特定的操作系统和CPU上运行，也就是说它可以在非Windows操作系统和非Intel兼容处理器的系统

上运行。但是，客观地说，.NET Framework 基本上仍然是侧重于 Windows 的技术，因此可移植性并不是 MSIL 的主要目的。

可移植性不是使用中间语言的唯一好处。二进制代码可以引用任意内存地址，而 MSIL 代码在加载到内存中时将受到类型安全性方面的检验，使某些类型的错误以及大量可能的攻击将变为不可能，这实现了更好的安全性和更高的可靠性。

使用中间语言可能会导致代码变慢，但由于 MSIL 总是在执行之前编译而不是解释，因此这在大多数情况下不会成为问题。

3.2　编译托管代码

编译基于 CLR(公共语言运行库)的语言编写的源代码时，生成以 MSIL 表达的指令和元数据(关于“MSIL 指令及其所操作的数据”的一些信息)。无论这些代码一开始是用 C#、VB 还是其他基于 CLR 的语言编写的，编译器都会将托管代码中的所有类型——类、结构、整数、委托及所有其他东西——统统转换成 MSIL 和元数据，该过程如图 3.2 所示。

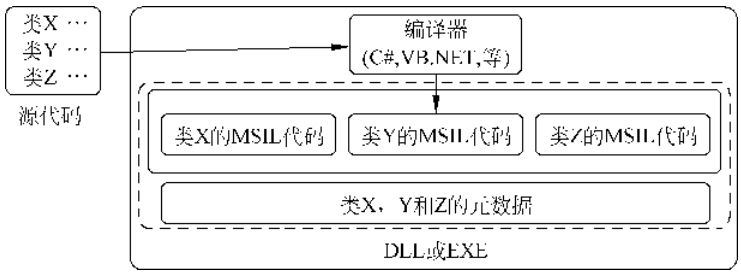

图 3.2　编译托管代码

图 3.2 中，正被编译的代码包含 3 个 CTS 类型，都是类。在使用适当的编译器编译这些代码之后，每一个类都对应于一套等价的 MSIL 代码，以及描述这些类的元数据。MSIL 和元数据都被存储于标准的 Windows 可移植可执行(Portable Executable，PE)文件中，可以是 DLL 文件或 EXE 文件，常见术语“模块”用于指代这些文件。以下将进一步讨论 MSIL 和元数据。

3.2.1　编译器选项

.NET Framework 包括一个 C# 的命令行编译器，名为 csc.exe。

1. 在 C# 中编译

要编译一个 HelloDemoCS.cs 应用程序的源代码，可以在 VS 的命令窗口中输入如下语句。

```
csc  HelloDemoCS.cs
```

这个命令行调用 C# 编译器。此处只需要指定要编译的文件名称，编译器就可以产生

可执行文件 HelloDemoCS.exe。

2. 命令行选项

在 C# 中，通过使用 /? 开关选项可以获得完整的命令行选项列表。

```
csc /?
```

命令行选项包括：/t 开关，指定编译目标；/r 开关，指定引用程序集；/out 开关，指定输出文件的名称。下面的命令行表示将 class1.cs 文件编译生成 class1.exe 文件。

```
csc  /out: c:\class1.exe        /t:exe          c:\class1.cs
     输出文件的全名称        生成文件的类型       要编译的源文件
```

其中 /t 开关中的各个选项及其含义如下：

- /t:exe　控制台应用。
- /t:winexe　Windows Form 应用。
- /t:library　包含清单的程序集。
- /t:module　不包含清单的独立程序集。

3.2.2 Microsoft 中间语言

MSIL 与处理器的本机指令集非常相似，但是，并不存在实际执行这些指令的任何硬件；相反，MSIL 代码在执行之前总是转换为相应的本机代码。客观地说，在 .NET Framework 环境中进行工作的开发人员并不需要完全理解 MSIL，但是，至少要稍微了解一下 MSIL 的内容。

实际上，MSIL 是 CLR 的汇编语言。关于这些 MSIL 指令集，值得注意的一点是，它与 CLR 的 CTS 抽象有着十分紧密的对应关系。它能直接支持对象、值类型，甚至装箱和拆箱操作；而且，某些操作（如创建新实例）与高级语言中的一些极为常见的操作符相似，而这些操作方法在典型机器指令中很少见。

对于希望直接处理那些低级机器码的开发人员，.NET Framework 提供了被称为 Ilasm 的 MSIL 汇编器。不过，开发人员一般不会去使用此工具，既然可以使用更简单、更强大的语言能够获得相同的结果，也就不必用 MSIL 编写了。

3.2.3 元数据

元数据是一种二进制信息，用以对存储在公共语言运行库的可移植可执行（PE）文件或存储在内存中的程序进行描述。将代码编译为 PE 文件时，便会将元数据插入到该文件的一部分中，而将代码转换为 Microsoft 中间语言（MSIL）并将其插入到该文件的另一部分中。在模块或程序集中定义和引用的每个类型和成员都将在元数据中进行说明。执行代码时，运行库将元数据加载到内存中，并引用它来发现有关代码的类、成员、继承等信息。

元数据包含以下信息：

- 类型的名称。

- 类型的可见性，可为 public 或 assembly。
- 基类。
- 所实现的接口。
- 所实现的方法。
- 所暴露的属性。
- 所提供的事件。

此外，还可以包括更多的细节信息。例如每一个方法的描述，包括方法的参数及其类型、方法的返回值类型等。

Visual Studio 2005 使用元数据提供智能感知(IntelliSense)功能，可向开发人员显示对于所输入的类名有哪些相应的方法可用；还可以使用 MSIL 反汇编工具 Ildasm 检查模块的元数据，该工具是 3.2.2 节提到的 Ilasm 工具的反工具，可提供包含在特定模块中的元数据的详细显示。

3.2.4　属性

属性(attribute)是存储在元数据中的值，可以使用属性来全面控制代码的执行方式，它与 System. Attribute 类相对应。属性(attribute)可添加到类型(如类)以及这些类型的字段、方法和属性(property)中，属性(attribute)是描述代码的代码，而属性(property)是描述对象的代码。.NET Framework 类库依赖属性(attribute)完成很多事情，包括指定事务要求、指示哪些方法应公开为 SOAP 可调用的 Web 服务，以及描述安全性要求。这些属性(attribute)有标准名称和功能，是由使用这些属性(attribute)的 .NET Framework 类库的各个部分所定义的。

开发人员也可以创建自定义的属性(attribute)，以一种与应用程序相关的方式来控制行为。为了创建定制的特性，使用基于 CLR 的编程语言(如 C#)时，可以定义一个继承于 System. Attribute 的类。一旦被编译，该类的一个实例便会自动将其值存储于元数据中。

3.3　组织托管代码：程序集

一个完整的应用程序通常由很多不同的文件组成，某些文件是模块(如 DLL 文件或 EXE 文件)，其中包含代码，另一些有可能包含诸如图形之类的种种资源。在 .NET Framework 应用程序中，构成一个“功能上的逻辑单元”的文件被聚合到一个程序集之中。程序集是开发、部署、运行 .NET Framework 应用程序的基础成分。

3.3.1　程序集的元数据：清单

程序集是一个逻辑上的构件，并不存在单一文件将所有必要文件包裹成一个程序集。实际上，只看磁盘目录列表，不可能说出哪些文件属于同一个程序集。要想搞清楚一个特定的程序集到底由哪些文件构成，必须查看该程序集的清单。如前所述，模块的特性包含了模块内的类型信息，与此相类似，程序集清单包含了“程序集内的所有模块及其他文件”的有关

信息，换句话说，清单就是程序集的特性，清单包含于程序集的某个文件里，并且包含了程序集的信息，以及“组成程序集的文件”的信息。Visual Studio 这样的工具不但会为每个编译模块生成元数据，也会为每个程序集生成相应的清单。

一个程序集可由单一文件或一组文件构成。图 3.3 显示了只包含单个文件的程序集的构成，程序集的元数据、类型元数据、MSIL 代码和资源都在一个文件 App1.dll 中。图 3.4 显示了包含多个文件的程序集的构成，这个程序集中共包含了 4 个文件，APP1.dll 包含了程序集的元数据、类型元数据和 MSIL 代码，但不包含资源；这个程序集使用了一个图像 Picture.jpeg，该图像没有嵌入在 App1.dll 中，而是在程序集的元数据中引用；程序集的元数据还引用了一个模块 Module，该模块只包含一个类的类型元数据和 MSIL 代码，不包含程序集的元数据，所以这个模块没有版本信息，也不能单独安装。这 3 个文件构成了一个程序集，这个程序集是一个安装单元；此外，在该程序集中还包含了另外一个文件放置程序集清单。

图 3.3　包含单个文件的程序集

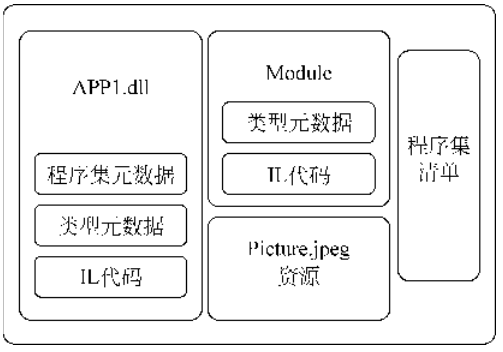

图 3.4　包含多个文件的程序集

对于一个“单文件程序集”，清单存储于文件自身；对于一个“多文件程序集”，清单存储在程序集的某个文件里。不论哪一种情况，清单描述整个程序集，而每一个模块里的元数据则详细描述了该模块内的类型。程序集清单包含以下一些信息。

- 程序集名称，所有程序集都有一个文本名称，以及一个可有可无的“强名称”。
- 程序集的版本号，号码形式为：＜主版本号＞.＜次版本号＞.＜内部版本号＞.＜修订号＞。例如，某个发行的应用程序里一个程序集的版本号有可能是 1.2.1397.0。注意，并不是每个模块有一个版本号，而是每个程序集有一个版本号。
- 程序集的文化背景，指明一个程序集所支持的文化或语言。
- 程序集包含的所有文件的列表，以及依据这些文件计算出来的一个散列(Hash)值。
- 程序集所依赖的其他程序集，以及每一个被依赖的程序集的版本号。

大多数程序集只包含单一 DLL。不过，无论它包含一个文件还是多个文件，程序集总是一个“逻辑上不可分割的单元”，例如，程序集定义了一个边界，用来为类型界定范围。对 CLR 而言，一个类型的完整名字其实是由该名称及其所处程序集的名称构成的。

从程序集的构成方式可以得出一个重要推论：CLR 类不同于 COM 类，并没有相关联的注册表入口(除非出于向后兼容的考虑而同时作为 COM 类来访问)。当 CLR 需要在另外某个程序集里寻找一个类时，它并不到注册表中去寻找，而是根据一个明确定义的算法去搜索。

“不需注册表入口”带来的另一个好处是：安装一个典型的程序集只需简单地将构成它

的那些文件复制到目标计算机磁盘上的适当路径即可。同样的道理，简单删除其构成的文件，就可以完成对一个程序集的卸载工作。和基于 COM 的应用程序不同，建造于 .NET Framework 之上的软件无需修改系统注册表。

3.3.2　程序集的分类

程序集的分类有多种方式。可以将程序集分为静态程序集和动态程序集：静态程序集由诸如 Visual Studio 这样的工具生成，其内容被存储于磁盘上，大多数开发人员都会生成静态程序集，因为他们的目标往往是构建一个“可被安装于一台或多台机器上”运行的应用程序；动态程序集的代码（以及元数据）直接在内存中生成，而后可立即运行，一旦创建完毕，动态程序集便可保存到磁盘上，日后可再次加载并运行，动态程序集的最常见例子是 ASP.NET 处理 .aspx 页时由 ASP.NET 所创建的程序集。

程序集也可以依照其命名来分类。所有程序集都有一个简单的文本名称（如 AccountAccess），也可以有一个“强名称”。任何程序集的完整命名都需要 3 个要素：程序集的名称、版本号和文化背景。强名称除了包含程序集名称通常的 3 个组成要素之外，还包括一个依据程序集计算而得到的数字签名，以及与“用于创建该数字签名”的私钥和其相对应的公钥。强名称是独一无二的，因此可用于毫无歧义地标识出某个特定程序集。如果有需要，也可以通过创建数字签名，将实体的证书嵌入到程序集内，这将允许任何使用程序集的人决定是否信任此实体，并愿意执行这个程序集。

程序集被加载时，CLR 会自动检查以强名称命名的程序集的版本号。对于无强名称的程序集，版本控制是创建和使用这些程序集的开发人员的责任。由于程序集拥有版本号，因此在同一时间内，同一台计算机上就有可能安装同一程序集的不同版本。

在 .NET 出现之前，安装应用程序的同时可能需要一并安装新版 DLL，这可能会对依赖于旧版 DLL 的其他应用程序造成破坏，这个问题被赋予了一个“饱含诗意”的名字：DLL 地狱。.NET Framework 的目标之一就是要解决这个问题，由于一个强名称程序集可以精确指出它所依赖的其他任何程序集的版本，基于 CLR 的应用程序就可以坚持它所依赖的每个程序集的特定版本，而不强求其他应用程序也得使用这个新版本，这样就最大程度地减小了过去因 DLL 冲突所导致的问题。不过开发人员仍需小心，否则仍有可能发生冲突。例如，某个程序集的两个版本都写入同一个临时文件，除非它们对如何共享这个文件达成一致意见，否则两个版本同时运行时将会出问题。

3.4　执行托管代码

程序集提供了一种“将模块打包到相应单元以便部署”的方法。然而编写代码的目标并不是为了打包和部署，而是为了运行。运行托管代码包括装载程序集、编译 MSIL 两个步骤。

3.4.1　装载程序集

要运行一个以 .NET Framework 创建的应用程序，首先必须找到构成这个应用程序的

程序集，并将其载入内存。非必要时，程序集不会被载入内存，因此如果应用程序从来不调用某个程序集里的任何方法，那么该程序集就不会被装载。然而在加载某个程序集的任何代码之前，首先必须找到这个程序集，那么查找过程究竟如何呢？

答案并不简单。事实上 CLR 用于查找程序集的过程太复杂了，以至于无法完整描述，不过这个过程的概念还是相当直白的。如图 3.5 所示，首先 CLR 确定它要找的是哪个程序集的哪个版本。默认情况下，它将从发起调用的地方查找“程序集清单内指定”的程序集精确版本，可通过设置各种配置文件来修改这个默认值，因此着手查找之前，CLR 首先检查这些配置文件。

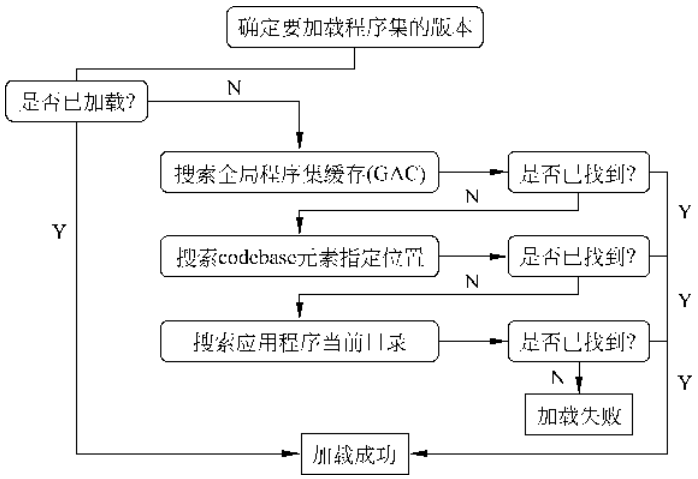

图 3.5　程序集搜索顺序

一旦精确决定了所需版本，CLR 就检查所需的程序集是否已被加载。如果是，搜索过程结束，已装载的版本将被使用；如果所需的程序集尚未被装载，CLR 将搜索多个地方以找到它。CLR 搜索的第一个地方通常是全局程序集缓存(Global Assembly Cache，GAC)，这是一个特殊的磁盘目录，用于容纳“被不止一个应用程序所使用”的程序集，把程序集安装于 GAC，所需操作只比简单复制程序集略微复杂一些，这个缓存区内只能放置拥有“强名称”的程序集。

如果 CLR 的搜索目标不在 GAC，它将继续搜索工作，检查应用程序的某个配置文件内的 codebase 元素。如果找到一个 codebase 元素，CLR 便查找该元素所指的位置，如果找到了正确的程序集，就意味着搜索过程结束了，而且该程序集将被加载使用；即使 codebase 元素所指位置上并未包含搜索目标(程序集)，搜索过程仍会结束。codebase 元素意味着精确指定了“在哪儿可以找到这个程序集”，如果程序集不在那个位置，肯定存在一些问题，CLR 也将放弃查找，那么装载程序集的操作也就失败了。

如果没有 codebase 元素，CLR 便开始在被称为 application base 的地方查找需要的程序集。所谓 application base 既可以是应用程序所在的根目录，也可以是个 URL，有可能在另外某台计算机上。如果这个难以寻觅的程序集在这里还找不到，CLR 会按照程序集的名字、文化背景等线索，在另外几个目录中查找，此处省略不提。

3.4.2　编译 MSIL

将 MSIL 编译成本机代码最常见的方式是让 CLR 加载程序集，然后在首次调用方法时编译其中对应的方法。由于每个方法是在首次被调用时编译的，因此该过程称为即时(JIT)编译。

图 3.6～图 3.8 演示了程序集中的代码如何进行 JIT 编译。这个简单的示例只显示了 3 个类，分别称为 X、Y 和 Z，每个类都包含若干个方法，每个方法对应一个模块。在图 3.6 中，只有 Y 类的方法 1 已得到编译，3 个类的所有其他方法中的所有其他代码仍为 MSIL，即它们加载时所使用的格式。当 Y 类的方法 1 调用 Z 类的方法 1 时，CLR 会注意到这个新调用的方法不是可执行格式，CLR 将调用 JIT 编译器编译 Z 类的方法 1，并将对该方法的调用重定向到该方法编译后的本机代码，该方法即可执行。

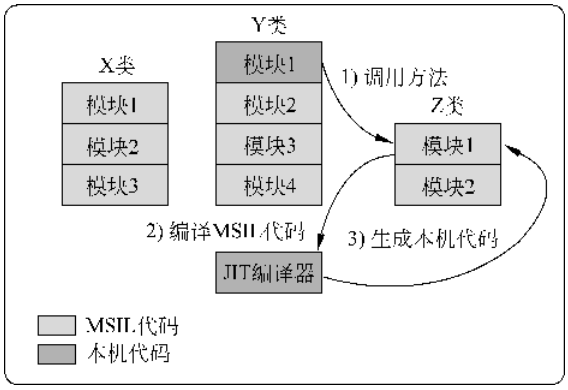

图 3.6　首次调用 Z 类的方法 1 时

图 3.7 中的情况类似，Y 类的方法 1 调用其自己的方法 4。如前所述，此方法仍为 MSIL，因此将自动调用 JIT 编译器编译该方法。与前面一样，对该方法的 MSIL 代码的引用将替换为对新创建的本机代码的引用，然后执行该方法。

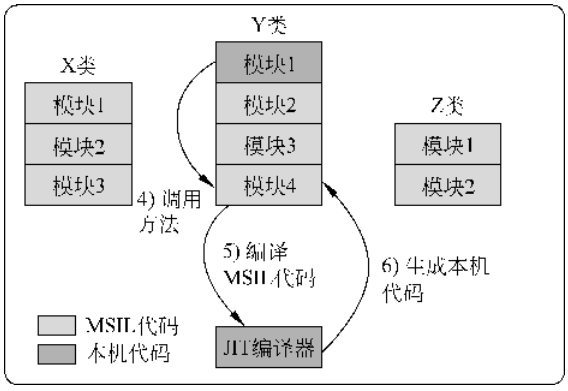

图 3.7　调用 Y 类的方法 4 时

图 3.8 说明了当类 Y 的方法 1 再次调用 Z 类中的方法 1 时所发生的情况。此方法已经经过 JIT 编译,因此无需再进行任何其他工作。本机代码已经保存在内存中,因此可直接执行,而不会调用 JIT 编译器。整个过程将以同样的方式继续,每个方法在首次被调用时进行编译。

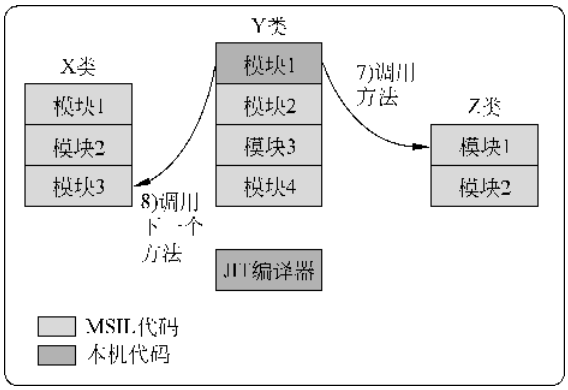

图 3.8 再次调用 Z 类的方法 1 时

在对方法进行 JIT 编译时,编译器还将检查其类型安全性,这个过程称为验证。验证过程检查方法的 MSIL 和元数据,以确保代码不会进行非法访问。CLR 内置的安全性功能依赖于验证,托管代码行为的其他方面也是如此。但是,系统管理员可以禁用不需要的验证,这意味着 CLR 可执行非类型安全的托管代码,这可能会很有用,因为某些编译器不能生成类型安全的代码。但是,一般而言,对于 .NET Framework 应用程序应尽可能使用验证。

如果使用 JIT 编译,则只有那些被调用的方法才会进行编译。如果程序集中的方法已加载,但是从未使用过,则它将仍保持其 MSIL 形式。

注意:编译后的本机代码不会存回到磁盘上,每次加载程序集时都要执行 JIT 编译过程。

3.4.3 垃圾回收

托管堆在 .NET Framework 应用程序的执行过程中扮演着重要的角色。引用类型(如每个类、每个字符串等)的每个实例都是在堆上分配的,当有应用程序不断地运行时,分配到堆的内存可能会被填满,因此在可以创建新的实例之前,必须使更多空间可用。使空间可用的过程称为垃圾回收,将释放未使用的对象。

当 CLR 注意到堆已填满时,它将自动运行垃圾回收器(应用程序也可以显式请求运行垃圾回收器,但是这并不是特别常见的做法)。若要了解垃圾回收的工作方式,请再次思考引用类型的分配方式。每个引用类型在堆栈上都有一个指向其在堆中的实际值的条目,如图 3.9 所示,堆栈包含十进制数值 32.4,一个引用代表字符串"Hello",整数值 14,以及一个引用代表“被装箱的整数值 169”,两个引用类型(字符串和被装箱的整数值)其值都存储在堆上。

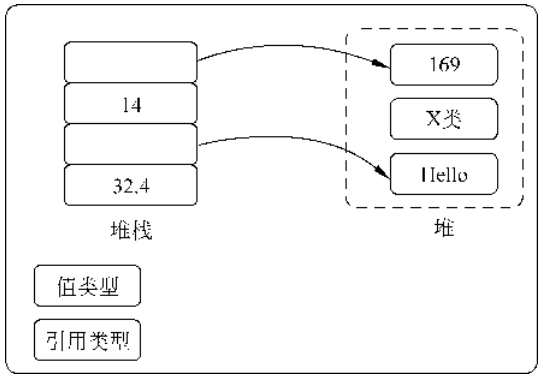

图 3.9　堆上由 X 类的对象所占据的空间为垃圾

但是应该注意，堆中还包含着 X 类的对象的信息。图 3.9 并不是按比例绘制的，所以此对象占据的空间可能比字符串或装箱整数多得多；也许此对象是由已经完成执行的方法创建的，因此从堆栈指向它的引用现在已经消失。不管怎样，此对象正占据着可做它用的空间，换言之，它就是垃圾。

当垃圾回收器运行时，它将扫描堆以查找此类垃圾。在获知堆的哪些部分为垃圾之后，它将重新安排堆的内容，将那些仍在使用的值更紧密地压缩在一起。例如，在垃圾回收后，源于 X 类的对象的垃圾已经消失，它以前占据的空间可重用，以存储其他仍在使用的信息，如图 3.10 所示。

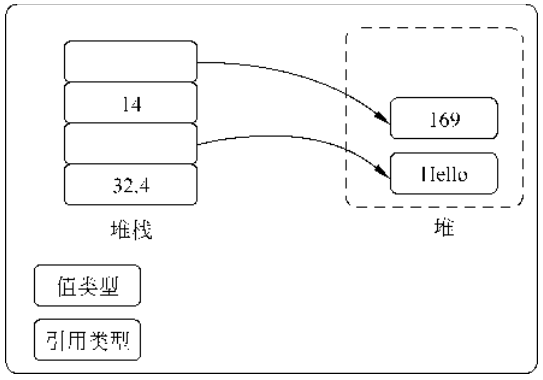

图 3.10　垃圾回收

如此例所示，长期存在的对象随着时间的推移被移至堆的某一端。在实际应用中，最新分配的对象最快变成垃圾的情况相当常见，在查找垃圾时，可以先查找堆中最新分配的对象。CLR 的垃圾回收器也正是如此，它先检查这些最新一代的对象，然后回收由垃圾占据的任何空间；如果这一轮操作后仍未能释放出足够的内存，则垃圾回收器将检查前一代对象，也就是稍早一些分配的那些对象；如果仍然未能释放足够的空间以满足当前需求，则回收器将检查托管堆中的所有剩余对象，释放不再使用的任何空间。

堆上的每个对象都有一个被称为终结器的特殊方法，但是，默认情况下，此方法不执行任何操作。如果类型需要在销毁之前执行某些最终清理操作，则创建该类型的开发人员可

重写该类型的默认终结器。在释放带有终结器的对象之前，该对象将置于终结列表中，最终，此列表中的每个对象都将调用对应的终结器。

3.4.4 应用程序域

CLR 是以 DLL 形式实现的，这使得 CLR 以一种常规的方式使用它。当然这也意味着还需要提供一个 EXE 来作为 CLR 的宿主。运行库宿主可提供此功能：加载并初始化 CLR，然后通常将控制权转移给托管代码。ASP.NET 与 SQL Server 2005、Internet Explorer 及其他应用程序一样都提供了运行库宿主，Windows 外壳程序也可充当运行库宿主，用于加载使用托管代码的独立可执行文件。

运行库宿主在其进程内创建一个或多个应用程序域（Application Domain，也称为应用域），每个进程包含一个默认应用程序域，并且每个程序集都被加载到特定应用程序域中。应用程序域类似于传统的操作系统进程，将它所包含的应用程序与所有其他应用域中的应用程序相隔离，但是，因为多个应用域可共存于同一个进程之中，所以这些域之间的通信会更加有效。

但是如何确保有效地隔离应用域呢？如果没有操作系统所提供的对进程的内置支持，怎样保证同一进程的两个不同应用域中运行的应用程序不会相互干扰呢？答案仍是“验证”。因为托管代码在进行 JIT 编译时将进行类型安全检查，所以系统可以肯定没有程序集能够直接访问其各自边界之外的任何内容。

可以有多种方法使用应用域。例如，ASP.NET 在其自己的应用域内运行各个 Web 应用程序，这将使应用程序相互保持隔离，但是不会导致运行多个不同进程的开销；运行库宿主 Internet Explorer 可从 Internet 上下载 Windows 窗体控件，然后在每个控件各自的应用域内运行该控件，这样做的好处是既保证了隔离以及隔离所带来的安全性，同时又不会有由跨进程通信所引起的开销。此外，由于应用程序可在同一进程的不同应用域中单独启动和停止，因此此方法还避免了为每个应用程序启动新进程的开销。

图 3.11 就说明了这种情况。应用域 1（默认应用域）包括程序集 A、B 和 C，程序集 D 和 E 已加载到应用域 2 中，而程序集 F 在应用域 3 中运行。即使所有这些程序集都在一个进程中运行，每个应用域的程序集仍完全独立于其他应用域中的程序集。

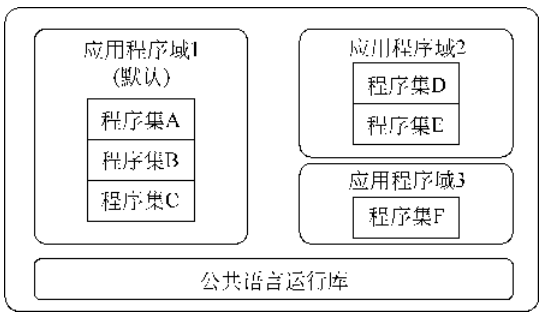

图 3.11 应用程序域