CS0009:未能打开元数据文件“c:\MultiFileLibrary.dll”–“导入程序集 “c:\MultiFileLibrary.dll”的模块“RUT.netmodule”时出错 – 系统找不到指定的文件。 ”



> 注意 以下内容供技术宅参考。元数据 token 是一个 4 字节的值。其中，高位字节指明 token 的类型 (0x01=TypeRef， 0x02=TypeDef，0x23=AssemblyRef，0x26=File(文件定义)，0x27=ExportedType)。要获取完整列表，请参见.NET Framework SDK 包含的 CorHdr.h 文件中的 CorTokenType 枚举类型。token 的三个低位字节指明对应的元数据表中的行。例如，0x26000001 这个实现 token 引用的是 File 表的第一行。大多数表的行从 1 而不是 0 开始编号。TypeDef 表的行号实际从 2 开始。

这意味着为了生成新程序集，所引用的程序集中的所有文件都必须存在。

客户端代码执行时会调用方法。一个方法首次调用时，CLR 检测作为参数、返回值或者局部变量而被方法引用的类型。然后，CLR 尝试加载所引用程序集中含有清单的文件。如果要访问的类型恰好在这个文件中，CLR 会执行其内部登记工作，允许使用该类型。如果清单指出被引用的类型在不同的文件中，CLR 会尝试加载需要的文件，同样执行内部登记，并允许使用该类型。注意，CLR 并非一上来就加载所有可能用到的程序集。只有在调用的方法确实引用了未加载程序集中的类型时，才会加载程序集。换言之，为了让应用程序运行起来，并不要求被引用程序集的所有文件都存在。

2.4.1 使用 Visual Studio IDE 将程序集添加到项目中

用 Visual Studio IDE 创建项目时，想引用的所有程序集都必须添加到项目中。为此，请打开解决方案资源管理器，右击想添加引用的项目，选择“添加引用”来打开“引用管理器”对话框，如图 2-2 所示。

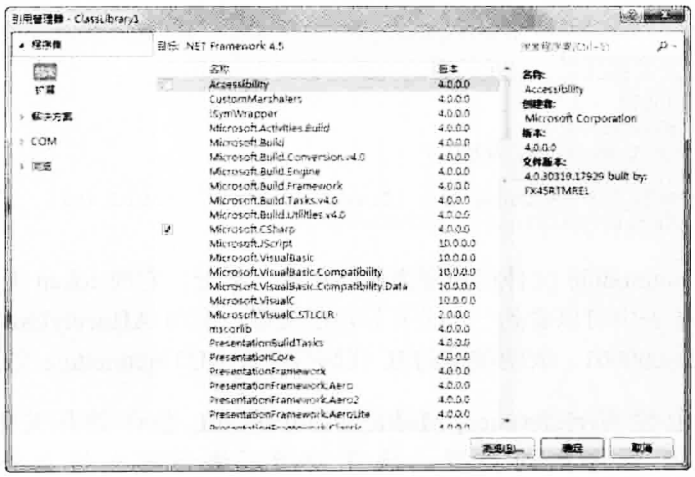


图 2-2 Visual Studio 的引用管理器

从列表中选择想让项目引用的程序集。如果程序集不在列表中，就单击“浏览”按钮，选择目标程序集(含清单的文件)并添加程序集引用。利用“解决方案”选项，当前项目可以引用同一个解决方案中的另一个项目创建的程序集。“COM”选项允许从托管源代码中访问一个非托管 COM 服务器，这是通过 Visual Studio 自动生成的一个托管代理类实现的。利用“项目”选项，可以选择最近添加到其他项目的程序集。

按照 *http://msdn.microsoft.com/en-us/library/wkze6zky(v=vs.110).aspx* 的指令进行操作，可以使自己的程序集出现在“引用管理器”中。

2.4.2　使用程序集链接器

除了使用 C#编译器，还可以使用“程序集链接器”实用程序 AL.exe 来创建程序集。如果程序集要包含由不同编译器生成的模块(而且这些编译器不支持与 C#编译器的**/addmodule** 开关等价的机制)，或者在生成时不清楚程序集的打包要求，程序集链接器就显得相当有用。还可用 AL.exe 生成只含资源的程序集，也就是所谓的**附属程序集**(satellite assembly)，它们通常用于本地化。本章稍后会讨论附属程序集的问题。

AL.exe 实用程序能生成 EXE 文件，或者生成只包含清单(对其他模块中的类型进行描述)的 DLL PE 文件。为了理解 AL.exe 的工作原理，让我们改变一下 MultiFileLibrary.dll 程序集的生成方式：

```
csc /t:module RUT.cs
csc /t:module FUT.cs
al /out:MultiFileLibrary.dll /t:library FUT.netmodule RUT.netmodule
```

图 2-3 展示了执行这些命令后生成的文件。

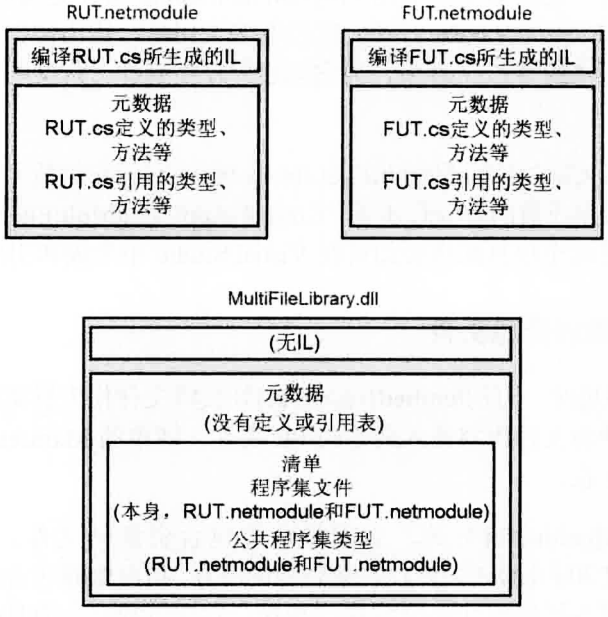


图 2-3　由三个托管模块构成的多文件程序集，其中一个含有清单

这个例子首先创建两个单独的模块，即 RUT.netmodule 和 FUT.netmodule。两个模块都不是程序集，因为都不包含清单元数据表。然后生成第三个文件 MultiFileLibrary.dll，它是 DLL PE 文件(因为使用了**/t[arget]:library** 开关)，其中不包含 IL 代码，但包含清单元数据表。清单元数据表指出 RUT.netmodule 和 FUT.netmodule 是程序集的一部分。最终的程序集由三个文件构成：MultiFileLibrary.dll，RUT.netmodule 和 FUT.netmodule。程序集链接器不能将多个文件合并成一个文件。

使用**/t[arget]:exe**，**/t[arget]:winexe** 或者**/t[arget]:appcontainerexe** 命令行开关，AL.exe 实用程序还可生成 CUI，GUI 或者 Windows Store 应用 PE 文件。但很少需要这样做，因为这意味着在得到的 EXE PE 文件中，IL 代码唯一做的事情就是调用另一个模块中的方法。调用 AL.exe 时添加**/main** 命令行开关，可指定模块的哪个方法是入口。例如：

```
csc /t:module /r:MultiFileLibrary.dll Program.cs
al /out:Program.exe /t:exe /main:Program.Main Program.netmodule
```

第一行将 Program.cs 文件生成为 Program.netmodule 文件。第二行生成包含清单元数据表的 Program.exe PE 文件。此外，由于使用了**/main:Program.Main** 命令行开关，AL.exe 还会生成一个小的全局函数，名为**__EntryPoint**，其中包含以下 IL 代码：

```
.method privatescope static void __EntryPoint$PST06000001() cil managed
{
    .entrypoint
    // Code size     8 (0x8)
    .maxstack 8
    IL_0000: tail.
    IL_0002: call    void [.module 'Program.netmodule']Program::Main()
    IL_0007: ret
} // end of method 'Global Functions'::__EntryPoint
```

可以看出，上述代码只是调用了一下在 Program.netmodule 文件定义的 **Program** 类型中包含的 **Main** 方法。AL.exe 的**/main** 开关实际没有多大用处，因为假如一个应用程序的入口不在清单元数据表所在的 PE 文件中，为它创建程序集有什么意义呢？开发人员只需知道有这个开关就可以了。

本书配套代码有一个 Ch02-3-BuildMultiFileLibrary.bat 文件，它封装了生成多文件程序集所需的全部步骤。作为生成前的命令行步骤，Ch02-4-AppUsingMultiFileLibrary 项目会调用该批处理文件。可参考这个项目来体会如何在 Visual Studio 中生成和引用多文件程序集。

2.4.3 为程序集添加资源文件

用 AL.exe 创建程序集时，可用**/embed[resource]**开关将文件作为资源添加到程序集。该开关获取任意文件，并将文件内容嵌入最终的 PE 文件。清单的 ManifestResourceDef 表会更新以反映新资源的存在。

AL.exe 还支持**/link[resource]**开关，它同样获取包含资源的文件，但只是更新清单的 ManifestResourceDef 和 FileDef 表以反映新资源的存在，指出资源包含在程序集的哪个文件中。资源文件不会嵌入程序集 PE 文件中；相反，它保持独立，而且必须和其他程序集文件一起打包和部署。

与 AL.exe 相似，C#编译器 CSC.exe 也允许将资源合并到编译器生成的程序集中。**/resource** 开关将指定的资源文件嵌入最终生成的程序集 PE 文件中，并更新 ManifestResourceDef 表。**/linkresource** 开关在 ManifestResourceDef 和 FileDef 清单表中添加记录项来引用独立存在的资源文件。

关于资源，最后注意可在程序集中嵌入标准的 Win32 资源。为此，只需在使用 AL.exe 或者 CSC.exe 时使用**/win32res** 开关指定一个.res 文件的路径名。还可在使用 AL.exe 或者 CSC.exe 时使用**/win32icon** 开关指定一个.ico 文件的路径名，从而在程序集中快速、简单地嵌入标准的 Win32 图标资源。要在 Visual Studio 中将资源文件添加到程序集中，可以显示项目的属性，在“应用程序”选项卡中添加资源文件。嵌入图标的目的一般是在 Windows 资源管理器中为托管的可执行文件显示特色图标。



注意　托管的程序集文件还包含 Win32 清单资源信息。C#编译器默认生成这种清单信息，但可用**/nowin32manifest** 开关告诉它不生成。C#编译器生成的默认清单是下面这样的：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
    <assemblyIdentity version="1.0.0.0" name="MyApplication.app" />
        <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
            <security>
                <requestedPrivileges xmlns="urn:schemas-microsoft-
                com:asm.v3">
                    <requestedExecutionLevel level="asInvoker" uiAccess=
                    "false"/>
                </requestedPrivileges>
            </security>
        </trustInfo>
</assembly>
```

2.5　程序集版本资源信息

AL.exe 或 CSC.exe 生成 PE 文件程序集时，还会在 PE 文件中嵌入标准的 Win32 版本资源。可查看文件属性来检查该资源。在应用程序代码中调用 **System.Diagnostics.FileVersionInfo** 的静态方法 **GetVersionInfo**，并传递程序集的路径作为参数，就可以获取并检查这些信息。图 2-4 显示的是 Ch02-3-MultiFileLibrary.dll 属性对话框的“详细信息”选项卡。

生成程序集时，应该使用定制特性设置各种版本资源字段，这些特性在源代码中应用于 assembly 级别。图 2-4 的版本信息用以下代码生成：

```
using System.Reflection;

// FileDescription 版本信息
[assembly: AssemblyTitle("MultiFileLibrary.dll")]

// Comments 版本信息
[assembly: AssemblyDescription("This assembly contains MultiFileLibrary's types")]

// CompanyName 版本信息
```

```
[assembly: AssemblyCompany("Wintellect")]

// ProductName 版本信息
[assembly: AssemblyProduct("Wintellect (R) MultiFileLibrary's Type Library")]

// LegalCopyright 版本信息
[assembly: AssemblyCopyright("Copyright (c) Wintellect 2013")]

// LegalTrademarks 版本信息
[assembly:AssemblyTrademark("MultiFileLibrary is a registered trademark of Wintellect")]

// AssemblyVersion 版本信息
[assembly: AssemblyVersion("3.0.0.0")]

// FILEVERSION/FileVersion 版本信息
[assembly: AssemblyFileVersion("1.0.0.0")]

// PRODUCTVERSION/ProductVersion 版本信息
[assembly: AssemblyInformationalVersion("2.0.0.0")]

// 设置 Language 字段(参见 2.6 节“语言文化”)
[assembly:AssemblyCulture("")]
```



图 2-4 “Ch02-3-MultiFileLibrary.dll 属性”对话框的“详细信息”选项卡



重要提示 Windows 资源管理器的属性对话框明显遗漏了一些特性值。最遗憾的是没有显示 **AssemblyVersion** 这个特性的值，因为 CLR 加载程序集时会使用这个值，详情将在第 3 章讨论。

表 2-4 总结了版本资源字段和对应的定制特性。如果用 AL.exe 生成程序集，可以用命令行开关设置这些信息，而不必使用定制特性。表 2-4 的第二列显示了与每个版本资源字段对应的 AL.exe 命令行开关。注意，C#编译器没有提供这些命令行开关。在这种情况下，最好是用定制特性设置这些信息。

表 2-4　版本资源字段和对应的 AL.exe 开关/定制特性

版本资源	AL.exe 开关	定制特性/说明
FILEVERSION	**/fileversion**	**System.Reflection.AssemblyFileVersionAttribute**
PRODUCTVERSION	**/productversion**	**System.Reflection.AssemblyInformationalVersionAttribute**
FILEFLAGSMASK	(无)	总是设为 **VS_FFI_FILEFLAGSMASK** (在 WinVer.h 中定义为 **0x0000003F**)
FILEFLAGS	(无)	总是 0
FILEOS	(无)	目前总是 **VOS__WINDOWS32**
FILETYPE	**/target**	如果指定了**/target:exe** 或**/target:winexe**，就设为 **VFT_APP**；如果指定了**/target:library**，就设为 **VFT_DLL**
FILESUBTYPE	(无)	总是设为 **VFT2_UNKNOWN**(该字段对于 **VFT_APP** 和 **VFT_DLL** 无意义)
AssemblyVersion	**/version**	**System.Reflection.AssemblyVersionAttribute**
Comments	**/description**	**System.Reflection.AssemblyDescriptionAttribute**
CompanyName	**/company**	**System.Reflection.AssemblyCompanyAttribute**
FileDescription	**/title**	**System.Reflection.AssemblyTitleAttribute**
FileVersion	**/version**	**System.Reflection.AssemblyFileVersionAttribute**
InternalName	**/out**	设为指定的输出文件的名称(无扩展名)
LegalCopyright	**/copyright**	**System.Reflection.AssemblyCopyrightAttribute**
LegalTrademarks	**/trademark**	**System.Reflection.AssemblyTrademarkAttribute**
OriginalFilename	**/out**	设为输出文件的名称(无路径)
PrivateBuild	(无)	总是空白
ProductName	**/product**	**System.Reflection.AssemblyProductAttribute**
ProductVersion	**/productversion**	**System.Reflection.AssemblyInformationalVersionAttribute**
SpecialBuild	(无)	总是空白



重要提示　Visual Studio 新建 C#项目时会在一个 Properties 文件夹中自动创建 AssemblyInfo.cs 文件。该文件除了包含本节描述的所有程序集版本特性，还包含要在第 3 章讨论的几个特性。可直接打开 AssemblyInfo.cs 文件并修改自己的程序集特有信息。Visual Studio 还提供了对话框来帮你编辑该文件。要打开这个对话框，请打开项目的属性页，在“应用程序”选项卡中单击“程序集信息”。随后会看到如图 2-5 所示的对话框。



图 2-5　Visual Studio 的“程序集信息”对话框

版本号

上一节指出可向程序集应用几个版本号。所有这些版本号都具有相同的格式，每个都包含 4 个以句点分隔的部分，如表 2-5 所示。

表 2-5 版本号格式[①]

	major(主版本号)	minor(次版本号)	build(内部版本号)	revision(修订号)
示例	2	5	719	2

表 2-5 展示了一个示例版本号：2.5.719.2。前两个编号构成了公众对版本的理解。公众会将这个例子看成是程序集的 2.5 版本。第三个编号 719 是程序集的 build 号。如果公司每天都生成程序集，那么每天都应该递增这个 build 号。最后一个编号 2 指出当前 build 的修订次数。如果因为某个原因，公司某一天必须生成两次程序集(可能是为了修复一个造成其他什么事情都干不了的 hot bug)，revision 号就应该递增。Microsoft 采用的就是这个版本编号方案，强烈建议你也采用。

注意：程序集有三个版本号，这使局面复杂化，并造成大量混淆。所以，有必要解释一下每个版本号的用途以及它们的正确用法。

- **AssemblyFileVersion**
  这个版本号存储在 Win32 版本资源中。它仅供参考，CLR 既不检查，也不关心这个版本号。通常，可以先设置好版本号的 major/minor 部分，这是希望公众看到的版本号。然后，每生成一次就递增 build 和 revision 部分。理想情况是 Microsoft 的工具(比如 CSC.exe 或者 AL.ex)能自动更新 build 和 revision 号(根据生成时的日期和时间)。但实情并非如此。在 Windows 资源管理器中能看到这个版本号。对客户系统进行故障诊断时，可根据它识别程序集的版本是多少。

- **AssemblyInformationalVersion**
  这个版本号也存储在 Win32 版本资源中，同样仅供参考。CLR 既不检查，也不关心它。这个版本号的作用是指出包含该程序集的产品的版本。例如，产品的 2.0 版本可能包含几个程序集，其中一个程序集标记为版本 1.0，因为它是新开发的，在产品的 1.0 版本中不存在。通常，可以设置这个版本号的 major 和 minor 部分来代表产品的公开版本号。以后每次打包所有程序集来生成完整产品，就递增 build 和 revision 部分。

- **AssemblyVersion**
  这个版本号存储在 AssemblyDef 清单元数据表中。CLR 在绑定到强命名程序集(第 3 章讨论)时会用到它。这个版本号很重要，它唯一性地标识了程序集。开始开发程序集时，应该设置好 major/minor/build/revision 部分。而且除非要开发程序集的下一个可部

① 根据习惯，本书保留了版本号 4 个组成部分的英文原文，即 major，minor，build，revision。——译注

署版本，否则不应变动。如果程序集 A 引用了强命名的程序集 B，程序集 B 的版本会嵌入程序集 A 的 AssemblyRef 表。这样一来，当 CLR 需要加载程序集 B 时，就准确地知道当初生成和测试的是程序集 B 的哪个版本。利用第 3 章将要讨论的绑定重定向(binding redirect)技术，可以让 CLR 加载一个不同的版本。

2.6　语言文化

除了版本号，程序集还将**语言文化**(culture)[①]作为其身份标识的一部分。例如，可能有一个程序集限定德语用户，第二个限定瑞士德语用户，第三个限定美国英语用户，以此类推。语言文化用包含主副标记的字符串进行标识(依据 RFC1766)。表 2-6 展示了一些例子。

表 2-6　程序集语言文化标记的例子

主标记	副标记	语言文化
de	(无)	德语
de	AT	奥地利德语
de	CH	瑞士德语
en	(无)	英语
en	GB	英国英语
en	US	美国英语

创建含代码的程序集时一般不指定具体的语言文化。这是因为代码只讲“逻辑”，不涉及具体的语言文化。未指定具体语言文化的程序集称为**语言文化中性**(culture neutral)。

如果应用程序包含语言文化特有的资源，Microsoft 强烈建议专门创建一个程序集来包含代码和应用程序的默认(或备用)资源。生成该程序集时不要指定具体的语言文化。其他程序集通过引用该程序集来创建和操纵它公开的类型。

然后，创建一个或多个单独的程序集，只在其中包含语言文化特有的资源——不要包含任何代码。标记了语言文化的程序集称为**附属程序集**(satellite assembly)。为附属程序集指定的语言文化应准确反映程序集中的资源的语言文化。针对想要支持的每种语言文化都要创建单独的附属程序集。

通常用 AL.exe 生成附属程序集。不用编译器是因为附属程序集本来就不该含有代码。使用 AL.exe 的**/c[ulture]:text** 开关指定语言文化。其中 **text** 是语言文化字符串，例如“en-US”代表美国英语。部署附属程序集时，应该把它保存到专门的子目录中，子目录名称和语言文化的文本匹配。例如，假定应用程序的基目录是 C:\MyApp，与美国英语对应的附属程序集就应该放到 C:\MyApp\en-US 子目录。在运行时，使用 **System.Resources.ResourceManager** 类访问附属程序集的资源。

① 文档翻译为“区域性”。——译注



> **注意** 虽然不建议，但创建包含代码的附属程序集也是可以的。如果愿意，可以使用定制特性 *System.Reflection.AssemblyCultureAttribute* 代替 AL.exe 的/culture 开关指定语言文化。例如：
>
> ```
> // 将程序集的语言文化设为瑞士德语
> [assembly:AssemblyCulture("de-CH")]
> ```

一般不要生成引用了附属程序集的程序集。换言之，程序集的 AssemblyRef 记录项只应引用语言文化中性的程序集。要访问附属程序集中的类型或成员，应使用第 23 章“程序集加载和反射”介绍的反射技术。

2.7 简单应用程序部署(私有部署的程序集)

前面解释了如何生成模块，如何将模块合并为程序集。接着要解释如何打包和部署程序集，使用户能运行应用程序。

Windows Store 应用对程序集的打包有一套很严格的规则，Visual Studio 会将应用程序所有必要的程序集打包成一个.appx 文件。该文件要么上传到 Windows Store，要么 side-load[①]到机器。用户安装.appx 文件时，其中包含的所有程序集都进入一个目录。CLR 从该目录加载程序集，Windows 则在用户的“开始”屏幕添加应用程序磁贴。如果其他用户安装相同的.appx 文件，程序集会使用之前安装好的，新用户只是在“开始”屏幕添加了一个磁贴。用户卸载 Windows Store 应用时，系统从“开始”屏幕删除磁贴。如果没有其他用户安装该应用，Windows 删除目录和其中的所有程序集。注意，不同用户可以安装同一个 Windows Store 应用的不同版本。为此，Windows 将程序集安装到不同的目录，使一个应用可以多版本并存。

对于非 Windows Store 的桌面应用，程序集的打包方式没有任何特殊要求。打包一组程序集最简单的方式就是直接复制所有文件。例如，可将所有程序集文件放到一张光盘上，将光盘分发给用户，执行上面的一个批处理程序，将光盘上的文件复制到用户硬盘上的一个目录。由于已经包含了所有依赖的程序集和类型，所以用户能直接运行应用程序，“运行时”会在应用程序目录查找引用的程序集。不需要对注册表进行任何修改就能运行程序。要卸载应用程序，删除所有文件就可以了——就是那么简单！

也可使用其他机制打包和安装程序集文件，比如使用.cab 文件(从 Internet 下载时使用，旨在压缩文件并缩短下载时间)。还可将程序集文件打包成一个 MSI 文件，以便由 Windows Installer 服务(MSIExec.exe)使用。使用 MSI 文件可实现程序集的“按需安装”——CLR 首次尝试加载一个程序集时才安装它。这不是 MSI 的新功能；非托管 EXE 和 DLL 文件也能这么加载。

---

① 不经应用商店而将软件拷贝到设备上，就称为 side-load。——译注



注意　使用批处理程序或其他简单的“安装软件”，足以将应用程序“弄”到用户的机器上。但要在用户桌面和“开始”菜单上创建快捷方式，仍需使用一款较高级的安装软件。除此之外，可以方便地备份和还原应用程序，或者在机器之间移动，但快捷方式仍需特殊处理。

当然，也可使用 Visual Studio 内建的机制发布应用程序。具体做法是打开项目属性页并点击“发布”标签。利用其中的选项，可以让 Visual Studio 生成 MSI 文件并将它复制到网站、FTP 服务器或者文件路径。这个 MSI 文件还能安装必备组件，比如.NET Framework 或 Microsoft SQL Server Express Edition。最后，利用 ClickOnce 技术，应用程序还能自动检查更新，并在用户的机器上安装更新。

在应用程序基目录或者子目录部署的程序集称为**私有部署的程序集**(privately deployed assembly)，这是因为程序集文件不和其他任何应用程序共享(除非其他应用程序也部署到该目录)。私有部署的程序集为开发人员、最终用户和管理员带来了许多便利，因为只需把它们复制到一个应用程序的基目录，CLR 便会加载它们并执行其中的代码。除此之外，要卸载应用程序，从目录中删除程序集即可。这使备份和还原也变得简单了。

之所以能实现这种简单的安装/移动/卸载，是因为每个程序集都用元数据注明了自己引用的程序集，不需要注册表设置。另外，引用(别的程序集的)程序集限定了每个类型的作用域。也就是说，一个应用程序总是和它生成和测试时的类型绑定。即便另一个程序集恰好提供了同名类型，CLR 也不可能加载那个程序集。这一点有别于 COM。在 COM 中，类型是在注册表中登记的，造成机器上运行的任何应用程序都能使用那些类型。

第 3 章将讨论如何部署可由多个应用程序访问的共享程序集。

2.8　简单管理控制(配置)

用户或管理员经常需要控制应用程序的执行。例如，管理员可能决定移动用户硬盘上的程序集文件，或者覆写程序集清单中的信息。还有一些情形涉及版本控制，第 3 章将进一步讨论。

为了实现对应用程序的管理控制，可在应用程序目录放入一个配置文件。应用程序的发布者可创建并打包该文件。安装程序会将配置文件安装到应用程序的基目录。另外，计算机管理员或最终用户也能创建或修改该文件。CLR 会解析文件内容来更改程序集文件的定位和加载策略。

配置文件包含 XML 代码，它既能和应用程序关联，也能和机器关联。由于使用的是一个单独的文件(而不是注册表设置)，用户可以方便地备份文件，管理员也能将应用程序方便地复制到其他机器——只要把必要的文件复制过去，管理策略就会被复制过去。

第 3 章将更详细探讨这个配置文件，目前只需对它有一个基本的认识。例如，假定应用程序的发布者想把 MultiFileLibrary 的程序集文件部署到和应用程序的程序集文件不同的目录，要求目录结构如下：

```
AppDir 目录(包含应用程序的程序集文件)
    Program.exe
    Program.exe.config(在下面讨论)

    AuxFiles 子目录(包含 MultiFileLibrary 的程序集文件)
        MultiFileLibrary.dll
        FUT.netmodule
        RUT.netmodule
```

由于 MultiFileLibrary 的文件不在应用程序的基目录，所以 CLR 无法定位并加载这些文件。运行程序将抛出 **System.IO.FileNotFoundException** 异常。为了解决问题，发布者创建了 XML 格式的配置文件，把它部署到应用程序的基目录。文件名必须是应用程序主程序集文件的名称，附加.config 扩展名，也就是 Program.exe.config。配置文件内容如下：

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

CLR 尝试定位程序集文件时，总是先在应用程序基目录查找。如果没有找到，就查找 AuxFiles 子目录。可为 **probing** 元素的 **privatePath** 特性指定多个以分号分隔的路径。每个路径都相对于应用程序基目录。不能用绝对或相对路径指定在应用程序基目录外部的目录。这个设计的出发点是应用程序能控制它的目录及其子目录，但不能控制其他目录。

这个 XML 配置文件的名称和位置取决于应用程序的类型。

- 对于可执行应用程序(EXE)，配置文件必须在应用程序的基目录，而且必须采用 EXE 文件全名作为文件名，再附加.config 扩展名。
- 对于 Microsoft ASP.NET Web 窗体应用程序，文件必须在 Web 应用程序的虚拟根目录中，而且总是命名为 Web.config。除此之外，子目录可以包含自己的 Web.config，而且配置设置会得到继承。例如，位于 *http://Wintellect.com/Training* 的 Web 应用程序既会使用虚拟根目录的 Web.config 设置，也会使用 Training 子目录的。

本节开头说过，配置设置既可应用于程序，也可应用于机器。.NET Framework 在安装时会创建一个 Machine.config。机器上安装的每个版本的 CLR 都有一个对应的 Machine.config。

Machine.config 文件在以下目录中：

```
%SystemRoot%\Microsoft.NET\Framework\version\CONFIG
```

其中，%SystemRoot%是 Windows 目录(一般是 C:\WINDOWS)，*version* 是.NET Framework 的版本号(形如 v4.0.#####)。

探测程序集文件

CLR 定位程序集时会扫描几个子目录。以下是加载一个语言文化中性的程序集时的目录探测顺序(其中，firstPrivatePath 和 secondPrivatePath 通过配置文件的 privatePath 特性指定):

```
AppDir\AsmName.dll
AppDir\AsmName\AsmName.dll
AppDir\firstPrivatePath\AsmName.dll
AppDir\firstPrivatePath\AsmName\AsmName.dll
AppDir\secondPrivatePath\AsmName.dll
AppDir\secondPrivatePath\AsmName\AsmName.dll
...
```

在这个例子中，如果 MultiFileLibrary 程序集的文件部署到 MultiFileLibrary 子目录，就不需要配置文件，因为 CLR 能自动扫描与目标程序集名称相符的子目录。

如果在上述任何子目录都找不到目标程序集，CLR 会从头再来，用.exe 扩展名替换.dll 扩展名。再找不到就抛出 **FileNotFoundException** 异常。

附属程序集(satellite assembly)遵循类似的规则，只是 CLR 会在应用程序基目录下的一个子目录中查找，子目录名称与语言文化相符。例如，假定向 AsmName.dll 应用了“en-US”语言文化，那么会探测以下子目录:

```
C:\AppDir\en-US\AsmName.dll
C:\AppDir\en-US\AsmName\AsmName.dll
C:\AppDir\firstPrivatePath\en-US\AsmName.dll
C:\AppDir\firstPrivatePath\en-US\AsmName\AsmName.dll
C:\AppDir\secondPrivatePath\en-US\AsmName.dll
C:\AppDir\secondPrivatePath\en-US\AsmName\AsmName.dll

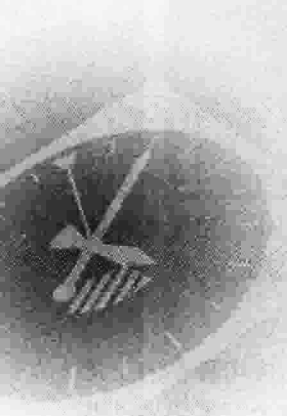
C:\AppDir\en-US\AsmName.exe
C:\AppDir\en-US\AsmName\AsmName.exe
C:\AppDir\firstPrivatePath\en-US\AsmName.exe
C:\AppDir\firstPrivatePath\en-US\AsmName\AsmName.exe
C:\AppDir\secondPrivatePath\en-US\AsmName.exe
C:\AppDir\secondPrivatePath\en-US\AsmName\AsmName.exe

C:\AppDir\en\AsmName.dll
C:\AppDir\en\AsmName\AsmName.dll
C:\AppDir\firstPrivatePath\en\AsmName.dll
C:\AppDir\firstPrivatePath\en\AsmName\AsmName.dll
C:\AppDir\secondPrivatePath\en\AsmName.dll
C:\AppDir\secondPrivatePath\en\AsmName\AsmName.dll

C:\AppDir\en\AsmName.exe
C:\AppDir\en\AsmName\AsmName.exe
C:\AppDir\firstPrivatePath\en\AsmName.exe
C:\AppDir\firstPrivatePath\en\AsmName\AsmName.exe
C:\AppDir\secondPrivatePath\en\AsmName.exe
C:\AppDir\secondPrivatePath\en\AsmName\AsmName.exe
```

如你所见，CLR 会探测具有.exe 或.dll 扩展名的文件。由于探测可能很耗时(尤其是 CLR 需要通过网络查找文件的时候)，所以最好在 XML 配置文件中指定一个或多个 culture 元素，限制 CLR 查找附属程序集时的探测动作。Microsoft 提供了 FusLogVw.exe 工具来帮助你了解 CLR 在运行时与程序集的绑定。访问 *http://msdn.microsoft.com/en-us/ibrary/e74a18c4(v=vs.110).aspx* 了解详情。

Machine.config 文件的设置是机器上运行的所有应用程序的默认设置。所以，管理员为了创建适用于整个机器的策略，修改一个文件即可。然而，管理员和用户一般应该避免修改该文件，因为该文件的许多设置都有着太多的牵连，使我们难免顾此失彼。另外，我们经常都要对应用程序的设置进行备份和还原，只有将这些设置保存到应用程序专用的配置文件，才能方便地做到这一点。



第 3 章　共享程序集和强命名程序集

本章内容：

- 两种程序集，两种部署
- 为程序集分配强名称
- 全局程序集缓存
- 在生成的程序集中引用强命名程序集
- 强命名程序集能防篡改
- 延迟签名
- 私有部署强命名程序集
- “运行时”如何解析类型引用
- 高级管理控制(配置)

第 2 章讲述了生成、打包和部署程序集的步骤。我将重点放在所谓的私有部署(private deployment)上。进行私有部署，程序集放在应用程序的基目录(或子目录)，由这个应用程序独享。以私有方式部署程序集，可以对程序集的命名、版本和行为进行最全面的控制。

本章重点是如何创建可由多个应用程序共享的程序集。Microsoft .NET Framework 随带的程序集就是典型的全局部署程序集，因为所有托管应用程序都要使用 Microsoft 在.NET Framework Class Library(FCL)中定义的类型。

第 2 章讲过，Windows 以前在稳定性上的口碑很差，主要原因是应用程序要用别人实现的代码进行生成和测试。(想想看，你开发的 Windows 应用程序是不是要调用由 Microsoft 开发人员写好的代码？)另外，许多公司都开发了供别人嵌入的控件。事实上，.NET Framework 鼓励这样做，以后的控件开发商会越来越多。

随着时间的推移，Microsoft 开发人员和控件开发人员会修改代码，这或许是为了修复 bug、进行安全更新、添加功能等。最终，新代码会进入用户机器。以前安装好的、正常工作的应用程序突然要面对“陌生”的代码，不再是应用程序最初生成和测试时的代码。因此，应用程序的行为不再是可以预测的，这是造成 Windows 不稳定的根源。

文件的版本控制是个难题。取得其他代码文件正在使用的一个文件，即使只修改其中一位(将 0 变成 1，或者将 1 变成 0)，就无法保证使用该文件的代码还能正常工作。使用文件的新版本时，道理是一样的。之所以这样说，是因为许多应用程序都有意或无意地利用了 bug。如果文件的新版本修复了 bug，应用程序就不能像预期的那样运行了。

所以现在的问题是：如何在修复 bug 并添加新功能的同时，保证不会中断应用程序的正常运行？我对这个问题进行过大量思考，最后结论是完全不可能！但是，这个答案明显不够好。分发的文件总是有 bug，公司总是希望推陈出新。必须有一种方式在分发新文件的同时，尽量保证应用程序良好工作。如果应用程序不能良好工作，必须有一种简单的方式将应用程序恢复到上一次已知良好的状态。

本章将解释.NET Framework 为了解决版本控制问题而建立的基础结构。事先说一句：要讲述的内容比较复杂。将讨论 CLR 集成的大量算法、规则和策略。还要提到应用程序开发人员必须熟练使用的大量工具和实用程序。之所以复杂，是因为如前所述，版本控制本来就是一个复杂的问题。

3.1 两种程序集，两种部署

CLR 支持两种程序集：**弱命名程序集**(weakly named assembly)和**强命名程序集**(strongly named assembly)。



重要提示 任何文档都找不到“弱命名程序集”这个术语，这是我自创的。事实上，文档中没有对应的术语来表示弱命名的程序集。通过自造术语，我在提到不同种类的程序集时可以避免歧义。

弱命名和强命名程序集结构完全相同。也就是说，它们都使用第 1 章和第 2 章讨论的 PE 文件格式、PE32(+)头、CLR 头、元数据、清单表以及 IL。生成工具也相同，都是 C#编译器或者 AL.exe。两者真正的区别在于，强命名程序集使用发布者的公钥/私钥进行了签名。这一对密钥允许对程序集进行唯一性的标识、保护和版本控制，并允许程序集部署到用户机器的任何地方，甚至可以部署到 Internet 上。由于程序集被唯一性地标识，所以当应用程序绑定到强命名程序集时，CLR 可以应用一些已知安全的策略。本章将解释什么是强命名程序集，以及 CLR 向其应用的策略。

程序集可采用两种方式部署：私有或全局。私有部署的程序集是指部署到应用程序基目录或者某个子目录的程序集。弱命名程序集只能以私有方式部署。第 2 章已讨论了私有部署的程序集。全局部署的程序集是指部署到一些公认位置的程序集。CLR 在查找程序集时，会检查这些位置。强命名程序集既可私有部署，也可全局部署。本章将解释如何创建和部署强命名程序集。表 3-1 总结了程序集的种类及其部署方式。

表 3-1　弱命名和强命名程序集的部署方式

程序集种类	可以私有部署	可以全局部署
弱命名	是	否
强命名	是	是

3.2　为程序集分配强名称

要由多个应用程序访问的程序集必须放到公认的目录。另外，检测到对程序集的引用时，CLR 必须能自动检查该目录。但现在的问题是：两个(或更多)公司可能生成具有相同文件名的程序集。所以，假如两个程序集都复制到相同的公认目录，最后一个安装的就是“老大”，造成正在使用旧程序集的所有应用程序都无法正常工作(这正是 Windows“DLL hell”的由来，因为共享 DLL 全都复制到 System32 目录)。

只根据文件名来区分程序集明显不够。CLR 必须支持对程序集进行唯一性标识的机制。这就是所谓的“强命名程序集”。强命名程序集具有 4 个重要特性，它们共同对程序集进行唯一性标识：文件名(不计扩展名)、版本号、语言文化和公钥。由于公钥数字很大，所以经常使用从公钥派生的小哈希值，称为**公钥标记**(public key token)。以下程序集标识字符串(有时称为**程序集显示名称**)标识了 4 个完全不同的程序集文件：

```
"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

"MyTypes, Version=1.0.8123.0, Culture="en-US", PublicKeyToken=b77a5c561934e089"

"MyTypes, Version=2.0.1234.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
```

第一个标识的是程序集文件 MyTypes.exe 或 MyTypes.dll (无法根据“程序集标识字符串”判断文件扩展名)。生成该程序集的公司为其分配的版本号是 1.0.8123.0，而且程序集中没有任何内容与一种特定语言文化关联，因为 **Culture** 设为 **neutral**。当然，任何公司都可以生成 MyTypes.dll(或 MyTypes.exe)程序集文件，为其分配相同的版本号 1.0.8123.0，并将语言文化设为中性。

所以，必须有一种方式区分恰好具有相同特性的两个公司的程序集。出于几方面的考虑，Microsoft 选择的是标准的公钥/私钥加密技术，而没有选择其他唯一性标识技术，如 GUID(Globally Unique Identifier，全局唯一标识符)、URL(Uniform Resource Locator，统一资源定位符)和 URN(Uniform Resource Name，统一资源名称)。具体地说，使用加密技术，不仅能在程序集安装到一台机器上时检查其二进制数据的完整性，还允许每个发布者授予一套不同的权限。本章稍后会讨论这些技术。所以，一个公司要想唯一性地标识自己的程序集，必须创建一对公钥/私钥。然后，公钥可以和程序集关联。没有任何两家公司有相同的公钥/私钥对。这样一来，两家公司可以创建具有相同名称、版本和语言文化的程序集，同时不会产生任何冲突。



注意 可利用辅助类 **System.Reflection.AssemblyName** 轻松构造程序集名称，并获取程序集名称的各个组成部分。该类提供了几个公共实例属性，比如 **CultureInfo**，**FullName**，**KeyPair**，**Name** 和 **Version**。还提供了几个公共实例方法，比如 **GetPublicKey**，**GetPublicKeyToken**，**SetPublicKey** 和 **SetPublicKeyToken**。

第 2 章介绍了如何命名程序集文件，以及如何应用程序集版本号和语言文化。弱命名程序集可在清单元数据中嵌入程序集版本和语言文化；然而，CLR 通过探测子目录查找附属程序集(satellite assembly)时，会忽略版本号，只用语言文化信息。由于弱命名程序集总是私有部署，所以 CLR 在应用程序基目录或子目录(具体子目录由 XML 配置文件的 **probing** 元素的 **privatePath** 特性指定)中搜索程序集文件时只使用程序集名称(添加.dll 或.exe 扩展名)。

强命名程序集除了有文件名、程序集版本号和语言文化，还用发布者私钥进行了签名。

创建强命名程序集第一步是用.NET Framework SDK 和 Microsoft Visual Studio 随带的 Strong Name 实用程序(SN.exe)获取密钥。SN.exe 允许通过多个命令行开关来使用一整套功能。注意所有命令行开关都区分大小写。为了生成公钥/私钥对，像下面这样运行 SN.exe：

```
SN -k MyCompany.snk
```

这告诉 SN.exe 创建 MyCompany.snk 文件。文件中包含二进制形式的公钥和私钥。

公钥数字很大；如果愿意，创建.snk 文件后可再次使用 SN.exe 查看实际公钥。这需要执行两次 SN.exe。第一次用**-p** 开关创建只含公钥的文件(MyCompany.PublicKey)：[①]

```
SN -p MyCompany.snk MyCompany.PublicKey sha256
```

第二次用**-tp** 开关执行，传递只含公钥的文件：

```
SN -tp MyCompany.PublicKey
```

执行上述命令，在我的机器上得到的输出如下：

```
Microsoft(R) .NET Framework 强名称实用工具 版本 4.0.30319.17929
版权所有(C) Microsoft Corporation。保留所有权利。

公钥(哈希算法: sha256):
002400000c800000940000000602000000240000525341310004000001000100e3e32597daeb0a
0d7acf5a6230a3f36d0c406571ae39b1feb5ec0bc93088145011dab48253762b44abf97838380c
29592c58af4eeecafe3dfe20f029e5b70a18d58214c40563fd7bec41c090c94931df579c2a6bb8
019303084444c2945eb488232ff4c43cee91f0977af0e1da2e1dce555803d518c6dfaf2f51c022
775ecf9d

公钥标记为 d4ef0f81895d6eed
```

① 本例使用.NET Framework 4.5 引入的增强型强命名(Enhanced Strong Naming)。要生成和以前版本的.NET Framework 兼容的程序集，还必须用 **AssemblySignatureKeyAttribute** 创建联署签名(counter-signature)。详情参见 *http://msdn.microsoft.com/en-us/library/hh415055(v=vs.110).aspx*。

注意，SN.exe 实用程序未提供任何显示私钥的途径。

公钥太大，难以使用。为了简化开发人员的工作(也为了方便最终用户)，人们设计了**公钥标记**(public key token)。公钥标记是公钥的 64 位哈希值。SN.exe 的**-tp** 开关在输出结果的末尾显示了与完整公钥对应的公钥标记。

知道了如何创建公钥/私钥对，创建强命名程序集就简单了。编译程序集时使用**/keyfile:**<*file*>编译器开关：

```
csc /keyfile:MyCompany.snk Program.cs
```

C#编译器看到这个开关会打开指定文件(MyCompany.snk)，用私钥对程序集进行签名，并将公钥嵌入清单。注意只能对含清单的程序集文件进行签名；程序集其他文件不能被显式签名。

要在 Visual Studio 中新建公钥/私钥文件，可显示项目属性，点击“签名”标签，勾选“为程序集签名”，然后从“选择强名称密钥文件”选择框中选择“<新建…>”。

“对文件进行签名”的准确含义是：生成强命名程序集时，程序集的 FileDef 清单元数据表列出构成程序集的所有文件。每将一个文件名添加到清单，都对文件内容进行哈希处理。哈希值和文件名一道存储到 FileDef 表中。要覆盖默认哈希算法，可使用 AL.exe 的**/algid** 开关，或者在程序集的某个源代码文件中，在 assembly 这一级上应用定制特性 **System.Reflection.AssemblyAlgorithmIdAttribute**。默认使用 SHA-1 算法。

生成包含清单的 PE 文件后，会对 PE 文件的完整内容(除去 Authenticode Signature、程序集强名称数据以及 PE 头校验和)进行哈希处理，如图 3-1 所示。哈希值用发布者的私钥进行签名，得到的 RSA 数字签名存储到 PE 文件的一个保留区域(进行哈希处理时，会忽略这个区域)。PE 文件的 CLR 头进行更新，反映数字签名在文件中的嵌入位置。

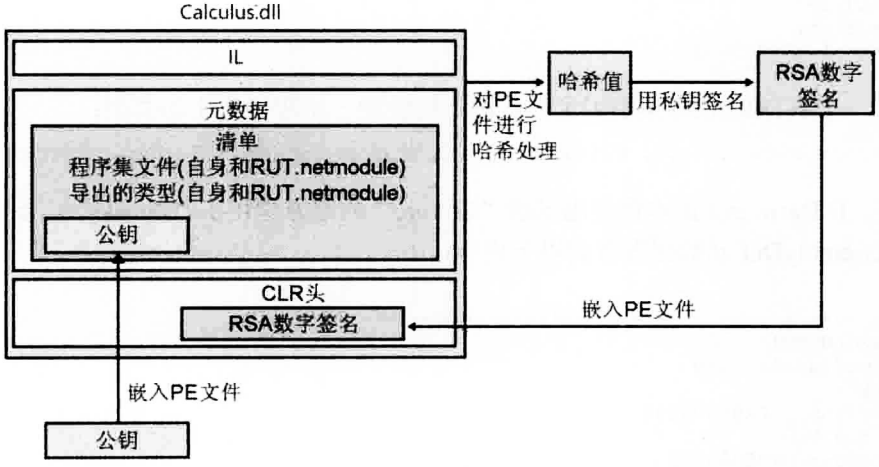


图 3-1　对程序集进行签名

发布者公钥也嵌入 PE 文件的 AssemblyDef 清单元数据表。文件名、程序集版本号、语言文化和公钥的组合为这个程序集赋予了一个强名称，它保证是唯一的。两家公司除非共享

密钥对，否则即使都生成了名为 OurLibrary 的程序集，公钥/私钥也不可能相同。

到此为止，程序集及其所有文件就可以打包和分发了。

如第 2 章所述，编译器在编译源代码时会检测引用的类型和成员。必须向编译器指定要引用的程序集——C#编译器是用**/reference** 编译器开关。编译器的一项工作是在最终的托管模块中生成 AssemblyRef 元数据表，其中每个记录项都指明被引用程序集的名称(无路径和扩展名)、版本号、语言文化和公钥信息。



重要提示 由于公钥是很大的数字，而一个程序集可能引用其他大量程序集，所以在最终生成的文件中，相当大一部分会被公钥信息占据。为了节省存储空间，Microsoft 对公钥进行哈希处理，并获取哈希值的最后 8 个字节。AssemblyRef 表实际存储的是这种简化的公钥值(称为“公钥标记”)。开发人员和最终用户一般看到的都是公钥标记，而不是完整公钥。

但要注意，CLR 在做出安全或信任决策时，永远都不会使用公钥标记，因为几个公钥可能在哈希处理之后得到同一个公钥标记。

下面是一个简单类库 DLL 文件的 AssemblyRef 元数据信息(使用 ILDasm.exe 获得)：

```
AssemblyRef #1 (23000001)
-------------------------------------------------------
Token: 0x23000001
Public Key or Token: b7 7a 5c 56 19 34 e0 89
Name: mscorlib
Version: 4.0.0.0
Major Version: 0x00000004
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
HashValue Blob:
Flags: [none] (00000000)
```

可以看出，这个 DLL 程序集引用了具有以下特性的一个程序集中的类型：

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

遗憾的是，ILDasm.exe 在本该使用术语“Culture”的地方使用了“Locale”。检查 DLL 程序集的 AssemblyDef 元数据表看到以下内容：

```
Assembly
-------------------------------------------------------
Token: 0x20000001
Name : SomeClassLibrary
Public Key :
Hash Algorithm : 0x00008004
Version: 3.0.0.0
Major Version: 0x00000003
Minor Version: 0x00000000
Build Number: 0x00000000
Revision Number: 0x00000000
Locale: <null>
Flags : [none] (00000000)
```

它等价于：

```
"SomeClassLibrary, Version=3.0.0.0, Culture=neutral, PublicKeyToken=null"
```

之所以没有公钥标记，是由于 DLL 程序集没有用公钥/私钥对进行签名，这使它成为弱命名程序集。如果用 SN.exe 创建密钥文件，再用**/keyfile** 编译器开关进行编译，最终的程序集就是经过签名的。使用 ILDasm.exe 查看新程序集的元数据，AssemblyDef 记录项就会在 Public Key 字段之后显示相应的字节，表明它是强命名程序集。顺便说一句，AssemblyDef 的记录项总是存储完整公钥，而不是公钥标记，这是为了保证文件没有被篡改。本章后面将解释强命名程序集如何防篡改。

3.3　全局程序集缓存

知道如何创建强命名程序集之后，接着学习如何部署它，以及 CLR 如何利用信息来定位并加载程序集。

由多个应用程序访问的程序集必须放到公认的目录，而且 CLR 在检测到对该程序集的引用时，必须知道检查该目录。这个公认位置就是**全局程序集缓存**(Global Assembly Cache，GAC)。GAC 的具体位置是一种实现细节，不同版本会有所变化。但是，一般能在以下目录发现它：

```
%SystemRoot%\Microsoft.NET\Assembly
```

GAC 目录是结构化的：其中包含许多子目录，子目录名称用算法生成。永远不要将程序集文件手动复制到 GAC 目录；相反，要用工具完成这项任务。工具知道 GAC 的内部结构，并知道如何生成正确的子目录名。

开发和测试时在 GAC 中安装强命名程序集最常用的工具是 GACUtil.exe。如果直接运行，不添加任何命令行参数，就会自动显示用法：

```
Microsoft (R) .NET Global Assembly Cache Utility.  Version 4.0.30319.17929
版权所有(C) Microsoft Corporation。保留所有权利

用法: Gacutil <命令> [ <选项> ]
命令:
  /i <assembly_path> [ /r <...> ] [ /f ]
    将某个程序集安装到全局程序集缓存中

  /il <assembly_path_list_file> [ /r <...> ] [ /f ]
    将一个或多个程序集安装到全局程序集缓存中

  /u <assembly_display_name> [ /r <...> ]
    将某个程序集从全局程序集缓存卸载

  /ul <assembly_display_name_list_file> [ /r <...> ]
    将一个或多个程序集从全局程序集缓存卸载

  /l [ <assembly_name> ]
    列出通过 <assembly_name> 筛选出的全局程序集缓存

  /lr [ <assembly_name> ]
    列出全局程序集缓存以及所有跟踪引用
```

```
/cdl
  删除下载缓存的内容

/ldl
  列出下载缓存的内容

/?
  显示详细帮助屏幕

选项:
/r <reference_scheme> <reference_id> <description>
  指定要安装(/i, /il)或卸载(/u, /ul)的跟踪引用

/f
  强制重新安装程序集

/nologo
  取消显示徽标版权标志

/silent
  取消显示所有输出
```

使用 GACUtil.exe 的/i 开关将程序集安装到 GAC，/u 开关从 GAC 卸载程序集。注意不能将弱命名程序集放到 GAC。向 GACUtil.exe 传递弱命名程序集的文件名会报错：**“将程序集添加到缓存失败：尝试安装没有强名称的程序集。”**



> **注意** GAC 默认只能由 Windows Administrators 用户组的成员操作。如果执行 GACUtil.exe 的用户不是该组的成员，GACUtil.exe 将无法安装或卸载程序集。

GACUtil.exe 的/i 开关方便开发人员在测试时使用。但如果是在生产环境中部署，建议安装或卸载程序集时除了指定/i 或/u 开关，还要指定/r 开关。/r 开关将程序集与 Windows 的安装与卸载引擎集成。简单地说，它告诉系统哪个应用程序需要程序集，并将应用程序与程序集绑定。



> **注意** 如果将强命名程序集打包到.cab 文件中，或者以其他方式进行压缩，程序集的文件首先必须解压成临时文件，然后才能使用 GACUtil.exe 将程序集文件安装到 GAC 中。安装好程序集的文件之后，临时文件可以删除。

.NET Framework 重分发包不随带提供 GACUtil.exe 工具。如果应用程序含有需要部署到 GAC 的程序集，应该使用 Windows Installer(MSI)，因为 MSI 是用户机器上肯定会安装，又能将程序集安装到 GAC 的工具。



重要提示　在 GAC 中全局部署是对程序集进行注册的一种形式，虽然这个过程对 Windows 注册表没有半点影响。将程序集安装到 GAC 破坏了我们想要达到的一个基本目标，即：简单地安装、备份、还原、移动和卸载应用程序。所以，建议尽量进行私有而不是全局部署。

为什么要在 GAC 中“注册”程序集？假定两家公司都生成了名为 OurLibrary 的程序集，两个程序集都由一个 OurLibrary.dll 文件构成。这两个文件显然不能存储到同一个目录，否则最后一个安装的会覆盖第一个，造成应用程序被破坏。相反，将程序集安装到 GAC，就会在%SystemRoot%\Microsoft.NET\Assembly 目录下创建专门的子目录，程序集文件会复制到其中一个子目录。

一般没人去检查 GAC 的子目录，所以 GAC 的结构对你来说并不重要。只要使用的工具和 CLR 知道这个结构就可以了。

3.4　在生成的程序集中引用强命名程序集

你生成的任何程序集都包含对其他强命名程序集的引用，这是因为 **System.Object** 在 MSCorLib.dll 中定义，后者就是强命名程序集。此外，程序集中还可引用由 Microsoft、第三方厂商或者你自己公司发布的其他强命名程序集。第 2 章介绍了如何使用 CSC.exe 的 **/reference** 编译器开关指定想引用的程序集文件名。如果文件名是完整路径，CSC.exe 会加载指定文件，并根据它的元数据生成程序集。如第 2 章所述，如果指定不含路径的文件名，CSC.exe 会尝试在以下目录查找程序集(按所列顺序)。

1. 工作目录。
2. CSC.exe 所在的目录，目录中还包含 CLR 的各种 DLL 文件。
3. 使用**/lib** 编译器开关指定的任何目录。
4. 使用 LIB 环境变量指定的任何目录。

所以，如果生成的程序集引用了 Microsoft 的 System.Drawing.dll，可在执行 CSC.exe 时使用**/reference:System.Drawing.dll** 开关。编译器会依次检查上述目录，并在 CSC.exe 自己所在的目录找到 System.Drawing.dll 文件(该目录还存储了与编译器对应的那个版本的 CLR 的各种支持 DLL)。虽然编译时会在这里寻找程序集，但运行时不会从这里加载程序集。

安装.NET Framework 时，实际会安装 Microsoft 的程序集文件的两套拷贝。一套安装到编译器/CLR 目录，另一套安装到 GAC 的子目录。编译器/CLR 目录中的文件方便你生成程序集，而 GAC 中的拷贝则方便在运行时加载。

CSC.exe 编译器之所以不在 GAC 中查找引用的程序集，是因为你必须知道程序集路径，而 GAC 的结构又没有正式公开。第二个方案是让 CSC.exe 允许你指定一个依然很长但相对比

较容易阅读的字符串，比如"System.Drawing, Version=v4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"。但这两个方案都不如在用户硬盘上安装两套一样的程序集文件。

除此之外，编译器/CLR 目录中的程序集不依赖机器。也就是说，这些程序集只包含元数据。由于编译时不需要 IL 代码，所以该目录不必同时包含程序集的 x86，x64 和 ARM 版本。GAC 中的程序集才同时包含元数据和 IL 代码，因为仅在运行时才需要代码。另外，由于代码可针对特定 CPU 架构进行优化，所以 GAC 允许存在一个程序集的多个拷贝。每个 CPU 架构都有一个专门的子目录来容纳这些拷贝。

3.5 强命名程序集能防篡改

用私钥对程序集进行签名，并将公钥和签名嵌入程序集，CLR 就可验证程序集未被修改或破坏。程序集安装到 GAC 时，系统对包含清单的那个文件的内容进行哈希处理，将哈希值与 PE 文件中嵌入的 RSA 数字签名进行比较(在用公钥解除了签名之后)。如果两个值完全一致，表明文件内容未被篡改。此外，系统还对程序集的其他文件的内容进行哈希处理，并将哈希值与清单文件的 FileDef 表中存储的哈希值进行比较。任何一个哈希值不匹配，表明程序集至少有一个文件被篡改，程序集将无法安装到 GAC。

应用程序需要绑定到程序集时，CLR 根据被引用程序集的属性(名称、版本、语言文化和公钥)在 GAC 中定位该程序集。找到被引用程序集，就返回包含它的子目录，并加载清单所在的文件。以这种方式查找程序集，可保证运行时加载的程序集和最初编译时生成的程序集来自同一个发布者，因为进行引用的程序集的 AssemblyRef 表中的公钥标记与被引用程序集的 AssemblyRef 表中的公钥匹配。如果被引用程序集不在 GAC 中，CLR 会查找应用程序的基目录，然后查找应用程序配置文件中标注的任何私有路径。然后，如果应用程序由 MSI 安装，CLR 要求 MSI 定位程序集。如果在任何位置都找不到程序集，那么绑定失败，抛出 **System.IO.FileNotFoundException** 异常。

如果强命名程序集文件从 GAC 之外的位置加载(通过应用程序的基目录，或者通过配置文件中的 **codeBase** 元素)，CLR 会在程序集加载后比较哈希值。也就是说，每次应用程序执行并加载程序集时，都会对文件进行哈希处理，以牺牲性能为代价，保证程序集文件内容没有被篡改。CLR 在运行时检测到不匹配的哈希值会抛出 **System.IO.FileLoadException** 异常。



注意 将强命名程序集安装到 GAC 时，系统会执行检查，确保包含清单的文件没有被篡改。这个检查仅在安装时执行一次。除此之外，为了增强性能，如果强命名程序集被完全信任，并加载到完全信任的 AppDomain 中，CLR 将不检查该程序集是否被篡改。相反，从非 GAC 的目录加载强命名程序集时，CLR 会校验程序集的清单文件，确保文件内容未被篡改，造成该文件每次加载都造成额外的性能开销。

3.6　延迟签名

本章前面讲过如何使用 SN.exe 工具生成公钥/私钥对。该工具生成密钥时会调用 Windows 提供的 Crypto API。密钥可存储到文件或其他存储设备中。例如，大公司(比如 Microsoft)会将自己的私钥保存到一个硬件设备中，再将硬件锁进保险库。公司只有少数人才能访问私钥。这项措施可防止私钥泄露，并保证了密钥的完整性。当然，公钥是完全公开的，可以自由分发。

准备打包自己的强命名程序集时，必须使用受严密保护的私钥对它进行签名。然而，在开发和测试程序集时，访问这些受严密保护的私钥可能有点碍事儿。有鉴于此，.NET Framework 提供了对**延迟签名**(delayed signing)的支持，该技术也称为**部分签名**(partial signing)。延迟签名允许只用公司的公钥生成程序集，暂时不用私钥。由于使用了公钥，引用了程序集的其他程序集会在它们的 AssemblyRef 元数据表的记录项中嵌入正确的公钥值。另外，它还使程序集能正确存储到 GAC 的内部结构中。当然，不用公司的私钥对文件进行签名，便无法实现防篡改保护。这是由于无法对程序集的文件进行哈希处理，无法在文件中嵌入数字签名。然而，失去这种保护不是一个大问题，因为只是在开发阶段才延迟签名。打包和部署程序集肯定会签名。

为了实现延迟签名，需要获取存储在文件中的公钥值，将文件名传给用于生成程序集的实用程序。(如本章前面所述，可用 SN.exe 的**-p** 开关从包含公钥/私钥对的文件中提取公钥。)另外，还必须让工具知道你想延迟对程序集的签名，暂不提供私钥。如果是 C#编译器，就指定**/delaysign** 编译器开关。如果是 Visual Studio，就打开项目属性页，在“签名”选项卡中勾选“仅延迟签名”。如果使用 AL.exe，就指定**/delay[sign]**命令行开关。

编译器或 AL.exe 一旦检测到要对程序集进行延迟签名，就会生成程序集的 AssemblyDef 清单记录项，其中将包含程序集的公钥。公钥使程序集能正确存储到 GAC。另外，这也不妨碍引用了该程序集的其他程序集的正确生成。在进行引用的程序集的 AssemblyRef 元数据表记录项中，会包含(被引用程序集的)正确公钥。创建程序集时，会在生成的 PE 文件中为 RSA 数字签名预留空间(实用程序根据公钥大小判断需预留多大空间)。要注意的是，文件内容不会在这个时候进行哈希处理。

目前生成的程序集没有有效签名。安装到 GAC 会失败，因为尚未对文件内容执行哈希处理——文件表面上已被篡改了。在需要安装到 GAC 的每台机器上，都必须禁止系统验证程序集文件的完整性。这要求使用 SN.exe 实用程序并指定**-Vr** 命令行开关。用这个开关执行 SN.exe，程序集的任何文件在运行时加载时，CLR 都会跳过对其哈希值的检查。在内部，SN 的**-Vr** 开关会将程序集的身份添加到以下注册表子项中：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\StrongName\Verification
```



重要提示 使用会修改注册表的任何实用程序时，请确保在 64 位操作系统中运行的是实用程序的 64 位版本。32 位 x86 实用程序默认安装到 C:\Program Files (x86)\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools 目录，64 位 x64 实用程序安装到 C:\Program Files(x86)\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools\x64 目录。

结束程序集的开发和测试之后，要正式对其进行签名，以便打包和部署它。为了对程序集进行签名，要再次使用 SN.exe 实用程序，但这次换用**-R** 开关，并指定包含了私钥的文件的名称。**-R** 开关指示 SN.exe 对文件内容进行哈希处理，用私钥对其进行签名，并将 RSA 数字签名嵌入文件中之前预留的空间。经过这一步之后，就可以部署完全签好名的程序集。在开发和测试机器上，不要忘记使用 SN.exe 的**-Vu** 或**-Vx** 命令行开关来重新启用对这个程序集的验证。下面总结了使用延迟签名技术开发程序集的步骤。

1. 开发期间，获取只含公司公钥的文件，使用**/keyfile** 和**/delaysign** 编译器开关编译程序集：

```
csc /keyfile:MyCompany.PublicKey /delaysign MyAssembly.cs
```

2. 生成程序集后执行以下命令，使 CLR 暂时信任程序集的内容，不对它进行哈希处理，也不对哈希值进行比较。这使程序集能顺利安装到 GAC(如果有必要的话)。现在，可以生成引用了这个程序集的其他程序集，并可随意测试程序集。注意，在每台开发用的机器上，以下命令行都只需执行一次，不必每次生成程序集都重复这一步：

```
SN.exe -Vr MyAssembly.dll
```

3. 准备好打包和部署程序集时，获取公司的私钥并执行以下命令。如果愿意，可以将这个新版本安装到 GAC 中。但只有先完成步骤 4，才能把它安装到 GAC 中。

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

4. 为了在实际环境中测试，请执行以下命令行，重新启用对这个程序集的验证：

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

本节开头就指出，大公司会将自己的密钥存储到硬件设备(比如智能卡)中。为了确保密钥的安全性，密钥值绝对不能固定存储在磁盘文件中。“加密服务提供程序”(Cryptographic Service Provider，CSP)提供了对这些密钥的位置进行抽象的容器。以 Microsoft 使用的 CSP 为例，一旦访问它提供的容器，就自动从一个硬件设备获取私钥。

如果公钥/私钥对在 CSP 容器中，必须为 CSC.exe、AL.exe 和 SN.exe 程序指定不同的开关。编译时(CSC.exe)指定**/keycontainer** 开关而不是**/keyfile** 开关；链接时(AL.exe)指定**/keyname** 开关而不是**/keyfile** 开关；使用强名称程序(SN.exe)对延迟签名的程序集进行重新签名时，指定**-Rc** 开关而不是**-R** 开关。SN.exe 提供了另外几个开关让开发人员与 CSP 互动。



重要提示　打包程序集前，如果想对它执行其他任何操作，延迟签名也非常有用。例如，可能想对程序集运行混淆器(obfuscator)程序。程序集完全签名后就不能运行混淆器了，否则哈希值就不正确了。所以，要混淆程序集文件，或者进行其他形式的“生成后”(post-build)操作，就利用延迟签名技术，先完成“生成后”操作，再用**-R** 或**-Rc** 开关运行 SN.exe 对程序集进行完全签名。

3.7　私有部署强命名程序集

在 GAC 中安装程序集有几方面的优势。GAC 使程序集能被多个应用程序共享，减少了总体物理内存消耗。另外，很容易将程序集的新版本部署到 GAC，让所有应用程序都通过发布者策略(本章稍后讲述)使用新版本。GAC 还实现了对程序集多个版本的并行管理。但 GAC 通常受到严密保护，只有管理员才能在其中安装程序集。另外，一旦安装到 GAC，就违反了“简单复制部署”[①]这一基本目标。

虽然强命名程序集能安装到 GAC，但绝非必须。事实上，只有由多个应用程序共享的程序集才应部署到 GAC。不用共享的应该私有部署。私有部署达成了“简单复制部署”目标，而且能更好地隔离应用程序及其程序集。另外，不要将 GAC 想象成新的 C:\Windows\System32 垃圾堆积场。这是因为新版本程序集不会相互覆盖，它们并行安装，每个安装都占用磁盘空间。

强命名程序集除了部署到 GAC 或者进行私有部署，还可部署到只有少数应用程序知道的目录。例如，假定强命名程序集由三个应用程序共享。安装时可创建三个目录，每个程序一个目录。再创建第四个目录，专门存储要共享的程序集。每个应用程序安装到自己的目录时都同时安装一个 XML 配置文件，用 **codeBase** 元素指出共享程序集路径。这样在运行时，CLR 就知道去那里查找共享程序集。但要注意，这个技术很少使用，也不太推荐使用，因为所有应用程序都不能独立决定何时卸载程序集的文件。



注意　配置文件的 **codeBase** 元素实际标记了一个 URL。这个 URL 可引用用户机器上的任何目录，也可引用 Web 地址。如果引用 Web 地址，CLR 会自动下载文件，并把它存储到用户的下载缓存(%UserProfile%\Local Settings\Application Data\Assembly 下的子目录)。将来引用时，CLR 将下载文件的时间戳与 URL 处的文件的时间戳进行对比。如果 URL 处的文件具有较新的时间戳，CLR 下载新版本并加载。否则，CLR 加载现有文件，不重复下载(从而增强性能)。本章稍后会展示包含 **codeBase** 元素的示例配置文件。

① 本来，简单复制一下程序集的文件就可以完成部署。但安装到 GAC 之后，就没有这么简单了。——译注

3.8 “运行时”如何解析类型引用

第 2 章开头展示了以下代码：

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

编译这些代码并生成程序集(假定名为 Program.exe)。运行应用程序，CLR 会加载并初始化自身，读取程序集的 CLR 头，查找标识了应用程序入口方法(**Main**)的 MethodDefToken，检索 MethodDef 元数据表找到方法的 IL 代码在文件中的偏移量，将 IL 代码 JIT 编译成本机代码(编译时会对代码进行验证以确保类型安全)，最后执行本机代码。下面就是 **Main** 方法的 IL 代码。要查看代码，请对程序集运行 ILDasm.exe 并选择“视图”|“显示字节”，双击树形视图中的 **Main** 方法。

```
.method public hidebysig static void Main() cil managed
// SIG: 00 00 01
{
  .entrypoint
  // Method begins at RVA 0x2050
  // Code size 11 (0xb)
  .maxstack 8
  IL_0000: /* 72 | (70)000001 */
           ldstr "Hi"
  IL_0005: /* 28 | (0A)000003 */
           call void [mscorlib]System.Console::WriteLine(string)
  IL_000a: /* 2A | */
           ret
} // end of method Program::Main
```

对这些代码进行 JIT 编译，CLR 会检测所有类型和成员引用，加载它们的定义程序集(如果尚未加载)。上述 IL 代码包含对 **System.Console.WriteLine** 的引用。具体地说，IL **call** 指令引用了元数据 token[①] 0A000003。该 token 标识 MemberRef 元数据表(表 0A)中的记录项 3。CLR 检查该 MemberRef 记录项，发现它的字段引用了 TypeRef 表中的记录项(**System.Console** 类型)。按照 TypeRef 记录项，CLR 被引导至一个 AssemblyRef 记录项："mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"。这时 CLR 就知道了它需要的是哪个程序集。接着，CLR 必须定位并加载该程序集。

解析引用的类型时，CLR 可能在以下三个地方找到类型。

- **相同文件**
  编译时便能发现对相同文件中的类型的访问，这称为**早期绑定**(early binding)[②]。类型直接从文件中加载，执行继续。

① 元数据 token 的详情请参见 2.4 节。——译注

② 对应地，在运行时通过反射机制绑定到类型并调用方法，就称为晚期绑定(late binding)。——译注

- **不同文件，相同程序集**
  “运行时”确保被引用的文件在当前程序集元数据的 FileDef 表中，检查加载程序集清单文件的目录，加载被引用的文件，检查哈希值以确保文件完整性。发现类型的成员，执行继续。

- **不同文件，不同程序集**
  如果引用的类型在其他程序集的文件中，“运行时”会加载被引用程序集的清单文件。如果需要的类型不在该文件中，就继续加载包含了类型的文件。发现类型的成员，执行继续。



注意　ModuleDef，ModuleRef 和 FileDef 元数据表在引用文件时使用了文件名和扩展名。但 AssemblyRef 元数据表只使用文件名，无扩展名。和程序集绑定时，系统通过探测目录来尝试定位文件，自动附加.dll 和 exe 扩展名，详见 2.8 节“简单管理控制(配置)”。

解析类型引用时有任何错误(找不到文件、文件无法加载、哈希值不匹配等)都会抛出相应异常。



注意　可以向 **System.AppDomain** 的 **AssemblyResolve**，**ReflectionOnly-AssemblyResolve** 和 TypeResolve 事件注册回调方法。在回调方法中执行解决绑定问题的代码，使应用程序不抛出异常而继续运行。

在上例中，CLR 发现 **System.Console** 在和调用者不同的程序集中实现。所以，CLR 必须查找那个程序集，加载包含程序集清单的 PE 文件。然后扫描清单，判断是哪个 PE 文件实现了类型。如果被引用的类型就在清单文件中，一切都很简单。如果类型在程序集的另一个文件中，CLR 必须加载那个文件，并扫描其元数据来定位类型。然后，CLR 创建它的内部数据结构来表示类型，JIT 编译器完成 **Main** 方法的编译。最后，**Main** 方法开始执行。图 3-2 演示了类型绑定过程。



重要提示　严格意义上，刚才的例子并非百分之百正确。如果引用的不是.NET Framework 程序集定义的方法和类型，刚才的讨论没有任何问题。但是，.NET Framework 程序集(MSCorLib.dll 就是其中之一)和当前运行的 CLR 版本紧密绑定。引用.NET Framework 程序集的任何程序集总是绑定到与 CLR 版本对应的那个版本(的.NET Framework 程序集)。这就是所谓的“统一”(Unification)。之所以要“统一”，是因为所有.NET Framework 程序集都是针对一个特定版本的 CLR 来完成测试的。因此，“统一”代码结构(code stack)可确保应用程序正确工作。

所以在前面的例子中，对 **System.Console** 的 **WriteLine** 方法的引用必然绑定到与当前CLR版本对应的MSCorLib.dll版本——无论程序集AssemblyRef元数据表引用哪个版本的 MSCorLib.dll。

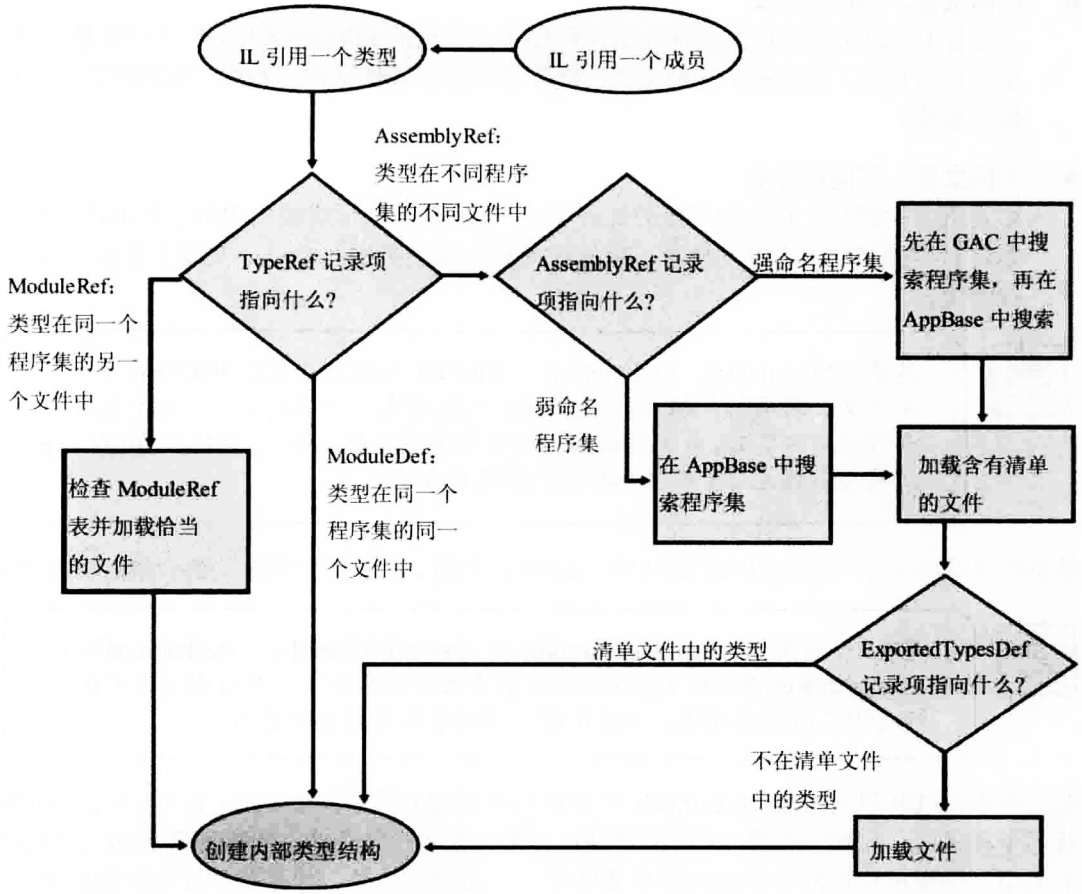


图 3-2　对于引用了方法或类型的 IL 代码，CLR 怎样通过元数据来定位定义了类型的程序集文件

还要注意，对于 CLR，所有程序集都根据名称、版本、语言文化和公钥来识别。但 GAC 根据名称、版本、语言文化、公钥和 CPU 架构来识别。在 GAC 中搜索程序集时，CLR 判断应用程序当前在什么类型的进程中运行，是 32 位 x64(可能使用 WoW64 技术)，64 位 x64，还是 32 位 ARM。然后，在 GAC 中搜索程序集时，CLR 首先搜索程序集的 CPU 架构专用版本。如果没有找到符合要求的，就搜索不区分 CPU 的版本。

本节描述的是 CLR 定位程序集的默认策略，但管理员或程序集发布者可能覆盖默认策略。接着两节将讨论如何更改 CLR 默认绑定策略。



注意　CLR 提供了将类型(类、结构、枚举、接口或委托)从一个程序集移动到另一个程序集的功能。例如，.NET 3.5 的 **System.TimeZoneInfo** 类在 System.Core.dll 程序集中定义。但在.NET 4.0 中，Microsoft 将这个类移动到了 MSCorLib.dll 程序集。将类型从一个程序集移动到另一个程序集，一般情况下会造成应用程序“中断”。但 CLR 提供了名为 **System.Runtime. CompilerServices.TypeForwardedToAttribute** 的特性，可将它应用于原始程序集(比如 System.Core.dll)。要向该特性的构造器传递一个 **System.Type** 类型的参数，指出应用程序要使用的新类型(现在是在 MSCorLib.dll 中定义)。CLR 的绑定器(binder)会利用到这个信息。由于 **TypeForwardedToAttribute** 的构造器获取的是 **Type**，所以包含该特性的程序集*要依赖于*现在定义类型的程序集。

为了使用这个功能，还要向新程序集中的类型应用名为 **System.Runtime. CompilerServices.TypeForwardedFromAttribute** 的特性，向该特性的构造器传递一个字符串来指出定义类型的旧程序集的全名。该特性一般由工具、实用程序和序列化使用。由于 **TypeForwardedFromAttribute** 的构造器获取的是 **String**，所以包含该特性的程序集不依赖于过去定义类型的程序集。

3.9　高级管理控制(配置)

2.8 节“简单管理控制(配置)”简要讨论了管理员如何影响 CLR 搜索和绑定程序集的方式。那一节演示了如何将被引用程序集的文件移动到应用程序基目录下的一个子目录，以及 CLR 如何通过应用程序的 XML 配置文件来定位发生移动的文件。

第 2 章只讨论了 **probing** 元素的 **privatePath** 属性，本节要讨论 XML 配置文件的其他元素。以下是一个示例 XML 配置文件：

```
<?xml version="1.0"?>
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
        <probing privatePath="AuxFiles;bin\subdir" />

        <dependentAssembly>

          <assemblyIdentity name="SomeClassLibrary"
            publicKeyToken="32ab4ba45e0a69a1" culture="neutral"/>

          <bindingRedirect
            oldVersion="1.0.0.0" newVersion="2.0.0.0" />

          <codeBase version="2.0.0.0"
            href="http://www.Wintellect.com/SomeClassLibrary.dll" />

        </dependentAssembly>

        <dependentAssembly>
```

```
            <assemblyIdentity name="TypeLib"
               publicKeyToken="1f2e74e897abbcfe" culture="neutral"/>

            <bindingRedirect
               oldVersion="3.0.0.0-3.5.0.0" newVersion="4.0.0.0" />

            <publisherPolicy apply="no" />

         </dependentAssembly>

      </assemblyBinding>
   </runtime>
</configuration>
```

这个 XML 文件为 CLR 提供了丰富的信息。具体如下所示。

- probing 元素
  查找弱命名程序集时，检查应用程序基目录下的 AuxFiles 和 bin\subdir 子目录。对于强命名程序集，CLR 检查 GAC 或者由 **codeBase** 元素指定的 URL。只有在未指定 **codeBase** 元素时，CLR 才会在应用程序的私有路径中检查强命名程序集。

- 第一个 dependentAssembly，assemblyIdentity 和 bindingRedirect 元素
  查找由控制着公钥标记 32ab4ba45e0a69a1 的组织发布的、语言文化为中性的 SomeClassLibrary 程序集的 1.0.0.0 版本时，改为定位同一个程序集的 2.0.0.0 版本。

- codeBase 元素
  查找由控制着公钥标记 32ab4ba45e0a69a1 的组织发布的、语言文化为中性的 SomeClassLibrary 程序集的 2.0.0.0 版本时，尝试在以下 URL 处发现它：*www.Wintellect.com/ SomeClassLibrary.dll*。虽然第 2 章没有特别指出，但 **codeBase** 元素也能用于弱命名程序集。如果是这样，程序集版本号会被忽略，而且根本就不应该在 XML **codeBase** 元素中写这个版本号。另外，**codeBase** 定义的 URL 必须指向应用程序基目录下的一个子目录。

- 第二个 dependentAssembly，assemblyIdentity 和 bindingRedirect 元素
  查找由控制着公钥标记 1f2e74e897abbcfe 的组织发布的、语言文化为中性的 TypeLib 程序集的 3.0.0.0 到 3.5.0.0 版本时(包括 3.0.0.0 和 3.5.0.0 在内)，改为定位同一个程序集的 4.0.0.0 版本。

- publisherPolicy 元素
  如果生成 TypeLib 程序集的组织部署了发布者策略文件(详情在下一节讲述)，CLR 应忽略该文件。

编译方法时，CLR 判断它引用了哪些类型和成员。根据这些信息，“运行时”检查进行引用的程序集的 AssemblyRef 表，判断程序集生成时引用了哪些程序集。然后，CLR 在应用程序配置文件中检查程序集/版本，进行指定的版本号重定向操作。随后，CLR 查找新的、重定向的程序集/版本。

如果 **publisherPolicy** 元素的 **apply** 特性设为 **yes**，或者该元素被省略，CLR 会在 GAC 中检

查新的程序集/版本，并进行程序集发布者认为有必要的任何版本号重定向操作。随后，CLR 查找新的、重定向的程序集/版本。(下一节将更详细讨论发布者策略。)最后，CLR 在机器的 Machine.config 文件中检查新的程序集/版本并进行指定的版本号重定向操作。

到此为止，CLR 已知道了它应加载的程序集版本，并尝试从 GAC 中加载。如果程序集不在 GAC 中，也没有 **codeBase** 元素，CLR 会像第 2 章描述的那样探测程序集。如果执行最后一次重定向操作的配置文件同时包含 **codeBase** 元素，CLR 会尝试从 **codeBase** 元素指定的 URL 处加载程序集。

利用这些配置文件，管理员可以实际地控制 CLR 加载的程序集。如果应用程序出现 bug，管理员可以和有问题的程序集的发布者取得联系。发布者将新程序集发送给管理员，让管理员安装。CLR 默认不加载新程序集，因为已生成的程序集并没有引用新版本。不过，管理员可以修改应用程序的 XML 配置文件，指示 CLR 加载新程序集。

如果管理员希望机器上的所有应用程序都使用新程序集，可以修改机器的 Machine.config 文件。这样每当应用程序引用旧程序集时，CLR 都自动加载新程序集。

如果发现新程序集没有修复 bug，管理员可以从配置文件中删除 **bindingRedirect** 设置，应用程序会恢复如初。说了这么多，其实重点只有一个：系统允许使用和元数据所记录的不完全匹配的程序集版本。这种额外的灵活性非常有用。

发布者策略控制

在上一节的例子中，是由程序集发布者将程序集的新版本发送给管理员，后者安装程序集，并手动编辑应用程序或机器的 XML 配置文件。通常，发布者希望在修复了程序集的 bug 之后，采用一种容易的方式将新程序集打包并分发给所有用户。但是，发布者还需要一种方式告诉每个用户的 CLR 使用程序集新版本，而不是继续使用旧版本。当然，可以指示每个用户手动修改应用程序或机器的 XML 配置文件，但这相当不便，而且容易出错。因此，发布者需要一种方式创建策略信息。新程序集安装到用户机器上时，会安装这种策略信息。本节将描述程序集的发布者如何创建这种策略信息。

假定你是程序集的发布者，刚刚修复了几个 bug，创建了程序集的新版本。打包要发送给所有用户的新程序集时，应同时创建一个 XML 配置文件。这个配置文件和以前讨论过的配置文件差不多。下面是用于 SomeClassLibrary.dll 程序集的示例文件(名为 SomeClassLibrary.config)：

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

        <assemblyIdentity name="SomeClassLibrary"
          publicKeyToken="32ab4ba45e0a69a1" culture="neutral"/>

        <bindingRedirect
          oldVersion="1.0.0.0" newVersion="2.0.0.0" />

        <codeBase version="2.0.0.0"
```

```
        href="http://www.Wintellect.com/SomeClassLibrary.dll"/>

    </dependentAssembly>
  </assemblyBinding>
 </runtime>
</configuration>
```

当然，发布者只能为自己创建的程序集设置策略。另外，发布者策略配置文件只能使用列出的这些元素；例如，**probing** 或 **publisherPolicy** 元素是不能使用的。

该配置文件告诉 CLR 一旦发现对 SomeClassLibrary 程序集的 1.0.0.0 版本的引用，就自动加载 2.0.0.0 版本。现在，发布者就可以创建包含该发布者策略配置文件的程序集，像下面这样运行 AL.exe：

```
AL.exe   /out:Policy.1.0. SomeClassLibrary.dll
         /version:1.0.0.0
         /keyfile:MyCompany.snk
         /linkresource: SomeClassLibrary.config
```

下面是对 AL.exe 的命令行开关的解释。

- **/out** 告诉 AL.exe 创建新 PE 文件，本例是 Policy.1.0.SomeClassLibrary.dll，其中除了一个清单什么都没有。程序集名称很重要。名称第一部分(Policy)告诉 CLR 该程序集包含发布者策略信息。第二部分和第三部分(1.0)告诉 CLR 这个发布者策略程序集适用于 major 和 minor 版本为 1.0 的任何版本的 SomeClassLibrary 程序集。发布者策略只能和程序集的 major 和 minor 版本号关联；不能和 build 或 revision 号关联。名称第四部分(SomeClassLibrary)指出与发布者策略对应的程序集名称。名称第五部分(dll)是现在要生成的发布者策略程序集文件的扩展名。

- /version
  标识发布者策略程序集的版本；这个版本号与 SomeClassLibrary 程序集本身没有任何关系。看得出来，发布者策略程序集本身也有一套版本机制。例如，发布者今天创建一个发布者策略，将 SomeClassLibrary 的版本 1.0.0.0 重定向到版本 2.0.0.0。未来，发布者可能将 SomeClassLibrary 的版本 1.0.0.0 重定向到版本 2.5.0.0。CLR 根据**/version** 开关指定的版本号来选择最新版本的发布者策略程序集。

- /keyfile
  告诉 AL.exe 使用发布者的“公钥/私钥对”对发布者策略程序集进行签名。这一对密钥还必须匹配所有版本的 SomeClassLibrary 程序集使用的密钥对。毕竟，只有这样，CLR 才知道 SomeClassLibrary 程序集和发布者策略文件由同一个发布者创建。

- /linkresource
  告诉 AL.exe 将 XML 配置文件作为程序集的一个单独的文件。最后的程序集由两个文件构成，两者必须随同新版本 SomeClassLibrary 程序集打包并部署到用户机器。顺便说一句，不能使用 AL.exe 的**/embedresource** 开关将 XML 配置文件嵌入程序集文件，从而获得一个单文件程序集。因为 CLR 要求 XML 文件独立。

一旦生成这个发布者策略程序集，就可随同新的 SomeClassLibrary.dll 程序集文件打包并部

署到用户机器。发布者策略程序集必须安装到 GAC。虽然 SomeClassLibrary 程序集也能安装到 GAC，但并不是必须的——它可以部署到应用程序基目录，也可部署到由 **codeBase** URL 标识的其他目录。



重要提示　只有部署程序集更新或 Service Pack 时才应创建发布者策略程序集。执行应用程序的全新安装不应安装发布者策略程序集。

关于发布者策略最后注意一点。假定发布者推出发布者策略程序集时，因为某种原因，新程序集引入的 bug 比它能修复的 bug 还要多，那么管理员可指示 CLR 忽略发布者策略程序集。这要求编辑应用程序的配置文件并添加以下 **publisherPolicy** 元素：

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

该元素可作为应用程序配置文件的**<assemblyBinding>**元素的子元素使用，使其应用于所有程序集；也可作为应用程序配置文件的**<dependantAssembly>**元素的子元素使用，使其应用于特定程序集。当 CLR 处理应用程序配置文件时，就知道自己不应在 GAC 中检查发布者策略程序集。CLR 会沿用旧版本程序集。但要注意，CLR 仍会检查并应用 Machine.config 文件中指定的任何策略。

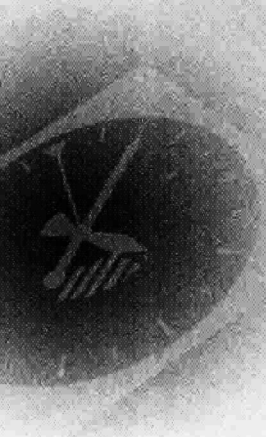


重要提示　创建发布者策略程序集，发布者相当于肯定了程序集不同版本的兼容性。如果新版本程序集不兼容某个老版本，就不应创建发布者策略程序集。通常，如果需要生成程序集的 bug 修复版本，就应该提供发布者策略程序集。作为发布者，应主动测试新版本程序集的向后兼容性。相反，如果要在程序集中增添新功能，就应该把它视为与之前版本没有关联的程序集，不要随带发布者策略程序集。另外，也不必测试这类程序集的向后兼容性。

[illegible]

[illegible]

[illegible]

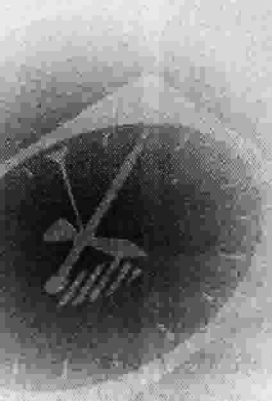


第II部分　设计类型

- 第 4 章　类型基础
- 第 5 章　基元类型、引用类型和值类型
- 第 6 章　类型和成员基础
- 第 7 章　常量和字段
- 第 8 章　方法
- 第 9 章　参数
- 第 10 章　属性
- 第 11 章　事件
- 第 12 章　泛型
- 第 13 章　接口

第II部分　设计实现

第4章
第5章
第6章
第7章
第8章
第9章
第10章
第11章
第12章
第13章



第4章 类型基础

本章内容：

- 所有类型都从 System.Object 派生
- 类型转换
- 命名空间和程序集
- 运行时的相互关系

本章讲述使用类型和 CLR 时需掌握的基础知识。具体地说，要讨论所有类型都具有的一组基本行为。还将讨论类型安全性、命名空间、程序集以及如何将对象从一种类型转换成另一种类型。本章最后会解释类型、对象、线程栈和托管堆在运行时的相互关系。

4.1 所有类型都从 System.Object 派生

“运行时”要求每个类型最终都从 **System.Object** 类型派生。也就是说，以下两个类型定义完全一致：

```
// 隐式派生自 Object
class Employee {
   ...
}
```

```
// 显式派生自 Object
class Employee : System.Object {
   ...
}
```

由于所有类型最终都从 **System.Object** 派生，所以每个类型的每个对象都保证了一组最基本的方法。具体地说，**System.Object** 类提供了如表 4-1 所示的公共实例方法。

表 4-1 System.Object 的公共方法

公共方法	说明
Equals	如果两个对象具有相同的值，就返回 **true**。欲知该方法的详情，请参见 5.3.2 节“对象相等性和同一性”
GetHashCode	返回对象的值的哈希码。如果某个类型的对象要在哈希表集合(比如 **Dictionary**)中作为键使用，类型应重写该方法。方法应该为不同对象提供良好分布[①]。将这个方法设计到 **Object** 中并不恰当。大多数类型永远不会在哈希表中作为键使用；该方法本该在接口中定义。欲知该方法的详情，请参见 5.4 节“对象哈希码”

① 所谓“良好分布”，是指针对所有输入，GetHashCode 生成的哈希值应该在所有整数中产生一个随机的分布。——译注

续表

公共方法	说明
ToString	默认返回类型的完整名称(**this.GetType().FullName**)。但经常重写该方法来返回包含对象状态表示的 **String** 对象。例如，核心类型(如 **Boolean** 和 **Int32**)重写该方法来返回它们的值的字符串表示。另外，经常出于调试的目的而重写该方法：调用后获得一个字符串，显示对象各字段的值。事实上，Microsoft Visual Studio 的调试器会自动调用该函数来显示对象的字符串表示。注意，**ToString** 理论上应察觉与调用线程关联的 **CultureInfo** 并采取相应行动。第 14 章“字符、字符串和文本处理”将更详细地讨论 **ToString**
GetType	返回从 **Type** 派生的一个类型的实例，指出调用 **GetType** 的那个对象是什么类型。返回的 **Type** 对象可以和反射类配合，获取与对象的类型有关的元数据信息。反射将在第 23 章“程序集加载和反射”讨论。**GetType** 是非虚方法，目的是防止类重写该方法，隐瞒其类型，进而破坏类型安全性

此外，从 **System.Object** 派生的类型能访问如表 4-2 所示的受保护方法。

表 4-2 System.Object 的受保护方法

受保护方法	说明
MemberwiseClone	这个非虚方法创建类型的新实例，并将新对象的实例字段设与 **this** 对象的实例字段完全一致。返回对新实例的引用[①]
Finalize	在垃圾回收器判断对象应该作为垃圾被回收之后，在对象的内存被实际回收之前，会调用这个虚方法。需要在回收内存前执行清理工作的类型应重写该方法。第 21 章“托管堆和垃圾回收”会更详细地讨论这个重要的方法

CLR 要求所有对象都用 **new** 操作符创建。以下代码展示了如何创建一个 **Employee** 对象：

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

以下是 **new** 操作符所做的事情。

1. 计算类型及其所有基类型(一直到 **System.Object**，虽然它没有定义自己的实例字段)中定义的所有实例字段需要的字节数。堆上每个对象都需要一些额外的成员[②]，包括“类型对象指针”(type object pointer)和“同步块索引”(sync block index)。CLR 利用这些成员管理对象。额外成员的字节数要计入对象大小。

2. 从托管堆中分配类型要求的字节数，从而分配对象的内存，分配的所有字节都设为零(0)。

① 作者在这段话里引用了两种不同的“实例”。一种是类的实例，也就是对象；另一种是类中定义的实例字段。所谓“实例字段”，就是指非静态字段，有时也称为“实例成员”。简单地说，实例成员属于类的对象，静态成员属于类。——译注

② 称为 overhead 成员，或者说“开销成员”。——译注

3. 初始化对象的“类型对象指针”和“同步块索引”成员。

4. 调用类型的实例构造器，传递在 **new** 调用中指定的实参(上例就是字符串 **"ConstructorParam1"**)。大多数编译器都在构造器中自动生成代码来调用基类构造器。每个类型的构造器都负责初始化该类型定义的实例字段。最终调用 **System.Object** 的构造器，该构造器什么都不做，简单地返回。

new 执行了所有这些操作之后，返回指向新建对象一个引用(或指针)。在前面的示例代码中，该引用保存到变量 **e** 中，后者具有 **Employee** 类型。

顺便说一句，没有和 **new** 操作符对应的 **delete** 操作符；换言之，没有办法显式释放为对象分配的内存。CLR 采用了垃圾回收机制(详情在第 21 章讲述)，能自动检测到一个对象不再被使用或访问，并自动释放对象的内存。

4.2　类型转换

CLR 最重要的特性之一就是类型安全。在运行时，CLR 总是知道对象的类型是什么。调用 **GetType** 方法即可知道对象的确切类型。由于它是非虚方法，所以一个类型不可能伪装成另一个类型。例如，**Employee** 类型不能重写 **GetType** 方法并返回一个 **SuperHero** 类型。

开发人员经常需要将对象从一种类型转换为另 种类型。CLR 允许将对象转换为它的(实际)类型或者它的任何基类型。每种编程语言都规定了开发人员具体如何进行这种转型操作。例如，C#不要求任何特殊语法即可将对象转换为它的任何基类型，因为向基类型的转换被认为是一种安全的隐式转换。然而，将对象转换为它的某个派生类型时，C#要求开发人员只能进行显式转换，因为这种转换可能在运行时失败。以下代码演示了向基类型和派生类型的转换：

```
// 该类型隐式派生自 System.Object
internal class Employee {
   ...
}

public sealed class Program {
   public static void Main() {
      // 不需要转型，因为 new 返回一个 Employee 对象，
      // 而 Object 是 Employee 的基类型
      Object o = new Employee();

      // 需要转型，因为 Employee 派生自 Object。
      // 其他语言(比如 Visual Basic) 也许不要求像
      // 这样进行强制类型转换
      Employee e = (Employee) o;
   }
}
```

这个例子展示了你需要做什么，才能让编译器顺利编译这些代码。接着，让我们看看运行时发生的事情。在运行时，CLR 检查转型操作，确定总是转换为对象的实际类型或者它的任何基类型。例如，以下代码虽然能通过编译，但会在运行时抛出 **InvalidCastException** 异常：

```
internal class Employee {
    ...
}
internal class Manager : Employee {
    ...
}

public sealed class Program {
    public static void Main() {
        // 构造一个 Manager 对象，把它传给 PromoteEmployee，
        // Manager "属于" (IS-A) Employee，所以 PromoteEmployee 能成功运行
        Manager m = new Manager();
        PromoteEmployee(m);

        // 构造一个 DateTime 对象，把它传给 PromoteEmployee。
        // DateTime 不是从 Employee 派生的，所以 PromoteEmployee
        // 抛出 System.InvalidCastException 异常
        DateTime newYears = new DateTime(2010, 1, 1);
        PromoteEmployee(newYears);
    }

    public static void PromoteEmployee(Object o) {
        // 编译器在编译时无法准确地获知对象 o
        // 引用的是什么类型，因此编译器允许代码
        // 通过编译。但在运行时，CLR 知道了 o 引用
        // 的是什么类型(在每次执行转型的时候)，
        // 所以它会核实对象的类型是不是 Employee 或者
        // 从 Employee 派生的任何类型
        Employee e = (Employee) o;
        ...
    }
}
```

Main 构造一个 **Manager** 对象并将其传给 **PromoteEmployee**。这些代码能成功编译并运行，因为 **Manager** 最终从 **Object** 派生，而 **PromoteEmployee** 期待的正是一个 **Object**。进入 **PromoteEmployee** 内部之后，CLR 核实对象 **o** 引用的就是一个 **Employee** 对象，或者是从 **Employee** 派生的一个类型的对象。由于 **Manager** 从 **Employee** 派生，所以 CLR 执行类型转换，允许 **PromoteEmployee** 继续执行。

PromoteEmployee 返回后，**Main** 构造一个 **DateTime** 对象，将其传给 **PromoteEmployee**。同样地，**DateTime** 从 **Object** 派生，所以编译器会顺利编译调用 **PromoteEmployee** 的代码。但进入 **PromoteEmployee** 内部之后，CLR 会检查类型转换，发现 **o** 引用一个 **DateTime**，既不是 **Employee**，也不是从 **Employee** 派生的任何类型。此时 CLR 会禁止转型，并抛出 **System.InvalidCastException** 异常。

如果 CLR 允许这样的转型，就毫无类型安全性可言了，将出现难以预料的结果——包括应用程序崩溃，以及安全漏洞的出现(因为一种类型能轻松地伪装成另一种类型)。类型伪装是许多安全漏洞的根源，它还会破坏应用程序的稳定性和健壮性。因此，类型安全是 CLR 极其重要的一个特点。

顺便说一句，声明 **PromoteEmployee** 方法的正确方式是将参数类型指定为 **Employee**，而非指定为 **Object**。如果这样修改，该例编译时就能报错，而非等到运行时才报错。我之所以使用 **Object**，是为了演示 C#编译器和 CLR 如何处理类型转换和类型安全性。

使用 C#的 is 和 as 操作符来转型

在C#语言中进行类型转换的另一种方式是使用**is**操作符。**is**检查对象是否兼容于指定类型，返回 **Boolean** 值 **true** 或 **false**。注意，**is** 操作符永远不抛出异常，例如以下代码：

```
Object o = new Object();
Boolean b1 = (o is Object);          // b1 为 true.
Boolean b2 = (o is Employee);        // b2 为 false.
```

如果对象引用 **null**，**is** 操作符总是返回 **false**，因为没有可检查其类型的对象。

is 操作符通常像下面这样使用：

```
if (o is Employee) {
    Employee e = (Employee) o;
    // 在 if 语句剩余的部分中使用 e
}
```

在上述代码中，CLR 实际检查两次对象类型。**is** 操作符首先核实 **o** 是否兼容于 **Employee** 类型。如果是，在 **if** 语句内部转型时，CLR 再次核实 **o** 是否引用一个 **Employee**。CLR 的类型检查增强了安全性，但无疑会对性能造成一定影响。这是因为 CLR 首先必须判断变量(**o**)引用的对象的实际类型。然后，CLR 必须遍历继承层次结构，用每个基类型去核对指定的类型(**Employee**)。由于这是一个相当常用的编程模式，所以 C#专门提供了 **as** 操作符，目的就是简化这种代码的写法，同时提升其性能。

```
Employee e = o as Employee;
if (e != null) {
    // 在 if 语句中使用 e
}
```

在这段代码中，CLR 核实 **o** 是否兼容于 **Employee** 类型；如果是，**as** 返回对同一个对象的非 **null** 引用。如果 **o** 不兼容于 **Employee** 类型，**as** 返回 **null**。注意，**as** 操作符造成 CLR 只校验一次对象类型。**if** 语句只检查 **e** 是否为 **null**；这个检查的速度比校验对象的类型快得多。

as 操作符的工作方式与强制类型转换一样，只是它永远不抛出异常——相反，如果对象不能转型，结果就是 **null**。所以，正确做法是检查最终生成的引用是否为 **null**。企图直接使用最终生成的引用会抛出 **System.NullReferenceException** 异常。以下代码对此进行了演示：

```
Object o = new Object();          // 新建 Object 对象
Employee e = o as Employee;       // 将 o 转型为 Employee
// 上述转型会失败，不抛出异常，但 e 被设为 null

e.ToString(); // 访问 e 抛出 NullReferenceException 异常
```

为了确定自己理解了上述内容，请完成以下小测验。假定以下两个类定义确实存在：

```
internal class B {          // 基类(Base class)
}

internal class D : B {      // 派生类(Derived class)
}
```

现在检查表 4-3 列出的 C#代码。针对每一行代码，都用勾号注明该行代码是将成功编译和

执行(OK)，将造成编译时错误(CTE)，还是将造成运行时错误(RTE)。

表 4-3 类型安全性测验

语句	OK	CTE(编译时错误)	RTE(运行时错误)
Object o1 = new Object();	√		
Object o2 = new B();	√		
Object o3 = new D();	√		
Object o4 = o3;	√		
B b1 = new B();	√		
B b2 = new D();	√		
D d1 = new D();	√		
B b3 = new Object();		√	
D d2 = new Object();		√	
B b4 = d1;	√		
D d3 = b2;		√	
D d4 = (D) d1;	√		
D d5 = (D) b2;	√		
D d6 = (D) b1;			√
B b5 = (B) o1;			√
B b6 = (D) b2;	√		



注意 C#允许类型定义转换操作符方法，详情参见 8.5 节“转换操作符方法”。只有在使用转型表达式时才调用这些方法；使用 C# **as** 或 **is** 操作符时永远不调用它们。

4.3 命名空间和程序集

命名空间对相关的类型进行逻辑分组，开发人员可通过命名空间方便地定位类型。例如，**System.Text** 命名空间定义了执行字符串处理的类型，而 **System.IO** 命名空间定义了执行 I/O 操作的类型。以下代码构造一个 **System.IO.FileStream** 对象和一个 **System.Text.StringBuilder** 对象：

```
public sealed class Program {
   public static void Main() {
      System.IO.FileStream fs = new System.IO.FileStream(...);
      System.Text.StringBuilder sb = new System.Text.StringBuilder();
   }
}
```

像这样写代码很繁琐，应该有一种简单方式直接引用 **FileStream** 和 **StringBuilder** 类型，减少打字量。幸好，许多编译器都提供了某种机制让程序员少打一些字。C#编译器通过 **using** 指令提供这个机制。以下代码和前面的例子完全一致：

```
using System.IO;       // 尝试附加"System.IO."前缀
using System.Text;     // 尝试附加"System.Text."前缀

public sealed class Program {
   public static void Main() {
      FileStream fs = new FileStream(...);
```

```
        StringBuilder sb = new StringBuilder();
    }
}
```

对于编译器，命名空间的作用就是为类型名称附加以句点分隔的符号，使名称变得更长，更可能具有唯一性。所以在本例中，编译器将对 **FileStream** 的引用解析为 **System.IO.FileStream**，将对 **StringBuilder** 的引用解析为 **System.Text.StringBuilder**。

C#的 **using** 指令是可选的，如果愿意，完全可以输入类型的完全限定名称。C#的 **using** 指令指示编译器尝试为类型名称附加不同的前缀，直至找到匹配项。



重要提示　CLR 对“命名空间”一无所知。访问类型时，CLR 需要知道类型的完整名称(可能是相当长的、包含句点符号的名称)以及该类型的定义具体在哪个程序集中。这样“运行时”才能加载正确程序集，找到目标类型，并对其进行操作。

在前面的示例代码中，编译器需要保证引用的每个类型都确实存在，而且代码以正确方式使用类型——也就是调用确实存在的方法，向方法传递正确数量的实参，保证实参具有正确类型，正确使用方法返回值，等等。如果编译器在源代码文件或者引用的任何程序集中找不到具有指定名称的类型，就会在类型名称前附加 **System.IO.**前缀，检查这样生成的名称是否与现有类型匹配。如果仍然找不到匹配项，就继续为类型名称附加 **System.Text.**前缀。在前面例子中的两个 **using** 指令的帮助下，只需在代码中输入 **FileStream** 和 **StringBuilder** 这两个简化的类型名称，编译器就会自动将引用展开成 **System.IO.FileStream** 和 **System.Text.StringBuilder**。这样不仅能极大减少打字，还增强了代码可读性。

检查类型定义时，编译器必须知道要在什么程序集中检查。第 2 章和第 3 章讲过，这通过 **/reference** 编译器开关实现。编译器扫描引用的所有程序集，在其中查找类型定义。一旦找到正确的程序集，程序集信息和类型信息就嵌入生成的托管模块的元数据中。为了获取程序集信息，必须将定义了被引用类型的程序集传给编译器。C#编译器自动在 MSCorLib.dll 程序集中查找被引用类型，即使没有显式告诉它这样做。MSCorLib.dll 程序集包含所有核心 Framework 类库(FCL)类型(如 **Object**，**Int32**，**String** 等)的定义。

编译器对待命名空间的方式存在潜在问题：可能两个(或更多)类型在不同命名空间中同名。Microsoft 强烈建议开发人员为类型定义具有唯一性的名称。但有时非不为也，是不能也。“运行时”鼓励组件重用。例如，应用程序可能同时利用了 Microsoft 和 Wintellect 创建的组件。假定两家公司都提供名为 **Widget** 的类型，两个类型做的事情完全不同。由于干涉不了类型命名，所以引用时用完全限定名称区分它们。用 **Microsoft.Widget** 引用 Microsoft 的 Widget，**用 Wintellect.Widget** 引用 Wintellect 的 Widget。以下代码对 Widget 的引用会产生歧义，C#编译器报告错误消息：**CS0104：“Widget”是“Microsoft.Widget”和“Wintellect.Widget”之间的不明确的引用**。

```
using Microsoft;       // 尝试附加"Microsoft."前缀
using Wintellect;      // 尝试附加"Wintellect."前缀

public sealed class Program {
```

```
    public static void Main() {
        Widget w = new Widget();  // 不明确的引用
    }
}
```

为了消除歧义，必须像下面这样显式告诉编译器要创建哪个 Widget。

```
using Microsoft;       // 尝试附加"Microsoft."前缀
using Wintellect;      // 尝试附加"Wintellect."前缀

public sealed class Program {
    public static void Main() {
        Wintellect.Widget w = new Wintellect.Widget(); // 无歧义
    }
}
```

C# **using** 指令的另一种形式允许为类型或命名空间创建别名。如果只想使用命名空间中的少量类型，不想它的所有类型都跑出来“污染”全局命名空间，别名就显得十分方便。以下代码演示了如何用另一个办法解决前例的歧义性问题：

```
using Microsoft;            // 尝试附加"Microsoft."前缀
using Wintellect;           // 尝试附加"Wintellect."前缀

// 将 WintellectWidget 符号定义成 Wintellect.Widget 的别名
using WintellectWidget = Wintellect.Widget;

public sealed class Program {
    public static void Main() {
        WintellectWidget w = new WintellectWidget(); // 现在没错误了
    }
}
```

这些消除类型歧义性的方法都十分有用，但有时还需更进一步。假定 Australian Boomerang Company(澳大利亚回旋镖公司，ABC)和 Alaskan Boat Corporation(阿拉斯加船业公司，ABC)都创建了名为 **BuyProduct** 的类型。该类型随同两家公司的程序集发布。两家公司都创建了名为 **ABC** 的命名空间，其中都包含名为 **BuyProduct** 的类型。任何人要开发应用程序来同时购买这两家公司出售的回旋镖和船都会遇到麻烦——除非编程语言提供了某种方式，能通过编程来区分程序集而非仅仅区分命名空间。幸好，C#编译器提供了名为**外部别名**(extern alias)的功能，它解决了这个虽然极其罕见但仍有可能发生的问题。外部别名还允许从同一个程序集的两个(或更多)不同的版本中访问一个类型。欲知外部别名的详情，请参见 C#语言规范。

要在库中设计由第三方使用的类型，应该在专门的命名空间中定义这些类型。这样编译器就能轻松消除它们的歧义。事实上，为了降低发生冲突的概率，应该使用自己的完整公司名称(而不是首字母缩写或者其他简称)来作为自己的顶级命名空间名称。参考文档，可以看到 Microsoft 为自己的类型使用了命名空间“Microsoft”，比如 **Microsoft.CSharp**，**Microsoft.VisualBasic** 和 **Microsoft.Win32**。

创建命名空间很简单，像下面这样写一个命名空间声明就可以了(以 C#为例)：

```
namespace CompanyName {
    public sealed class A {              // TypeDef: CompanyName.A
    }
```

```
    namespace X {
       public sealed class B { ... }    // TypeDef: CompanyName.X.B
    }
}
```

类定义右侧的注释指出编译器在类型定义元数据表中添加的实际类型名称；这是 CLR 看到的实际类型名称。

一些编译器根本不支持命名空间，还有一些编译器允许自由定义“命名空间”对于语言的含义。在 C#中，**namespace** 指令的作用只是告诉编译器为源代码中出现的每个类型名称附加命名空间名称前缀，让程序员少打一些字。

命名空间和程序集的关系

注意，命名空间和程序集(实现类型的文件)不一定相关。特别是，同一个命名空间中的类型可能在不同程序集中实现。例如，**System.IO.FileStream** 类型在 MSCorLib.dll 程序集中实现，而 **System.IO.FileSystemWatcher** 类型在 System.dll 程序集中实现。

同一个程序集也可能包含不同命名空间中的类型。例如，**System.Int32** 和 **System.Text.StringBuilder** 类型都在 MSCorLib.dll 程序集中。

在文档中查找类型时，文档会明确指出类型所属的命名空间，以及实现了该类型的程序集。如图 4-1 所示，可以清楚地看到(在“语法”小节的上方)，**ResXFileRef** 类型是 **System.Resources** 命名空间的一部分，在 **System.Windows.Forms.dll** 程序集中实现。为了编译引用了 **ResXFileRef** 类型的代码，要在源代码中添加 **using System.Resources;** 指令，而且要使用 **/r:System.Windows.Forms.dll** 编译器开关。



图 4-1　文档显示了类型的命名空间和程序集信息

4.4 运行时的相互关系

本节将解释类型、对象、线程栈和托管堆在运行时的相互关系。此外，还将解释调用静态方法、实例方法和虚方法的区别。首先从一些计算机基础知识开始。虽然下面讨论的东西不是 CLR 特有的，但掌握了这些之后，就有了一个良好的理论基础。接着，就可以将我们的讨论转向 CLR 特有的内容。

图 4-2 展示了已加载 CLR 的一个 Windows 进程。该进程可能有多个线程。线程创建时会分配到 1 MB 的栈。栈空间用于向方法传递实参，方法内部定义的局部变量也在栈上。图 4-2 展示了线程的栈内存(右侧)。栈从高位内存地址向低位内存地址构建。图中线程已执行了一些代码，栈上已经有一些数据了(栈顶部的阴影区域)。现在，假定线程执行的代码要调用 **M1** 方法。

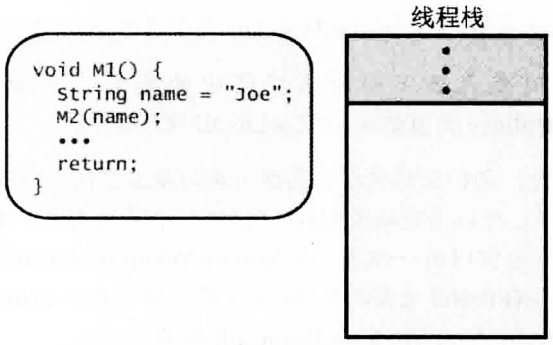


图 4-2 一个线程的栈，当前准备调用 M1 方法

最简单的方法包含“序幕”(prologue)代码，在方法开始做工作前对其进行初始化；还包含“尾声”(epilogue)代码，在方法做完工作后对其进行清理，以便返回至调用者。**M1** 方法开始执行时，它的“序幕”代码在线程栈上分配局部变量 **name** 的内存，如图 4-3 所示。

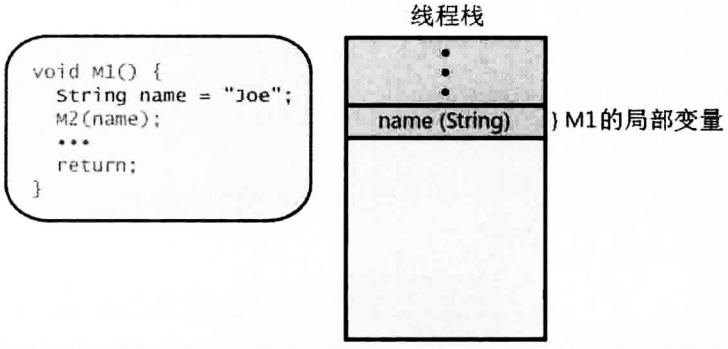


图 4-3 在线程栈上分配 M1 的局部变量

然后，**M1** 调用 **M2** 方法，将局部变量 **name** 作为实参传递。这造成 **name** 局部变量中的地址被压入栈(参见图 4-4)。**M2** 方法内部使用参数变量 **s** 标识栈位置(注意，有的 CPU 架构用

寄存器传递实参以提升性能，但这个区别对于当前的讨论来说并不重要)。另外，调用方法时还会将“返回地址”压入栈。被调用的方法在结束之后应返回至该位置(同样参见图 4-4)。

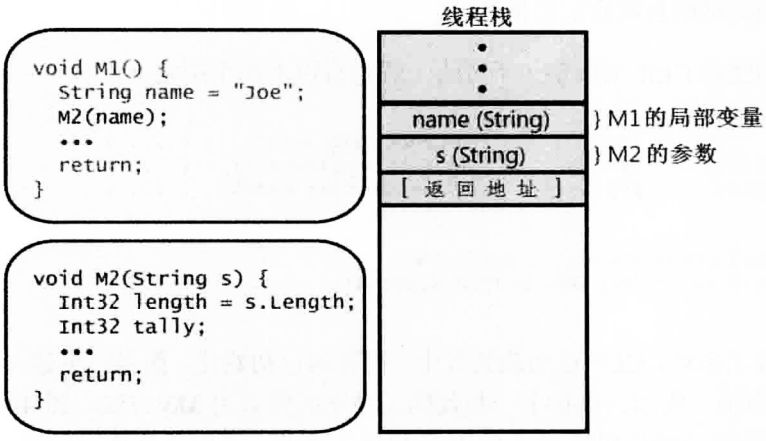


图 4-4　M1 调用 M2 时，将实参和返回地址压入线程栈

M2 方法开始执行时，它的“序幕”代码在线程栈中为局部变量 **length** 和 **tally** 分配内存，如图 4-5 所示。然后，**M2** 方法内部的代码开始执行。最终，**M2** 抵达它的 **return** 语句，造成 CPU 的指令指针被设置成栈中的返回地址，M2 的栈帧[①]展开(unwind)[②]，恢复成图 4-3 的样子。之后，**M1** 继续执行 **M2** 调用之后的代码，**M1** 的栈帧将准确反映 **M1** 需要的状态。

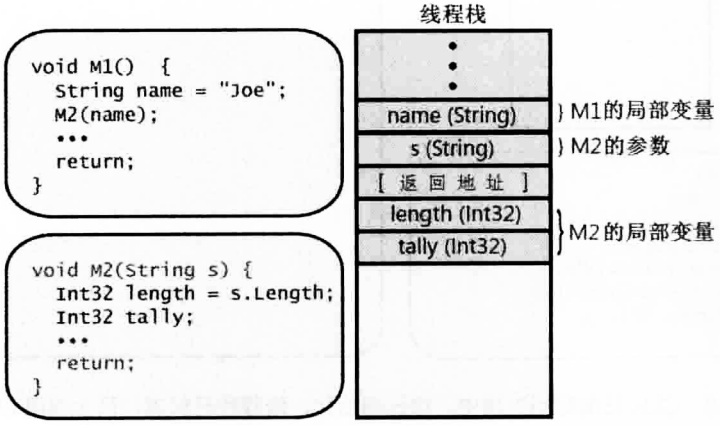


图 4-5　在线程栈上分配 M2 的局部变量

最终，**M1** 会返回到它的调用者。这同样通过将 CPU 的指令指针设置成返回地址来实现(这

① 栈帧(stack frame)代表当前线程的调用栈中的一个方法调用。执行线程的过程中，进行的每个方法调用都会在调用栈中创建并压入一个 **StackFrame**。——译注

② unwind 一般翻译成“展开”，但这并不是一个很好的翻译。wind 和 unwind 源于生活。把线缠到线圈上称为 wind；从线圈上松开称为 unwind。同样地，调用方法时压入栈帧，称为 wind；方法执行完毕，弹出栈帧，称为 unwind。把这几张图的线程栈看成一个线圈，就很容易理解了。——译注

个返回地址在图中未显示，但它应该刚好在栈中的 **name** 实参上方)，**M1** 的栈帧展开(unwind)，恢复成图 4-2 的样子。之后，调用 **M1** 的方法继续执行 **M1** 调用之后的代码，那个方法的栈帧将准确反映它需要的状态。

现在，让我们围绕 CLR 来调整一下讨论。假定有以下两个类定义：

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

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

Windows 进程已启动，CLR 已加载到其中，托管堆已初始化，而且已创建一个线程(连同它的 1 MB 栈空间)。线程已执行了一些代码，马上就要调用 **M3** 方法。图 4-6 展示了目前的状态。**M3** 方法包含的代码演示了 CLR 是如何工作的。平时不这样写代码，因为它们没有做什么真正有用的事情。

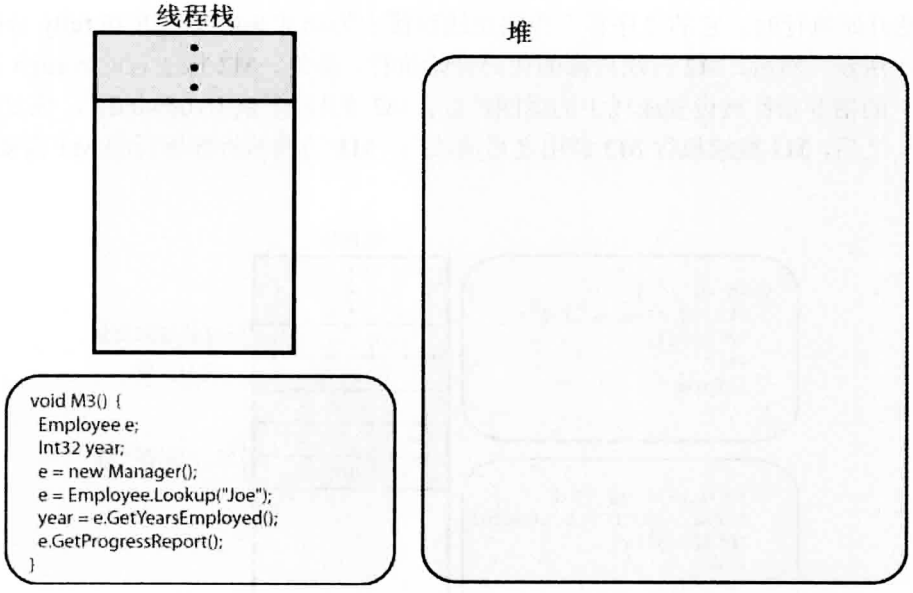


图 4-6　CLR 已加载到进程中，堆已初始化，线程栈已创建，马上调用 M3 方法

JIT 编译器将 **M3** 的 IL 代码转换成本机 CPU 指令时，会注意到 M3 内部引用的所有类型，包括 **Employee**，**Int32**，**Manager** 以及 **String**(因为**"Joe"**)。这时 CLR 要确认定义了这些类型的所有程序集都已加载。然后，利用程序集的元数据，CLR 提取与这些类型有关的信息，创建一些数据结构来表示类型本身。图 4-7 展示了为 **Employee** 和 **Manager** 类型对象使用的数据结构。由于线程在调用 **M3** 前已执行了一些代码，所以不妨假定 **Int32** 和 **String** 类型对象已经创建好了(这是极有可能的，因为它们都是很常用的类型)，所以图中没有显示它们。

稍微讨论一下这些类型对象。本章前面讲过，堆上所有对象都包含两个额外成员：类型对象指针(type object pointer)和同步块索引(sync block index)。如图所示，**Employee** 和 **Manager** 类型对象都有这两个成员。定义类型时，可在类型内部定义静态数据字段。为这些静态数据字段提供支援的字节在类型对象自身中分配。每个类型对象最后都包含一个方法表。在方法表中，类型定义的每个方法都有对应的记录项。第 1 章已讨论过该方法表。由于 **Employee** 类型定义了 3 个方法(**GetYearsEmployed**，**GetProgressReport** 和 **Lookup**)，所以 **Employee** 的方法表有 3 个记录项。**Manager** 类型只定义了 1 个方法(**GetProgressReport** 的重写版本)，所以 **Manager** 的方法表只有 1 个记录项。

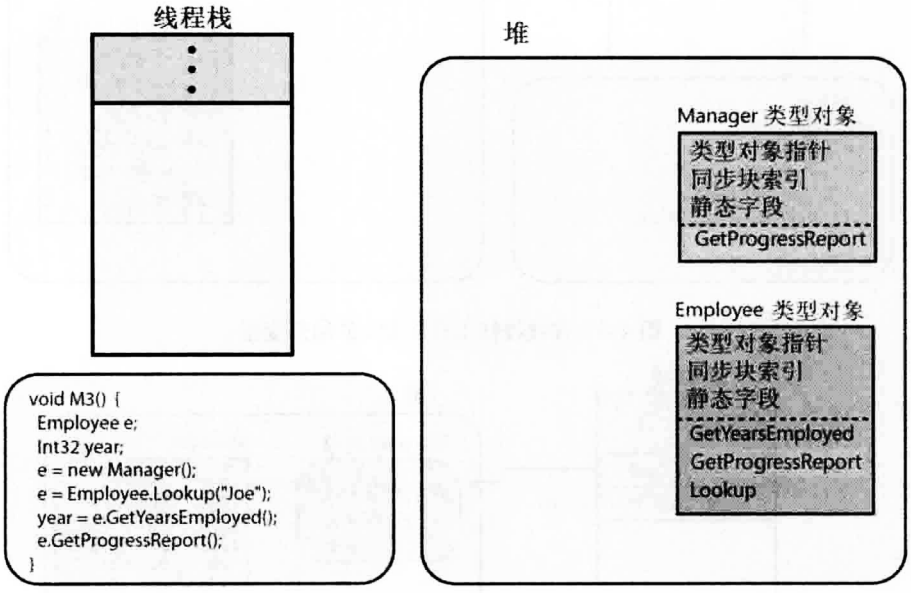


图 4-7　Employee 和 Manager 类型对象在 M3 被调用时创建

当 CLR 确认方法需要的所有类型对象都已创建，**M3** 的代码已经编译之后，就允许线程执行 **M3** 的本机代码。**M3** 的“序幕”代码执行时必须在线程栈中为局部变量分配内存，如图 4-8 所示。顺便说一句，作为方法“序幕”代码的一部分，CLR 自动将所有局部变量初始化为 **null** 或 **0**(零)。然而，如果代码试图访问尚未显式初始化的局部变量，C#会报告错误消息：**使用了未赋值的局部变量**。

然后，**M3** 执行代码构造一个 **Manager** 对象。这造成在托管堆创建 **Manager** 类型的一个实例(也就是一个 **Manager** 对象)，如图 4-9 所示。可以看出，和所有对象一样，**Manager** 对象也有类型对象指针和同步块索引。该对象还包含必要的字节来容纳 **Manager** 类型定义的所有实例数据字段，以及容纳由 **Manager** 的任何基类(本例就是 **Employee** 和 **Object**)定义的所有实例字段。任何时候在堆上新建对象，CLR 都自动初始化内部的“类型对象指针”成员来引用和对象对应的类型对象(本例就是 **Manager** 类型对象)。此外，在调用类型的构造器(本质上是可能修改某些实例数据字段的方法)之前，CLR 会先初始化同步块索引，并将对象的所有实例字段设为 **null** 或 **0**(零)。**new** 操作符返回 **Manager** 对象的内存地址，该

地址保存到变量 **e** 中(**e** 在线程栈上)。

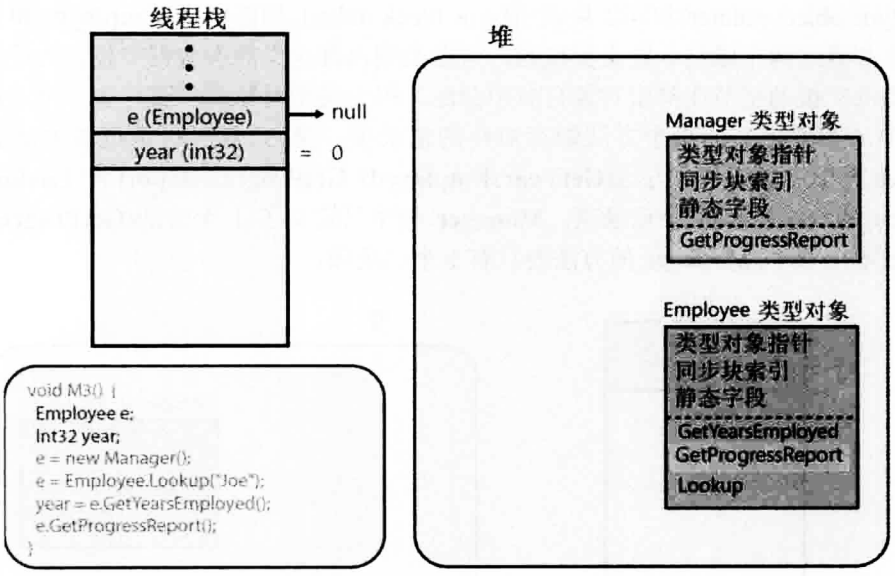


图 4-8　在线程栈上分配 M3 的局部变量

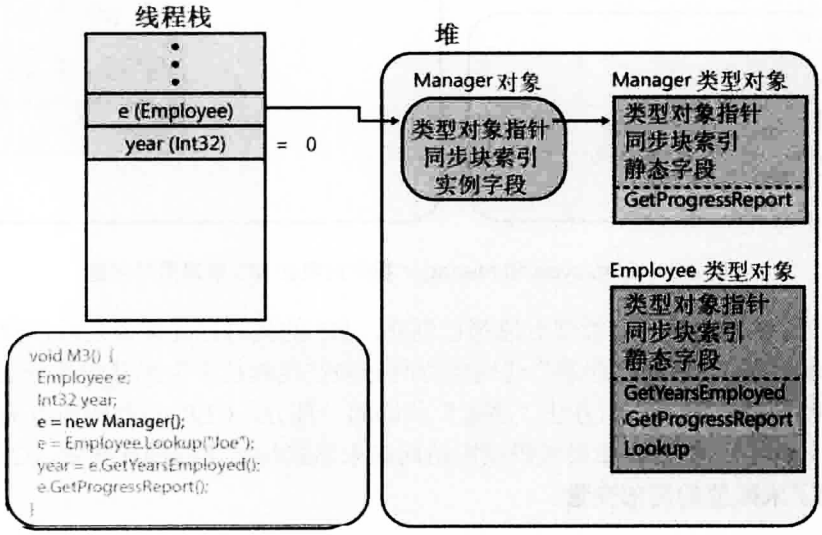


图 4-9　分配并初始化 Manager 对象

M3 的下一行代码调用 **Employee** 的静态方法 **Lookup**。调用静态方法时，CLR 会定位与定义静态方法的类型对应的类型对象。然后，JIT 编译器在类型对象的方法表中查找与被调用方法对应的记录项，对方法进行 JIT 编译(如果需要的话)，再调用 JIT 编译好的代码。本例假定 **Employee** 的 **Lookup** 方法要查询数据库来查找 Joe。再假定数据库指出 Joe 是公司的一名经理，所以在内部，**Lookup** 方法在堆上构造一个新的 **Manager** 对象，用 Joe 的信息初始化它，返回该对象的地址。该地址保存到局部变量 **e** 中。这个操作的结果如图 4-10 所示。

注意，**e** 不再引用第一个 **Manager** 对象。事实上，由于没有变量引用该对象，所以它是未来垃圾回收的主要目标。垃圾回收机制将自动回收(释放)该对象占用的内存。

M3 的下一行代码调用 **Employee** 的非虚实例方法 **GetYearsEmployed**。调用非虚实例方法时，JIT 编译器会找到与“发出调用的那个变量(**e**)的类型(**Employee**)”对应的类型对象(**Employee** 类型对象)。这时的变量 **e** 被定义成一个 **Employee**。如果 **Employee** 类型没有定义正在调用的那个方法，JIT 编译器会回溯类层次结构(一直回溯到 **Object**)，并在沿途的每个类型中查找该方法。之所以能这样回溯，是因为每个类型对象都有一个字段引用了它的基类型，这个信息在图中没有显示。

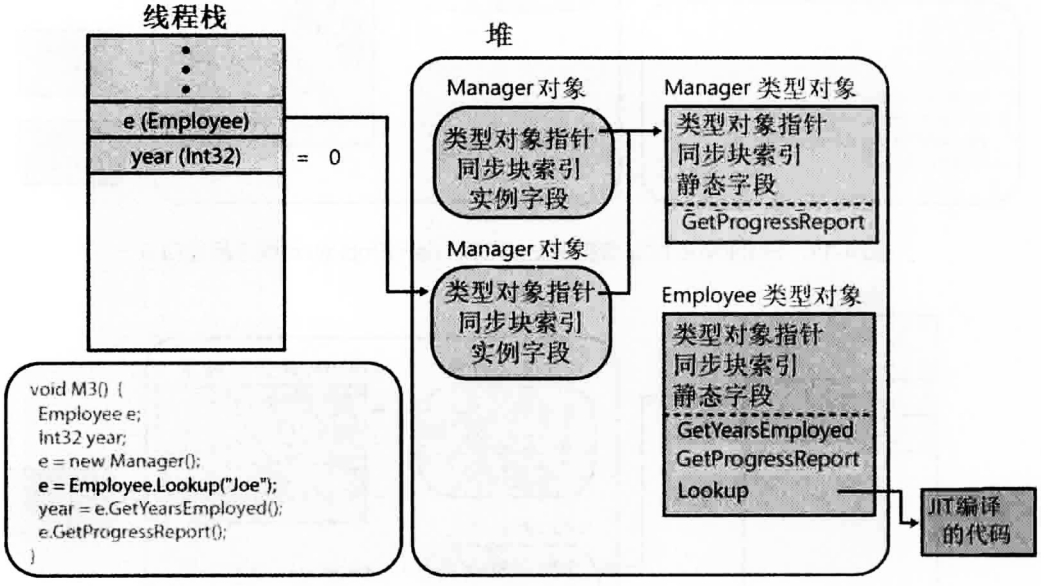


图 4-10　Employee 的静态方法 Lookup 为 Joe 分配并初始化 Manager 对象

然后，JIT 编译器在类型对象的方法表中查找引用了被调用方法的记录项，对方法进行 JIT 编译(如果需要的话)，再调用 JIT 编译好的代码。本例假定 **Employee** 的 **GetYearsEmployed** 方法返回 **5**，因为 Joe 已被公司雇用 5 年。这个整数保存到局部变量 **year** 中。这个操作的结果如图 4-11 所示。

M3 的下一行代码调用 **Employee** 的虚实例方法 **GetProgressReport**。调用虚实例方法时，JIT 编译器要在方法中生成一些额外的代码；方法每次调用都会执行这些代码。这些代码首先检查发出调用的变量，并跟随地址来到发出调用的对象。变量 **e** 当前引用的是代表“Joe”的 **Manager** 对象。然后，代码检查对象内部的“类型对象指针”成员，该成员指向对象的实际类型。然后，代码在类型对象的方法表中查找引用了被调用方法的记录项，对方法进行 JIT 编译(如果需要的话)，再调用 JIT 编译好的代码。由于目前 **e** 引用一个 **Manager** 对象，所以会调用 **Manager** 的 **GetProgressReport** 实现。这个操作的结果如图 4-12 所示。

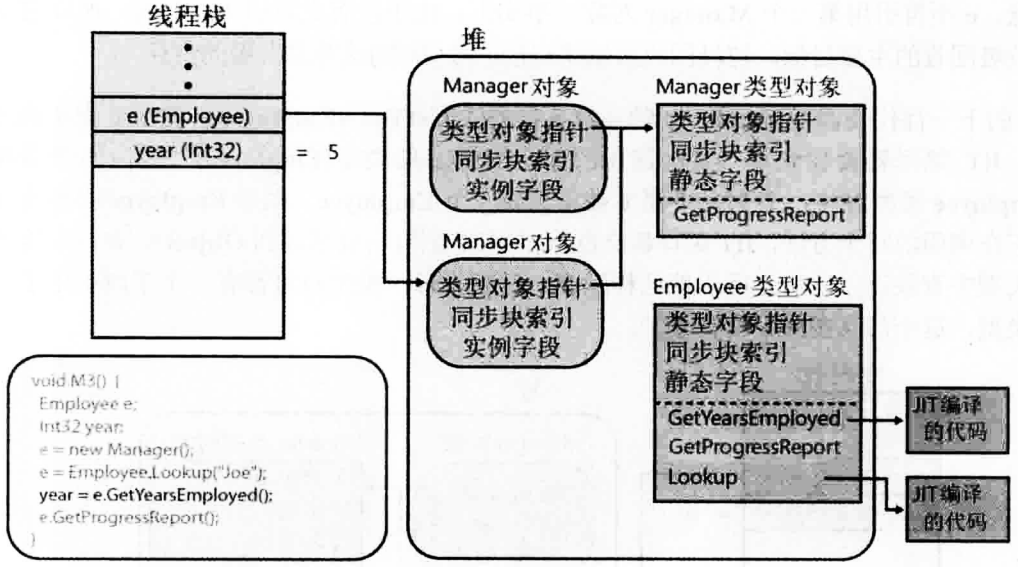


图 4-11　Employee 的非虚实例方法 GetYearsEmployed 调用后返回 5

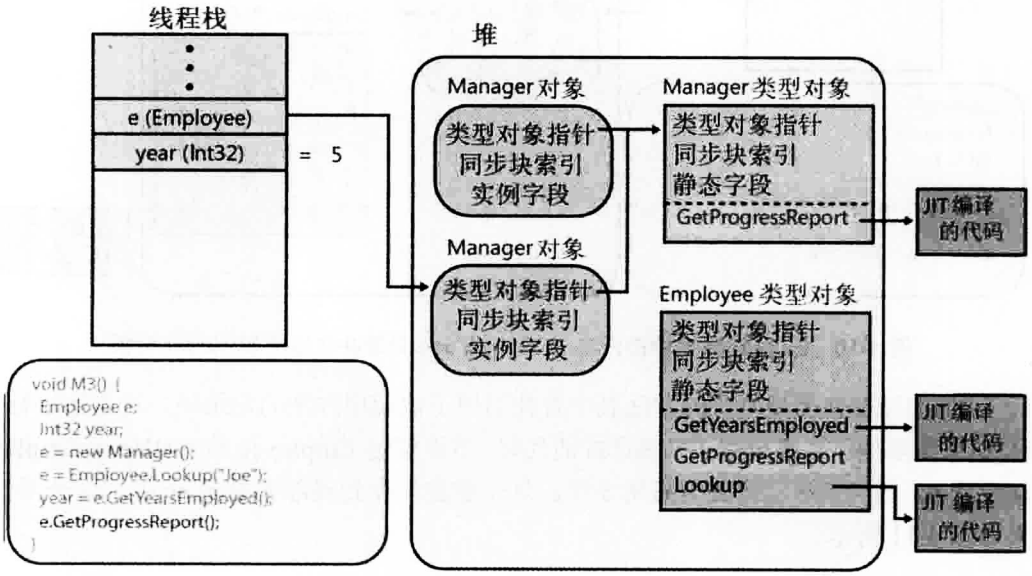


图 4-12　调用 Employee 的虚实例方法 GetProgressReport，最终执行 Manager 重写的版本

注意，如果 **Employee** 的 **Lookup** 方法发现 Joe 是 **Employee** 而不是 **Manager**，**Lookup** 会在内部构造一个 **Employee** 对象，它的类型对象指针将引用 **Employee** 类型对象。这样最终执行的就是 **Employee** 的 **GetProgressReport** 实现，而不是 **Manager** 的。

至此，我们已经讨论了源代码、IL 和 JIT 编译的代码之间的关系。还讨论了线程栈、实参、局部变量以及这些实参和变量如何引用托管堆上的对象。还知道对象含有一个指针指向对象的类型对象(类型对象中包含静态字段和方法表)。还讨论了 JIT 编译器如何决定静态方法、非虚实例方法以及虚实例方法的调用方式。理解这一切之后，可以深刻地认识 CLR 的

工作方式。以后在建构、设计和实现类型、组件以及应用程序时，这些知识会带来很大帮助。结束本章之前，让我们深入探讨一下 CLR 内部发生的事情。

注意 **Employee** 和 **Manager** 类型对象都包含“类型对象指针”成员。这是由于类型对象本质上也是对象。CLR 创建类型对象时，必须初始化这些成员。初始化成什么呢？你肯定会这样问。CLR 开始在一个进程中运行时，会立即为 MSCorLib.dll 中定义的 **System.Type** 类型创建一个特殊的类型对象。**Employee** 和 **Manager** 类型对象都是该类型的“实例”。因此，它们的类型对象指针成员会初始化成对 **System.Type** 类型对象的引用，如图 4-13 所示。

当然，**System.Type** 类型对象本身也是对象，内部也有“类型对象指针”成员。这个指针指向什么？它指向它本身，因为 **System.Type** 类型对象本身是一个类型对象的“实例”。现在，我们总算理解了 CLR 的整个类型系统及其工作方式。顺便说一句，**System.Object** 的 **GetType** 方法返回存储在指定对象的“类型对象指针”成员中的地址。也就是说，**GetType** 方法返回指向对象的类型对象的指针。这样就可判断系统中任何对象(包括类型对象本身)的真实类型。

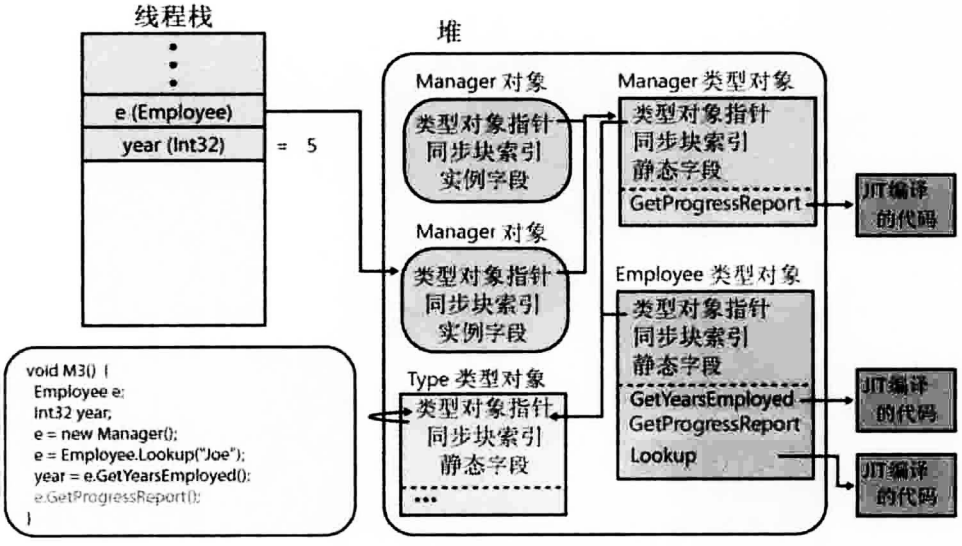
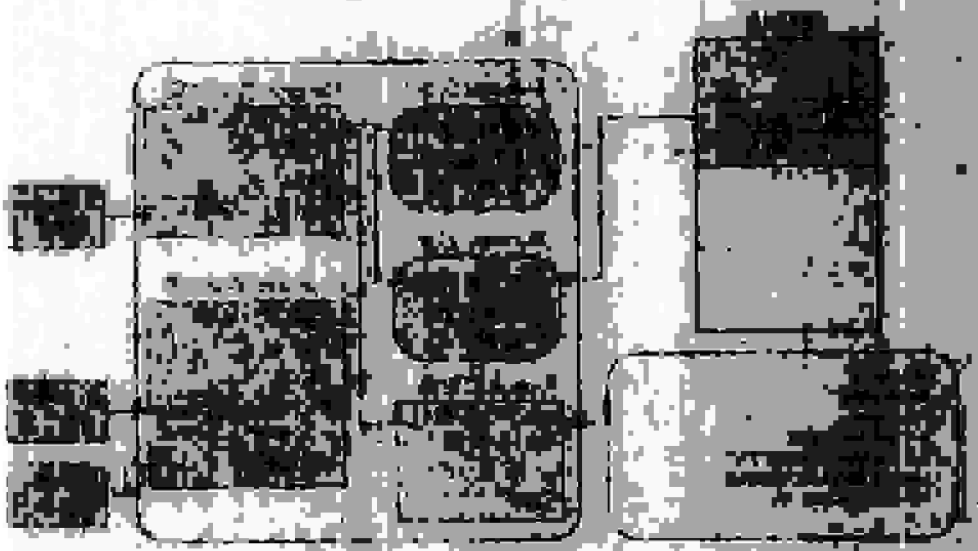


图 4-13　Employee 和 Manager 类型对象是 System.Type 类型的实例



第 5 章　基元类型、引用类型和值类型

本章内容：

- 编程语言的基元类型
- 引用类型和值类型
- 值类型的装箱和拆箱
- 对象哈希码
- **dynamic** 基元类型

本章将讨论 Microsoft .NET Framework 开发人员经常要接触的各种类型。所有开发人员都应熟悉这些类型的不同行为。我首次接触.NET Framework 时没有完全理解基元类型、引用类型和值类型的区别，造成在代码中不知不觉引入 bug 和性能问题。通过解释类型之间的区别，希望开发人员能避免我所经历的麻烦，同时提高编码效率。

5.1　编程语言的基元类型

某些数据类型如此常用，以至于许多编译器允许代码以简化语法来操纵它们。例如，可用以下语法分配一个整数：

```
System.Int32 a = new System.Int32();
```

但你肯定不愿意用这种语法声明并初始化整数，它实在是太繁琐了。幸好，包括 C#在内的许多编译器都允许换用如下所示的语法：

```
int a = 0;
```

这种语法不仅增强了代码可读性，生成的 IL 代码还与使用 **System.Int32** 生成的 IL 代码完全一致。编译器直接支持的数据类型称为**基元类型**(primitive type)[①]。基元类型直接映射到 Framework 类库(FCL)中存在的类型。例如，C#的 **int** 直接映射到 **System.Int32** 类型。因此，以下 4 行代码都能正确编译，并生成完全相同的 IL：

---

① MSDN 文档将 primitive type 翻译成“基元类型”，而不是容易使人混淆的“基本类型”。——译注

```
int             a = 0;                          // 最方便的语法
System.Int32    a = 0;                          // 方便的语法
int             a = new int();                  // 不方便的语法
System.Int32    a = new System.Int32()          // 最不方便的语法
```

表 5-1 列出的 FCL 类型在 C#中都有对应的基元类型。只要是符合公共语言规范(CLS)的类型，其他语言都提供了类似的基元类型。但是，不符合 CLS 的类型语言就不一定要支持了。

表 5-1 C#基元类型与对应的 FCL 类型

C#基元类型	FCL 类型	符合 CLS	说明
sbyte	**System.SByte**	否	有符号 8 位值
byte	**System.Byte**	是	无符号 8 位值
short	**System.Int16**	是	有符号 16 位值
ushort	**System.UInt16**	否	无符号 16 位值
int	**System.Int32**	是	有符号 32 位值
uint	**System.UInt32**	否	无符号 32 位值
long	**System.Int64**	是	有符号 64 位值
ulong	**System.UInt64**	否	无符号 64 位值
char	**System.Char**	是	16 位 Unicode 字符(**char** 不像在非托管 C++中那样代表一个 8 位值)
float	**System.Single**	是	IEEE 32 位浮点值
double	**System.Double**	是	IEEE 64 位浮点值
bool	**System.Boolean**	是	**true/false** 值
decimal	**System.Decimal**	是	128 位高精度浮点值，常用于不容许舍入误差的金融计算。128 位中，1 位是符号，96 位是值本身(N)，8 位是比例因子(k)。**decimal** 实际值是$\pm N \times 10^k$，其中 $-28 <= k <= 0$。其余位没有使用
string	**System.String**	是	字符数组
object	**System.Object**	是	所有类型的基类型
dynamic	**System.Object**	是	对于 CLR，**dynamic** 和 **object** 完全一致。但 C#编译器允许使用简单的语法让 **dynamic** 变量参与动态调度。详情参见本章最后的 5.5 节“**dynamic** 基元类型”

从另一个角度，可认为 C#编译器自动假定所有源代码文件都添加了以下 **using** 指令(参考第 4 章)：

```
using sbyte  = System.SByte;
using byte   = System.Byte;
using short  = System.Int16;
using ushort = System.UInt16;
using int    = System.Int32;
using uint   = System.UInt32;
...
```

C#语言规范称：“从风格上说，最好是使用关键字，而不是使用完整的系统类型名称。”我不同意语言规范：我情愿使用 FCL 类型名称，完全不用基元类型名称。事实上，我希望编译器根本不提供基元类型名称，而是强迫开发人员使用 FCL 类型名称。理由如下。

1. 许多开发人员纠结于是用 **string** 还是 **String**。由于 C#的 **string**(一个关键字)直接映射到 **System.String**(一个 FCL 类型)，所以两者没有区别，都可使用。类似地，一些开发人员说应用程序在 32 位操作系统上运行，**int** 代表 32 位整数；在 64 位操作系统上运行，**int** 代表 64 位整数。这个说法完全错误。C#的 **int** 始终映射到 **System.Int32**，所以不管在什么操作系统上运行，代表的都是 32 位整数。如果程序员习惯在代码中使用 **Int32**，像这样的误解就没有了。

2. C#的 **long** 映射到 **System.Int64**，但在其他编程语言中，**long** 可能映射到 **Int16** 或 **Int32**。例如，C++/CLI 就将 **long** 视为 **Int32**。习惯于用一种语言写程序的人在看用另一种语言写的源代码时，很容易错误理解代码意图。事实上，大多数语言甚至不将 **long** 当作关键字，根本不编译使用了它的代码。

3. FCL 的许多方法都将类型名作为方法名的一部分。例如，**BinaryReader** 类型的方法包括 **ReadBoolean**，**ReadInt32**，**ReadSingle** 等；而 **System.Convert** 类型的方法包括 **ToBoolean**，**ToInt32**，**ToSingle** 等。以下代码虽然语法没问题，但包含 **float** 的那一行显得很别扭，无法一下子判断该行的正确性：

```
BinaryReader br = new BinaryReader(...);
float val    = br.ReadSingle();    // 正确，但感觉别扭
Single val   = br.ReadSingle();    // 正确，感觉自然
```

4. 平时只用 C#的许多程序员逐渐忘了还可以用其他语言写面向 CLR 的代码，“C#主义”逐渐入侵类库代码。例如，Microsoft 的 FCL 几乎是完全用 C#写的，FCL 团队向库中引入了像 **Array** 的 **GetLongLength** 这样的方法。该方法返回 **Int64** 值。这种值在 C#中确实是 **long**，但在其他语言(比如 C++/CLI)中不是。另一个例子是 **System.Linq.Enumerable** 的 **LongCount** 方法。

考虑到所有这些原因，本书坚持使用 FCL 类型名称。

在许多编程语言中，以下代码都能正确编译并运行：

```
Int32 i = 5; // 32 位值
Int64 l = i; // 隐式转型为 64 位值
```

但根据第 4 章对类型转换的讨论，你或许认为上述代码无法编译。毕竟，**System.Int32** 和 **System.Int64** 是不同的类型，相互不存在派生关系。但事实上，你会欣喜地发现 C#编译器正确编译了上述代码，运行起来也没有问题。这是为什么呢？原因是 C#编译器非常熟悉基元类型，会在编译代码时应用自己的特殊规则。也就是说，编译器能识别常见的编程模式，并生成必要的 IL，使写好的代码能像预期的那样工作。具体地说，C#编译器支持与类型转换、字面值[①]以及操作符有关的模式。接着的几个例子将对它们进行演示。

首先，编译器能执行基元类型之间的隐式或显式转型，例如：

```
Int32 i = 5;                    // 从 Int32 隐式转型为 Int32
```

① 即 literal，也称为直接量或文字常量。本书将采用“字面值”这一译法。——译注

```
Int64 l = i;                // 从 Int32 隐式转型为 Int64
Single s = i;               // 从 Int32 隐式转型为 Single
Byte b = (Byte) i;          // 从 Int32 显式转型为 Byte
Int16 v = (Int16) s;        // 从 Single 显式转型为 Int16
```

只有在转换“安全”的时候，C#才允许隐式转型。所谓“安全”，是指不会发生数据丢失的情况，比如从 **Int32** 转换为 **Int64**。但如果可能不安全，C#就要求显式转型。对于数值类型，“不安全”意味着转换后可能丢失精度或数量级。例如，**Int32** 转换为 **Byte** 要求显式转型，因为大的 **Int32** 数字可能丢失精度；**Single** 转换为 **Int16** 也要求显式转型，因为 **Single** 能表示比 **Int16** 更大数量级的数字。

注意，不同编译器可能生成不同代码来处理这些转型。例如，将值为 6.8 的 **Single** 转型为 **Int32**，有的编译器可能生成代码对其进行截断(向下取整)，最终将 6 放到一个 **Int32** 中；其他编译器则可能将结果向上取整为 7。顺便说一句，C#总是对结果进行截断，而不进行向上取整。要了解 C#对基元类型进行转型时的具体规则，请参见 C#语言规范的“转换”一节。

除了转型，基本类型还能写成字面值 (literal)。字面值可被看成是类型本身的实例，所以可像下面为实例(123 和 456)调用实例方法：

```
Console.WriteLine(123.ToString() + 456.ToString());      // "123456"
```

另外，如果表达式由字面值构成，编译器在编译时就能完成表达式求值，从而增强应用程序性能：

```
Boolean found = false;          // 生成的代码将 found 设为 0
Int32 x = 100 + 20 + 3;         // 生成的代码将 x 设为 123
String s = "a " + "bc";         // 生成的代码将 s 设为"a bc"
```

最后，编译器知道如何和以什么顺序解析代码中的操作符(比如+，-，*，/，%，&，^，|，==，!=，>，<，>=，<=，<<，>>，~，!，++，--等)：

```
Int32 x = 100;                          // 赋值操作符
Int32 y = x + 23;                       // 加和赋值操作符
Boolean lessThanFifty = (y < 50);       // 小于和赋值操作符
```

checked 和 unchecked 基元类型操作

对基元类型执行的许多算术运算都可能造成溢出：

```
Byte b = 100;
b = (Byte) (b + 200); // b 现在包含 44 (或者十六进制值 2C)
```



重要提示 执行上述算术运算时，第一步要求所有操作数都扩大为 32 位值(或者 64 位值，如果任何操作数需要超过 32 位来表示的话)。所以 b 和 200(两个都不超过 32 位)首先转换成 32 位值，然后加到一起。结果是一个 32 位值(十进制 300，或十六进制 12C)。该值在存回变量 b 前必须转型为 Byte。C#不隐式执行这个转型操作，这正是第二行代码需要强制转型 Byte 的原因。

溢出大多数时候是我们不希望的。如果没有检测到这种溢出，会导致应用程序行为失常。但极少数时候(比如计算哈希值或者校验和)，这种溢出不仅可以接受，还是我们希望的。

不同语言处理溢出的方式不同。C 和 C++不将溢出视为错误，允许值回滚(wrap)[①]；应用程序将“若无其事”地运行。相反，Microsoft Visual Basic 总是将溢出视为错误，并在检测到溢出时抛出异常。

CLR 提供了一些特殊的 IL 指令，允许编译器选择它认为最恰当的行为。CLR 有一个 **add** 指令，作用是将两个值相加，但不执行溢出检查。还有一个 **add.ovf** 指令，作用也是将两个值相加，但会在发生溢出时抛出 **System.OverflowException** 异常。除了用于加法运算的 IL 指令，CLR 还为减、乘和数据转换提供了类似的 IL 指令，分别是 **sub/sub.ovf**、**mul/mul.ovf** 和 **conv/conv.ovf**。

C#允许程序员自己决定如何处理溢出。溢出检查默认关闭。也就是说，编译器生成 IL 代码时，将自动使用加、减、乘以及转换指令的无溢出检查版本。结果是代码能更快地运行——但开发人员必须保证不发生溢出，或者代码能预见到溢出。

让 C#编译器控制溢出的一个办法是使用**/checked+**编译器开关。该开关指示编译器在生成代码时，使用加、减、乘和转换指令的溢出检查版本。这样生成的代码在执行时会稍慢一些，因为 CLR 会检查这些运算，判断是否发生溢出。如果发生溢出，CLR 抛出 **OverflowException** 异常。

除了全局性地打开或关闭溢出检查，程序员还可在代码的特定区域控制溢出检查。C#通过 **checked** 和 **unchecked** 操作符来提供这种灵活性。下面是使用了 **unchecked** 操作符的例子：

```
UInt32 invalid = unchecked((UInt32) (-1));   // OK
```

下例则使用了 **checked** 操作符：

```
Byte b = 100;
b = checked((Byte) (b + 200));   // 抛出 OverflowException 异常
```

在这个例子中，**b** 和 **200** 首先转换成 32 位值，然后加到一起，结果是 300。然后，因为显式转型的存在，**300** 被转换成一个 **Byte**，这造成 **OverflowException** 异常。**Byte** 在 **checked** 操作符外部转型则不会发生异常：

```
b = (Byte) checked(b + 200);   // b 包含 44；不会抛出 OverflowException 异常
```

除了 **checked** 和 **unchecked** 操作符，C#还支持 **checked** 和 **unchecked** 语句，它们造成一个块中的所有表达式都进行或不进行溢出检查：

```
checked {                          // 开始 checked 块
   Byte b = 100;
   b = (Byte) (b + 200);           // 该表达式会进行溢出检查
}                                  // 结束 checked 块
```

① 所谓“回滚”，是指一个值超过了它的类型所允许的最大值，从而“回滚”到一个非常小的、负的或者未定义的值。wrap 是 wrap-around 的简称。——译注