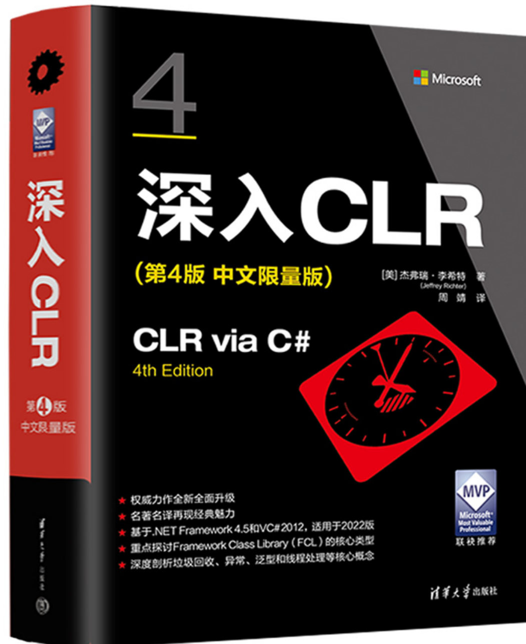
《深入 CLR》（原 CLR via C#）2024 修订版

(美) Jeffrey Richter　著

周靖 译（https://bookzhou.com）



中文试读版 1-13 章和译者后记（全书 1/3 内容），翻译原稿，仅供参考，

获取更多精彩内容，请购买正式版：京东购买>> 淘宝购买>>

配套资源和试读下载：　ys168 网盘>>　百度网盘>>

访问中文版主页，获取最新资讯

清华大学出版社

北 京

CLR via C# (第 4 版)

(美) Jeffrey Richter　著

周 靖　译

推荐序

大家好，我们又见面了。谁预见到了今天啊？哈哈，我就预见到了！一旦步入婚姻的殿堂，就相当于过上了“土拨鼠日”。如果还没有看过那部电影，就去看看吧。看了之后，就会明白为什么自己老是犯同样的错误。当 Jeff 说他不再写书的时候，我就知道这是一个“瘾君子”开的空头支票。Jeff 不可能停止写书。就在今天，我们还在说起他“绝对”不可能写的另一本书呢(实际情况是，有一章已经在写了)。写书已深入到他骨子里面去了。千里马生来就是要奔跑的，Jeff 生来就是要写作的。

Jeff 太有规律了。他就是离不开硬盘里那些小小的 0 和 1。忽视它们是不可能的。凌晨 3 点，我们睡梦正酣的时候，Jeff 的生物钟就在催促他起床了(巧合的是，我们 4 岁大的小儿子也恰好在这个时候爬到我们的床上。爷儿俩的行为模式我都理解不了啊)。一股神秘的力量促使 Jeff 的大脑自动释放出解决方案、头脑风暴和臭虫之类的东西，迫使他跑到办公室把这些宝贝从脑袋里倒腾出来。而我们呢，则安心地翻个身继续呼呼大睡，知道 Jeff 会解决那些问题——就像一个神秘的网络超级英雄，防止线程又成为薄弱环节[①]。

但积累这些知识供自己使用，这对 Jeff 来说远远不够。好东西不该独享。所以必须把它们传播开来，必须把它们写下来。知识就像电波，有心人能接收得到。这就是他为你所做的，亲爱的读者，是他热爱微软技术的明证。

本书还有另一层意义在里面。Jeff 每次绕着太阳“公转”一圈，都会变得更老一些。经过多年的积累，他也学着“向后看”了。由于看事情的方式变得更成熟，所以他重写了讲反射的那一章。或许你也应该跟他一起回顾一下这个主题。可以学到怎样让代码自个儿询问关于代码的事儿，进而更深入地思考为什么反射要那样工作。穿上便服，找一把舒服的椅子坐下，花些时间想想自己的代码以及它们生命中更深层次的意义。

本书还讲了一样有趣的东西，就是异步/等待[②]。和我老公以前鼓捣过一阵子的 `AsyncEnumerator` 相比，这个东西显然进步了不少。哎，我还以为今后离不开它了呢！事实上，虽然他跟我讲了好多次 `AsyncEnumerator`，但这个东西根本就没有在我的脑子里“阻塞”嘛！于是我窃想，如果知道什么是 enumerator 的话，也许就能明白他讲的是啥了。于是我查了一下维基百科，发现 enumerator 是“人口普查员”的意思。这一章难道是讲人口普查员怎样协调工作的事儿？那也太浪费纳税人的钱了吧！不过，我相信它在计算机里面的意义比我查到的好。Jeff 和微软的团队一起工作，将异步/等待打磨得很完美。你现在

[①] saving the thread from becoming just another loose end，直译就是“防止线头儿又松了”。——译注

[②] `async` 和 `await` 是 C#的两个关键字，允许用顺序编程模型执行异步操作。——译注

通过这本书就能舒舒服服地享受他们的成果了。我建议你好好读一下。嗯，要顺着读。[1]

本书的另一个重头戏是我感觉最兴奋的。希望你们都来看看关于 WinRT 的内容。这个术语太书呆子气了，我的理解就是“马上为我无敌帅气的平板搞一些很酷的应用出来！”你猜得没错，新的 Windows Runtime 就是围绕无敌帅气的触摸屏展开的。孩子喜欢小鸟飞向小猪，我则喜欢跟鲜花有关的东西，而你完全可以用平板做其他事情。没有做不到，只有想不到！去折腾出一些“奇思妙想非常牛掰”(Wonderful Innovative Nifty Really Touchy，WinRT)的东西出来。就当是为了我，好好看看这一章。否则的话，我对 Jeff 和他无休止的写作事业可能真的会失去耐心，会把他关到一间只有针头线脑[2]而且没有电的小黑屋里面。你们程序员看着办吧，是用 WinRT 写一些很酷的应用，还是再也没有 Jeff 的新书看！

总之，在你们的力挺之下，Jeff 的又一部大作诞生了。我们的家庭貌似又可以回归正常状态了。但真的正常吗？或许他不停写书才真的正常吧。

让我们耐心等待下一本书的神秘召唤。

Kristin Trace(Jeffrey 的妻子)

2012 年 10 月

救命！Jeff 要被关小黑屋了！

[1] Kristin 用“sequentially”一词来吐槽顺序编程模型。——译注

[2] Kristin 又在吐槽“thread”了。——译注

前言

1999 年 10 月，微软的一些人首次向我展示了 Microsoft .NET Framework、公共语言运行时(Common Language Runtime，CLR)和 C#编程语言。看到这一切时，我惊呆了，我知道我写软件的方式要发生非常大的变化了。他们请我为团队做一些顾问工作，我当即同意了。刚开始，我以为.NET Framework 是 Win32 API 和 COM 上的一个抽象层。但随着我投入越来越多的时间研究，我意识到它是一个更宏伟的项目。某种程度上，它是自己的操作系统。它有自己的内存管理器、自己的安全系统、自己的文件加载器、自己的错误处理机制、自己的应用程序隔离边界(AppDomain)、自己的线程处理模型等。本书解释了所有这些主题，帮你为这个平台高效地设计和实现应用程序和组件。

我写这本书是 2012 年 10 月，距离首次接触.NET Framework 和 C#正好 13 年。13 年来，我以 Microsoft 顾问身份开发过各式各样的应用程序，为.NET Framework 本身也贡献良多。作为我自己公司 Wintellect(*http://Wintellect.com*)[①]的合伙人，我还要为大量客户工作，帮他们设计、调试、优化软件以及解决使用.NET Framework 时遇到的问题。正是因为有了这些资历，所以我才知道如何用.NET Framework 进行高效率编程。贯穿本书所有主题，你都会看到我的经验之谈。

本书面向的读者

本书旨在解释如何为.NET Framework 开发应用程序和可重用的类。具体地说，我要解释 CLR 的工作原理及其提供的功能，还要讨论 Framework Class Library(FCL)的各个部分。没有一本书能完整地解释 FCL——其中含有数以千计的类型，而且这个数字正在以惊人速度增长。所以，我准备将重点放在每个开发人员都需要注意的核心类型上面。另外，虽然不会专门讲 Windows 窗体、Windows Presentation Foundation(WPF)、Microsoft Silverlight、XML Web 服务、Web 窗体、Microsoft ASP.NET MVC、Windows Store 应用等，但本书描述的技术适用于所有这些应用程序类型。

[①] 译注：Wintellect 目前已被 Atmosera 收购并更名为 WintellectNOW，目前提供微软全系列产品(包括 Azure 云计算服务和 AI 等)的按需培训服务。新网址是 *https://www.wintellectnow.com/*。

本书围绕 Microsoft Visual Studio 2012/2013，.NET Framework 4.5.*x* 和 C# 5.0 展开。[①]由于微软在发布这些技术的新版本时，会试图保持很大程度的向后兼容性，所以本书描述的许多内容也适合之前的版本。所有示例代码都用 C#编程语言写成。但由于 CLR 可由许多编程语言使用，所以本书内容也适合非 C#程序员。

注意：本书配套代码和勘误请从译者主页下载：*https://bookzhou.com*。

我和我的编辑们进行了艰苦卓绝的工作，试图为你提供最准确、最新、最深入、最容易阅读和理解、没有错误的信息。但是，即便有如此完美的团队协作，疏漏和错误也在所难免。如果你发现了本书的任何错误或者想提出一些建设性的意见，请发送邮件给本书中文版编辑 *coo@netease.com*。

致谢

没有别人的帮助和技术援助，我是不可能写好这本书的。尤其要感谢我的家人。写好一本书所投入的时间和精力无法衡量。我只知道，没有我的妻子 Kristin 和两个儿子 Aidan 和 Grant 的支持，根本不可能有这本书的面世。多少次想花些时间一家人小聚，都因为本书而放弃。现在，本书总算告一段落，终于有时间做自己喜欢做的事情了。

本书的修订得到了一些“高人”的协助。.NET Framework 团队的一些人(其中许多都是我的朋友)审阅了其中的章节，我和他们进行了许多发人深省的对话。Christophe Nasarre 参与了我几本书的出版，在审阅本书并确保我能以最恰当的方式来表达的过程中，表现出了非凡的才能。他对本书的品质有至关重要的影响。和往常一样，我和 Microsoft Press 的团队进行了令人愉快的合作。特别感谢 Ben Ryan，Devon Musgrave 和 Carol Dillingham。另外，感谢 Susie Carr 和 Candace Sinclair 提供的编辑与制作支持。

勘误和支持

我们尽最大努力保证本书的准确性。英文版勘误或更改会添加到以下网页：

http://www.oreilly.com/catalog/errata.csp?isbn=0790145353665

http://go.microsoft.com/FWLink/?Linkid=266601

[①] 原书虽然基于 Visual Studio 2012/2013，.NET Framework 4.5.x 和 C# 5.0，但在目前最新的 Visual Studio 2022(.NET Framework 4.8.*x* 和 C# 10)上，除了关于 AppDomain 的部分内容，本书绝大多数内容都适合当前的情况。毕竟，整本书讲的都是基础。所有“语法糖”在 ILDasm.exe 的面前，都是一样的。——译注

如果发现未列出的错误，可通过相同的网页报告。

如需其他支持，请致函 Microsoft Press Book Support 部门：

mspinput@microsoft.com

注意，上述邮件地址不提供产品支持。

最后，本书中文版的勘误和资源下载请访问译者博客，中文版已反映了英文版到目前(2023 年)为止的所有勘误：

https://bookzhou.com

作者简介

Jeffrey Richter，Wintellect 联合创始人。数十年如一日痴迷于 Windows 和.NET 的大师，数十年以来影响了若干代程序员的灵魂人物，经典著作《Windows 核心编程》和《CLR via C#》系列版本的缔造者。他崇尚大道至简，注重效率与实用性，尤其热爱化繁为简，去芜存菁。值得一提的是，他特别擅长授人以渔，能够以通俗易懂、条理清晰的方式将普通读者望而却步的关键概念讲得透彻，讲得精彩。Jeffrey 出版过 12 本 Windows / .NET 畅销书，曾经担任过 *MSDN Magazine* 特约编辑、Win32 Q&A 专栏作者、.NET Q&A 专栏作者和 *Concurrent Affairs*(关于并发那些事儿)专栏作者。他还在很多全球性的商业研讨会上发表演讲。作为懂 Windows 和.NET 的人，他从 1990 年开始就以顾问身份为微软提供服务——他写的代码先后被微软的很多产品广泛采用。此外，他还曾经为 AT&T、IBM、英特尔、梦工厂、通用和惠普等提供过咨询服务。Jeffrey 爱好广泛，他拥有飞机驾照和直升机驾照，是国际魔法师协会成员。他爱好音乐(尤其是 20 世纪 70 年代的爵士乐和前卫摇滚乐)、击鼓、火车模型和空手道。他还喜欢旅游和戏剧。目前，他与爱妻 Kristin 及爱子 Aidan 和 Grant 居住在华盛顿州的柯克兰。

目录

推荐序 **3**

前言 **5**

本书面向的读者 5

致谢 6

勘误和支持 6

作者简介 **8**

目录 **9**

第 I 部分 CLR 基础 **25**

第 1 章 CLR 的执行模型 **27**

1.1 将源代码编译成托管模块 27

1.2 将托管模块合并成程序集 31

1.3 加载公共语言运行时(CLR) 33

1.4 执行程序集的代码 37

1.4.1 IL 和验证 43

1.4.2 不安全的代码 44

1.5 本机代码生成器：NGen.exe 45

1.6 Framework 类库简介 49

1.7 通用类型系统 52

1.8 公共语言规范 54

1.9 与非托管代码的互操作性 59

第 2 章 生成、打包、部署和管理应用程序及类型 **61**

2.1 .NET Framework 部署目标 61

2.2 将类型生成到模块中 63

响应文件 64
2.3 元数据概述 66
2.4 将模块合并成程序集 73
2.4.1 使用 Visual Studio IDE 将程序集添加到项目中 79
2.4.2 使用程序集链接器 81
2.4.3 为程序集添加资源文件 82
2.5 程序集版本资源信息 83
版本号 88
2.6 语言文化 89
2.7 简单应用程序部署(私有部署的程序集) 90
2.8 简单管理控制(配置) 91
第 3 章 共享程序集和强命名程序集 95
3.1 两种程序集，两种部署 96
3.2 为程序集分配强名称 97
3.3 全局程序集缓存 101
3.4 在生成的程序集中引用强命名程序集 104
3.5 强命名程序集能防篡改 105
3.6 延迟签名 106
3.7 私有部署强命名程序集 108
3.8 “运行时”如何解析类型引用 109
3.9 高级管理控制(配置) 113
发布者策略控制 115
第II部分 设计类型 118
第 4 章 类型基础 119
4.1 所有类型都从 System.Object 派生 119

4.2 类型转换 121
使用 C#的 is 和 as 操作符来转型 124
4.3 命名空间和程序集 125
4.4 在运行时的相互关系 129
第 5 章 基元类型、引用类型和值类型 139
5.1 编程语言的基元类型 139
checked 和 unchecked 基元类型操作 142
5.2 引用类型和值类型 148
5.3 值类型的装箱和拆箱 154
5.3.1 使用接口更改已装箱值类型中的字段(以及为什么不应该这样做) 166
5.3.2 对象相等性和同一性 169
5.4 对象哈希码 172
5.5 dynamic 基元类型 174
第 6 章 类型和成员基础 181
6.1 类型的各种成员 181
6.2 类型的可见性 185
友元程序集 185
6.3 成员的可访问性 186
6.4 静态类 188
6.5 分部类、结构和接口 190
6.6 组件、多态和版本控制 191
6.6.1 CLR 如何调用虚方法、属性和事件 193
6.6.2 合理使用类型的可见性和成员的可访问性 197
6.6.3 对类型进行版本控制时的虚方法的处理 199
第 7 章 常量和字段 205

7.1 常量 205

7.2 字段 207

第 8 章 方法 211

8.1 实例构造器和类(引用类型) 211

8.2 实例构造器和结构(值类型) 215

8.3 类型构造器 217

8.4 操作符重载方法 221

操作符和编程语言互操作性 223

8.5 转换操作符方法 224

8.6 扩展方法 228

8.6.1 规则和指导原则 230

8.6.2 用扩展方法扩展各种类型 231

8.6.3 ExtensionAttribute 类 233

8.7 分部方法 234

规则和指导原则 237

第 9 章 参数 239

9.1 可选参数和命名参数 239

9.1.1 规则和指导原则 240

9.1.2 DefaultParameterValue 和 Optional 特性 242

9.2 隐式类型的局部变量 242

9.3 以传引用的方式向方法传递参数 244

9.4 向方法传递可变数量的参数 250

9.5 参数和返回类型的设计规范 252

9.6 常量性 254

第 10 章 属性 255

10.1 无参属性 255
10.1.1 自动实现的属性 259
10.1.2 合理定义属性 260
10.1.3 对象和集合初始化器 262
10.1.4 匿名类型 264
10.1.5 System.Tuple 类型 267
10.2 有参属性 269
10.3 调用属性访问器方法时的性能 274
10.4 属性访问器的可访问性 275
10.5 泛型属性访问器方法 275
第 11 章 事件 277
11.1 设计要公开事件的类型 278
11.1.1 第一步：定义类型来容纳所有需要发送给事件通知接收者的附加信息 279
11.1.2 第二步：定义事件成员 279
11.1.3 第三步：定义负责引发事件的方法来通知事件的登记对象 281
11.1.4 第四步：定义方法将输入转化为期望事件 283
11.2 编译器如何实现事件 283
11.3 设计侦听事件的类型 285
11.4 显式实现事件 287
第 12 章 泛型 291
12.1 FCL 中的泛型 296
12.2 泛型基础结构 297
12.2.1 开放类型和封闭类型 298
12.2.2 泛型类型和继承 300
12.2.3 泛型类型同一性 302

12.2.4 代码爆炸 303
12.3 泛型接口 303
12.4 泛型委托 305
12.5 委托和接口的逆变和协变泛型类型实参 307
12.6 泛型方法 309
泛型方法和类型推断 310
12.7 泛型和其他成员 312
12.8 可验证性和约束 313
12.8.1 主要约束 315
12.8.2 次要约束 316
12.8.3 构造器约束 317
12.8.4 其他可验证性问题 318
第 13 章 接口 322
13.1 类和接口继承 322
13.2 定义接口 323
13.3 继承接口 324
13.4 关于调用接口方法的更多探讨 327
13.5 隐式和显式接口方法实现(幕后发生的事情) 328
13.6 泛型接口 330
13.7 泛型和接口约束 332
13.8 实现多个具有相同方法名和签名的接口 333
13.9 用显式接口方法实现来增强编译时类型安全性 335
13.10 谨慎使用显式接口方法实现 337
13.11 设计：基类还是接口 340
第Ⅲ部分 基本类型 342

第 14 章 字符、字符串和文本处理 343

14.1 字符 343

14.2 System.String 类型 346

14.2.1 构造字符串 347

14.2.2 字符串是不可变的 349

14.2.3 比较字符串 349

14.2.4 字符串留用 356

14.2.5 字符串池 358

14.2.6 检查字符串中的字符和文本元素 358

14.2.7 其他字符串操作 361

14.3 高效率构造字符串 362

14.3.1 构造 StringBuilder 对象 362

14.3.2 StringBuilder 的成员 363

14.4 获取对象的字符串表示：ToString 366

14.4.1 指定具体的格式和语言文化 366

14.4.2 将多个对象格式化成一个字符串 370

14.4.3 提供定制格式化器 371

14.5 解析字符串来获取对象：Parse 374

14.6 编码：字符和字节的相互转换 376

14.6.1 字符和字节流的编码和解码 381

14.6.2 Base-64 字符串编码和解码 382

14.7 安全字符串 383

第 15 章 枚举类型和位标志 387

15.1 枚举类型 387

15.2 位标志 393

15.3 向枚举类型添加方法 397

第 16 章 数组 399

16.1 初始化数组元素 401

16.2 数组转型 403

16.3 所有数组都隐式派生自 System.Array 406

16.4 所有数组都隐式实现 IEnumerable，ICollection 和 IList 407

16.5 数组的传递和返回 408

16.6 创建下限非零的数组 409

16.7 数组的内部工作原理 410

16.8 不安全的数组访问和固定大小的数组 414

第 17 章 委托 417

17.1 初识委托 417

17.2 用委托回调静态方法 420

17.3 用委托回调实例方法 421

17.4 委托揭秘 422

17.5 用委托回调多个方法(委托链) 425

17.5.1 C#对委托链的支持 430

17.5.2 取得对委托链调用的更多控制 431

17.6 委托定义不要太多(泛型委托) 433

17.7 C#为委托提供的简化语法 434

17.7.1 简化语法 1：不需要构造委托对象 435

17.7.2 简化语法 2：不需要定义回调方法(lambda 表达式) 435

17.7.3 简化语法 3：局部变量不需要手动包装到类中即可传给回调方法 439

17.8 委托和反射 442

第 18 章 定制特性 447

18.1 使用定制特性 447
18.2 定义自己的特性类 451
18.3 特性构造器和字段/属性数据类型 454
18.4 检测定制特性 455
18.5 两个特性实例的相互匹配 460
18.6 检测定制特性时不创建从 Attribute 派生的对象 463
18.7 条件特性类 466
第 19 章 可空值类型 468
19.1 C#对可空值类型的支持 470
19.2 C#的空接合操作符 473
19.3 CLR 对可空值类型的特殊支持 474
19.3.1 可空值类型的装箱 474
19.3.2 可空值类型的拆箱 476
19.3.3 通过可空值类型调用 GetType 476
19.3.4 通过可空值类型调用接口方法 476
第IV部分 核心机制 477
第 20 章 异常和状态管理 479
20.1 定义"异常" 479
20.2 异常处理机制 481
20.2.1 try 块 482
20.2.2 catch 块 482
20.2.3 finally 块 484
20.3 System.Exception 类 487
20.4 FCL 定义的异常类 491
20.5 抛出异常 495

20.6 定义自己的异常类 496
20.7 牺牲可靠性来换取开发效率 500
20.8 设计规范和最佳实践 508
20.8.1 善用 finally 块 509
20.8.2 不要什么都捕捉 510
20.8.3 得体地从异常中恢复 511
20.8.4 发生不可恢复的异常时回滚部分完成的操作——维持状态 513
20.8.5 隐藏实现细节来维系协定 514
20.9 未处理的异常 517
20.10 对异常进行调试 522
20.11 异常处理的性能问题 524
20.12 约束执行区域(CER) 528
20.13 代码协定 531
第 21 章 托管堆和垃圾回收 539
21.1 托管堆基础 539
21.1.1 从托管堆分配资源 540
21.1.2 垃圾回收算法 541
21.1.3 垃圾回收和调试 544
21.2 代：提升性能 547
21.2.1 垃圾回收触发条件 552
21.2.2 大对象 552
21.2.3 垃圾回收模式 553
21.2.4 强制垃圾回收 555
21.2.5 监视应用程序的内存使用 557
21.3 使用需要特殊清理的类型 558

21.3.1 使用包装了本机资源的类型 564

21.3.2 一个有趣的依赖性问题 570

21.3.3 GC 为本机资源提供的其他功能 571

21.3.4 终结的内部工作原理 575

21.3.5 手动监视和控制对象的生存期 577

第 22 章 CLR 寄宿和 AppDomain 585

22.1 CLR 寄宿 585

22.2 AppDomain 587

跨越 AppDomain 边界访问对象 590

22.3 卸载 AppDomain 602

22.4 监视 AppDomain 603

22.5 AppDomain FirstChance 异常通知 605

22.6 宿主如何使用 AppDomain 605

22.6.1 可执行应用程序 605

22.6.2 Microsoft Silverlight 富 Internet 应用程序 606

22.6.3 Microsoft ASP.NET 和 XML Web 服务应用程序 606

22.6.4 Microsoft SQL Server 607

22.6.5 更多的用法只局限于想象力 607

22.7 高级宿主控制 607

22.7.1 使用托管代码管理 CLR 608

22.7.2 写健壮的宿主应用程序 608

22.7.3 宿主如何拿回它的线程 609

第 23 章 程序集加载和反射 613

23.1 程序集加载 614

23.2 使用反射构建动态可扩展应用程序 618

23.3 反射的性能 620

23.3.1 发现程序集中定义的类型 621

23.3.2 类型对象的准确含义 621

23.3.3 构建 Exception 派生类型的层次结构 623

23.3.4 构造类型的实例 625

23.4 设计支持加载项的应用程序 627

23.5 使用反射发现类型的成员 630

23.5.1 发现类型的成员 630

23.5.2 调用类型的成员 635

23.5.3 使用绑定句柄减少进程的内存消耗 640

第 24 章 运行时序列化 643

24.1 序列化/反序列化快速入门 644

24.2 使类型可序列化 648

24.3 控制序列化和反序列化 651

24.4 格式化器如何序列化类型实例 654

24.5 控制序列化/反序列化的数据 656

要实现 ISerializable 但基类型没有实现怎么办？ 662

24.6 流上下文 663

24.7 类型序列化为不同类型，对象反序列化为不同对象 665

24.8 序列化代理 668

代理选择器链 671

24.9 反序列化对象时重写程序集/类型 673

第 25 章 与 WinRT 组件互操作 675

25.1 CLR 投射与 WinRT 组件类型系统规则 677

WinRT 类型系统的核心概念 677

25.2 框架投射 681

25.2.1 从.NET 代码中调用异步 WinRT API........ 682

25.2.2 WinRT 流和.NET 流之间的互操作........ 686

25.2.3 在 CLR 和 WinRT 之间传输数据块 688

25.3 用 C#定义 WinRT 组件 691

第V部分 线程处理 698

第 26 章 线程基础 699

26.1 Windows 为什么要支持线程........ 699

26.2 线程开销........ 700

26.3 停止疯狂........ 704

26.4 CPU 发展趋势 706

26.5 CLR 线程和 Windows 线程........ 707

26.6 使用专用线程执行异步的计算限制操作 708

26.7 使用线程的理由........ 710

26.8 线程调度和优先级........ 712

26.9 前台线程和后台线程........ 718

26.10 继续学习........ 720

第 27 章 计算限制的异步操作........721

27.1 CLR 线程池基础........ 723

27.2 执行简单的计算限制操作........ 723

27.3 执行上下文........ 725

27.4 协作式取消和超时 727

27.5 任务........ 731

27.5.1 等待任务完成并获取结果........ 732

27.5.2 取消任务........ 734

27.5.3 任务完成时自动启动新任务 735
27.5.4 任务可以启动子任务 737
27.5.5 任务内部揭秘 738
27.5.6 任务工厂 739
27.5.7 任务调度器 741
27.6 Parallel 的静态 For，ForEach 和 Invoke 方法 743
27.7 并行语言集成查询(PLINQ) 748
27.8 执行定时计算限制操作 751
太多的计时器，太难选择 753
27.9 线程池如何管理线程 754
27.9.1 设置线程池限制 754
27.9.2 如何管理工作者线程 755
第 28 章 I/O 限制的异步操作 757
28.1 Windows 如何执行 I/O 操作 757
28.2 C#的异步函数 761
28.3 编译器如何将异步函数转换成状态机 764
28.4 异步函数扩展性 768
28.5 异步函数和事件处理程序 771
28.6 FCL 的异步函数 772
28.7 异步函数和异常处理 774
28.8 异步函数的其他功能 774
28.9 应用程序及其线程处理模型 777
28.10 以异步方式实现服务器 780
28.11 取消 I/O 操作 780
28.12 有的 I/O 操作必须同步进行 782

FileStream 特有的问题 782

28.13 I/O 请求优先级 783

第 29 章 基元线程同步构造 786

29.1 类库和线程安全 788

29.2 基元用户模式和内核模式构造 789

29.3 用户模式构造 790

29.3.1 易变构造 791

29.3.2 互锁构造 796

29.3.3 实现简单的自旋锁 801

29.3.4 Interlocked Anything 模式 804

29.4 内核模式构造 806

29.4.1 Event 构造 810

29.4.2 Semaphore 构造 813

29.4.3 Mutex 构造 815

第 30 章 混合线程同步构造 819

30.1 一个简单的混合锁 819

30.2 自旋、线程所有权和递归 821

30.3 FCL 中的混合构造 824

30.3.1 ManualResetEventSlim 类和 SemaphoreSlim 类 824

30.3.2 Monitor 类和同步块 825

30.3.3 ReaderWriterLockSlim 类 831

30.3.4 OneManyLock 类 835

30.3.5 CountdownEvent 类 837

30.3.6 Barrier 类 837

30.3.7 线程同步构造小结 838

30.4 著名的双检锁技术 839

30.5 条件变量模式 844

30.6 异步的同步构造 846

30.7 并发集合类 851

译者后记 856

第 I 部分　CLR 基础

- 第 1 章　CLR 的执行模型
- 第 2 章　生成、打包、部署和管理应用程序及类型
- 第 3 章　共享程序集和强命名程序集

第 1 章 CLR 的执行模型

本章内容：

- 将源代码编译成托管模块
- 将托管模块合并成程序集
- 加载公共语言运行时(CLR)
- 执行程序集的代码
- 本机代码生成器：NGen.exe
- Framework 类库简介
- 通用类型系统
- 公共语言规范(CLS)
- 与非托管代码的互操作性

Microsoft .NET Framework 引入了许多新概念、技术和术语。本章概述了.NET Framework 是如何设计的，介绍了 Framework 包含的一些新技术，并定义了今后要用到的许多术语。除此之外，还要展示如何将源代码生成(build)为一个应用程序，或者生成为一组可重新分发的组件(文件)——这些组件(文件)中包含类型(类和结构等)。最后，本章解释了应用程序如何执行。

1.1 将源代码编译成托管模块

决定将.NET Framework 作为自己的开发平台之后，第一步便是决定要生成什么类型的应用程序或组件。假定你已完成了这个小细节。换言之，一切均已设计好，规范已经写好，可以着手开发了。

现在，必须决定要使用哪一种编程语言。这通常是一个艰难的抉择，因为不同的语言各有长处。例如，非托管 C/C++可对系统进行低级控制。可完全按自己的想法管理内存，必要时能方便地创建线程。另一方面，使用 Microsoft Visual Basic 可以快速生成 UI 应用程序，并且可以方便地控制 COM 对象和数据库。

顾名思义，**公共语言运行时**(Common Language Runtime，CLR)是一个可由多种编程语言使

用的“运行时”。[1]CLR 的核心功能(比如内存管理、程序集加载、安全性、异常处理和线程同步)可由面向 CLR 的所有语言使用。例如，“运行时”使用异常来报告错误；因此，面向它的任何语言都能通过异常来报告错误。另外，“运行时”允许创建线程，所以面向它的任何语言都能创建线程。

事实上，在运行时，CLR 根本不介意开发人员用哪一种语言写源代码。这意味着在选择编程语言时，应选择最容易表达自己意图的语言。可以使用任何编程语言开发代码，只要编译器是面向 CLR 的。

既然如此，不同编程语言的优势何在呢？事实上，可以将编译器视为语法检查器和“正确代码”分析器。它们检查源代码，确定你写的一切都有意义，并输出对你的意图进行描述的代码。不同编程语言允许用不同的语法来开发。不要低估这个选择的价值。例如，对于数学或金融应用程序，使用 APL 语法来表达自己的意图，相较于使用 Perl 语法来表达同样的意图，可以节省许多开发时间。

微软创建好了几个面向“运行时”的语言编译器，其中包括：C++/CLI、C#(发音是“C sharp”)、Visual Basic、F#(发音是“F sharp”)、Iron Python、Iron Ruby 以及一个“中间语言”(Intermediate Language，IL)汇编器。除了微软，另一些公司、学院和大学也创建了自己的编译器，也能面向 CLR 生成代码。我所知道的有针对下列语言的编译器：Ada，APL，Caml，COBOL，Eiffel，Forth，Fortran，Haskell，Lexico，LISP，LOGO，Lua，Mercury，ML，Mondrian，Oberon，Pascal，Perl，PHP，Prolog，RPG，Scheme，Smalltalk 和 Tcl/Tk。

图 1-1 展示了编译源代码文件的过程。如图所示，可以使用支持 CLR 的任何语言创建源代码文件，然后用对应的编译器检查语法和分析源代码。无论选择哪个编译器，结果都是一个**托管模块**(managed module)。托管模块是标准的 32 位 Microsoft Windows 可移植执行体(PE32)文件[2]，或者是标准的 64 位 Windows 可移植执行体(PE32+)文件，它们都需要 CLR 才能执行。顺便说一句，托管程序集总是利用 Windows 的数据执行保护(Data Execution Prevention，DEP)和地址空间布局随机化(Address Space Layout Randomization，ASLR)，这两个功能旨在增强整个系统的安全性。

---

① CLR 在早期文档中翻译为“公共语言运行库”，但“库”一词很容易让人误解，所以本书翻译为“公共语言运行时”，或简称为“运行时”。为了和“程序运行的时候”区分，“运行时”在作为名词时会添加引号。——译注

② PE 是 Portable Executable(可移植执行体)的简称。——译注

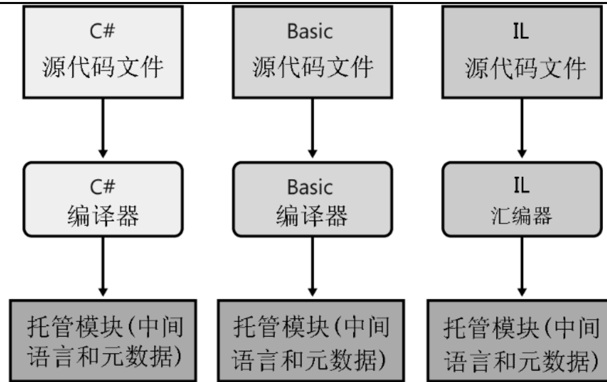


图 1-1 将源代码编译成托管模块

表 1-1 总结了托管模块的各个组成部分。

表 1-1 托管模块的各个部分

组成部分	说明
PE32 或 PE32+头	标准 Windows PE 文件头，类似于“公共对象文件格式”(Common Object File Format，COFF)头。如果这个头使用 PE32 格式，那么文件能在 32 位或 64 位 Windows 上运行。如果这个头使用PE32+格式，那么文件只能在64位Windows 上运行。这个头还标识了文件类型，包括GUI，CUI或者DLL，并包含一个时间标记(时间戳)来指出文件的生成时间。对于只包含 IL 代码的模块，PE32(+)头的大多数信息会被忽视。如果是包含本机(native)CPU 代码的模块，那么这个头会包含与本机 CPU 代码有关的信息
CLR 头	包含使这个模块成为托管模块的信息(可由CLR和一些实用程序进行解释)。头中包含要求的 CLR 版本，一些标志(flag)，托管模块入口方法(`Main` 方法)的 `MethodDef` 元数据 token，以及模块的元数据、资源、强名称、一些标志及其他不太重要的数据项的位置/大小
元数据	每个托管模块都包含元数据表。主要有两种表：一种描述源代码中定义的类型和成员，另一种描述源代码引用的类型和成员
IL(中间语言)代码	编译器编译源代码时生成的代码。在运行时，CLR 将 IL 编译成本机 CPU 指令

本机代码编译器(native code compilers)[①]生成的是面向特定 CPU 架构(比如 x86，x64 或 ARM)的代码。相反，每个面向 CLR 的编译器生成的都是 IL(中间语言)代码。(本章稍后会详细讨论 IL 代码。)IL 代码有时称为**托管代码**(managed code)，因为是由 CLR 管理它的执行。

除了生成 IL，所有面向 CLR 的编译器还要在每个托管模块中生成完整的元数据(metadata)。元数据简单地说就是数据表的一个集合。一些数据表描述了模块中定义了什么(比如类型及其成员)，另一些描述了模块引用了什么(比如导入的类型及其成员)。元数据是一些老技术的超集。这些老技术包括 COM 的“类型库”(Type Library)和“接口定义语言”(Interface Definition Language，IDL)文件。但是，CLR 元数据远比它们全面。另外，和类型库及 IDL 不同，元数据总是与包含 IL 代码的文件关联。事实上，元数据总是嵌入和代码相同的 EXE/DLL 文件中，这使两者密不可分。由于编译器同时生成元数据和代码，把它们绑定一起，并嵌入最终生成的托管模块，所以元数据和它描述的 IL 代码永远不会失去同步。

[①] native 在文档中翻译为“本机”，个人更喜欢“原生”，比如原生类库、原生 C/C++代码、原生堆。一切非托管的，都是 native 的。——译注

元数据有多种用途，下面仅列举一部分。

- 元数据避免了编译时对原生 C/C++头和库文件的需求，因为在实现类型/成员的 IL 代码文件中，已包含有关引用类型/成员的全部信息。编译器直接从托管模块读取元数据。
- Microsoft Visual Studio 用元数据帮助你写代码。“智能感知”(IntelliSense)技术会解析元数据，告诉你一个类型提供了哪些方法、属性、事件和字段。对于方法，还能告诉你需要的参数。
- CLR 的代码验证过程使用元数据确保代码只执行“类型安全”的操作。(稍后就会讲到验证。)
- 元数据允许将对象的字段序列化到内存块，将其发送给另一台机器，然后反序列化，在远程机器上重建对象状态。
- 元数据允许垃圾回收器跟踪对象生存期。垃圾回收器能判断任何对象的类型，并从元数据知道那个对象中的哪些字段引用了其他对象。

第 2 章“生成、打包、部署和管理应用程序及类型”将更详细地讲述元数据。

Microsoft 的 C#，Visual Basic，F#和 IL 汇编器总是生成包含托管代码(IL)和托管数据(支持垃圾回收的数据类型)的模块。为了执行包含托管代码以及/或者托管数据的模块，最终用户(end users)必须在自己的计算机上安装好 CLR(目前作为.NET Framework 的一部分提供)。这类似于为了运行 MFC 或者 Visual Basic 6.0 应用程序，用户必须安装 Microsoft Foundation Class(MFC)库或者 Visual Basic DLL。

Microsoft 的 C++编译器默认生成包含非托管(native)代码的 EXE/DLL 模块，并在运行时操纵非托管数据(native 内存)。这些模块不需要 CLR 即可执行。然而，通过指定**/CLR** 命令行开关，C++编译器就能生成包含托管代码的模块。当然，最终用户必须安装 CLR 才能执行这种代码。在前面提到的所有 Microsoft 编译器中，C++编译器是最特别的，只有它才允许开发人员同时写托管和非托管代码，并生成到同一个模块中。它也是唯一允许开发人员在源代码中同时定义托管和非托管数据类型的 Microsoft 编译器。Microsoft C++编译器的灵活性是其他编译器无法比拟的，因为它允许开发人员在托管代码中使用原生 C/C++代码，时机成熟后再使用托管类型。

1.2 将托管模块合并成程序集

CLR 实际不和模块打交道。它和程序集打交道。**程序集**(assembly)是抽象概念，初学者很难把握它的精髓。首先，程序集是一个或多个模块/资源文件的逻辑性分组。其次，程序集是重用、安全性以及版本控制的最小单元。取决于你选择的编译器或工具，既可生成单文件程序集，也可生成多文件程序集。在 CLR 的世界中，程序集相当于“组件”。

第 2 章会深入探讨程序集，这里不打算花费太多笔墨。只想提醒一句：利用“程序集”这种概念性的东西，可将一组文件当作一个单独的实体来看待。

图 1-2 有助于理解程序集。图中一些托管模块和资源(或数据)文件准备交由一个工具处理。该工具生成代表文件逻辑分组的一个 PE32(+)文件。实际发生的事情是，这个 PE32(+)文件包含一个名为**清单**(manifest)的数据块。清单也是元数据表的一个集合。这些表描述了构成程序集的文件、程序集中的文件所实现的公开导出的类型[①]以及与程序集关联的资源或数据文件。

① 所谓公开导出的类型，就是程序集中定义的 **public** 类型，它们在程序集内部外部均可见。——译注

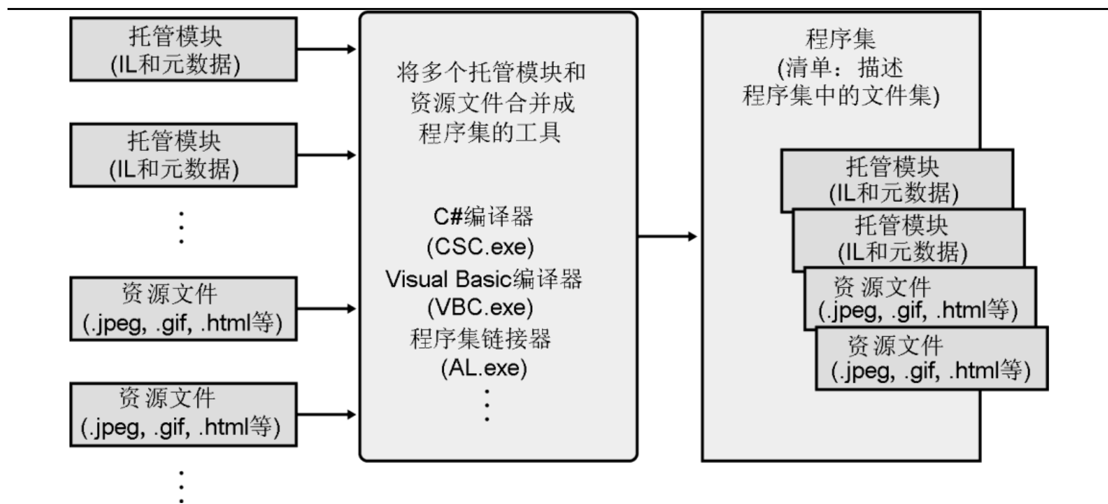


图 1-2 将托管模块合并成程序集

编译器默认将生成的托管模块转换成程序集。也就是说，C#编译器生成含有清单的一个托管模块。清单指出程序集只由一个文件构成。所以，对于只有一个托管模块且无资源(或数据)文件的项目，程序集就是托管模块，生成过程中无需执行任何额外的步骤。但是，如果希望将一组文件合并到程序集中，就必须掌握更多的工具(比如程序集链接器 AL.exe)及其命令行选项。第 2 章将解释这些工具和选项。

对于一个可重用的、可保护的、可版本控制的组件，程序集把它的逻辑表示和物理表示区分开。具体如何用不同的文件划分代码和资源，这完全取决于个人。例如，可以将很少用到的类型或资源放到单独的文件中，并把这些文件作为程序集的一部分。在运行的时候，可以根据需要从网上下载这些单独的文件。如果文件永远用不上，那么永远不会下载。这样不仅节省磁盘空间，还缩短了安装时间。利用程序集，可以在不同的地方部署文件，同时仍然将所有文件作为一个整体来对待。

在程序集的模块中，还包含与引用的程序集有关的信息(包括其版本号)。这些信息使程序集能够**自描述**(self-describing)。也就是说，CLR 能判断为了执行程序集中的代码，程序集需要哪些直接依赖项(immediate dependency)。不需要在注册表或 Active Directory Domain Services(ADDS)中保存额外的信息。由于无需额外信息，所以和非托管组件相比，程序集更容易部署。

1.3 加载公共语言运行时(CLR)

生成的每个程序集既可以是可执行应用程序，也可以是 DLL(其中含有一组由可执行程序使用的类型)。当然，最终是由 CLR 管理这些程序集中的代码的执行。这意味着目标机器必须安装好.NET Framework。Microsoft 创建了一个重分发包(redistribution package)，允许

将.NET Framework 免费分发并安装到用户的计算机上。现在的 Windows 通常都预装了.NET Framework。

要知道是否已安装.NET Framework，只需检查%SystemRoot%\System32 目录中的 MSCorEE.dll 文件。存在该文件，表明.NET Framework 已安装。然而，一台机器可能同时安装好几个版本的.NET Framework。要了解安装了哪些版本的.NET Framework，请检查以下目录的子目录：

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

.NET Framework SDK 提供了名为 CLRVer.exe 的命令行实用程序，能列出机器上安装的所有 CLR 版本。还能列出机器中正在运行的进程使用的 CLR 版本号，方法是使用`-all`命令行开关，或向其传递目标进程 ID。

在学习 CLR 具体如何加载之前，让我们稍微花点时间讨论一下 Windows 的 32 位和 64 位版本。如果你的程序集文件只包含类型安全的托管代码，那么代码在 32 位和 64 位 Windows 上都能正常工作。在这两种 Windows 上运行时，源代码无需任何改动。事实上，编译器最终生成的 EXE/DLL 文件在 Windows 的 x86 和 x64 版本上都能正常工作。此外，Windows Store 应用或类库能在 Windows RT 机器(使用 ARM CPU)上运行。也就是说，只要机器上安装了对应版本的.NET Framework，文件就能运行。

极少数情况下，开发人员希望代码只在一个特定版本的 Windows 上运行。例如，要使用不安全的代码，或者要和面向一种特定 CPU 架构的非托管代码进行互操作，就可能需要这样做。为了帮助这些开发人员，C#编译器提供了一个`/platform`命令行开关选项。这个开关允许指定最终生成的程序集只能在运行 32 位 Windows 版本的 x86 机器上使用，只能在运行 64 位 Windows 的 x64 机器上使用，或者只能在运行 64 位 Windows RT 的 ARM 机器上使用。不指定具体平台的话，默认选项就是`anycpu`，表明最终生成的程序集能在任何版本的 Windows 上运行。Visual Studio 用户要想设置目标平台，可以打开项目的属性页，从“生成”标签页的“目标平台”列表中选择一个选项，如图 1-3 所示。

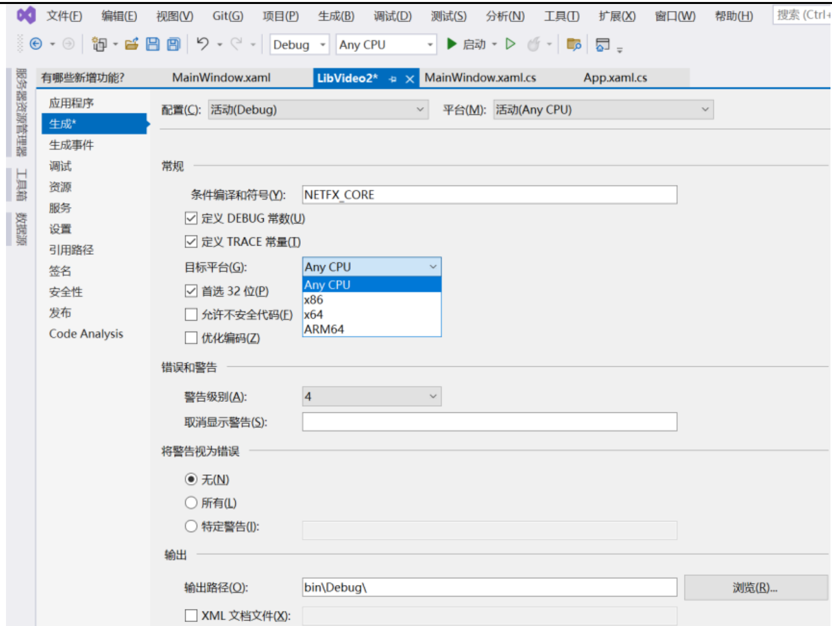


图 1-3 在 Visual Studio 中设置目标平台

取决于/platform开关选项，C#编译器生成的程序集包含的要么是PE32头，要么是PE32+头。除此之外，编译器还会在头中指定要求什么CPU架构(如果使用默认值anycpu，则代表任意CPU架构)。Microsoft发布了SDK命令行实用程序DumpBin.exe和CorFlags.exe，可用它们检查编译器生成的托管模块所嵌入的信息。

可执行文件运行时，Windows检查文件头，判断需要32位还是64位地址空间。PE32文件在32位或64位地址空间中均可运行，PE32+文件则需要64位地址空间。Windows还会检查头中嵌入的CPU架构信息，确保当前计算机的CPU符合要求。最后，Windows的64位版本通过WoW64(Windows on Windows 64)技术运行32位Windows应用程序。

表1-2总结了两方面的信息。其一，为C#编译器指定不同/platform命令行开关将得到哪种托管模块。其二，应用程序在不同版本的Windows上如何运行。[1]

表1-2 /platform开关选项对生成的模块的影响以及在运行时的影响

/platform开关	生成的托管模块	x86 Windows	x64 Windows	ARM Windows RT
anycpu(默认)	PE32/任意CPU架构	作为32位应用程序运行	作为64位应用程序运行	作为32位应用程序运行
anycpu32bitpreferred	PE32/任意CPU架构	作为32位应用程序运行	作为WoW64应用程序运行	作为32位应用程序运行
x86	PE32/x86	作为32位应用程序运行	作为WoW64应用程序运行	不运行
x64	PE32+/x64	不运行	作为64位应用程序运行	不运行
arm	PE32/ARM	不运行	不运行	作为32位应用程序运行
arm64	PE32+/ARM	不运行	不运行	作为64位应用程序运行

Windows检查EXE文件头，决定是创建32位还是64位进程之后，会在进程地址空间加载

[1] ARM版的Windows一般只提供给OEM厂商，不提供给最终用户。2023年主要还在使用ARM架构的设备是Microsoft Surface(Surface Pro 9可选x86和ARM版本)和Apple的各种使用M1/M2芯片的电脑。因此，可以在MacOS上用虚拟机安装ARM版的Windows。注意，目前ARM版本的Win10测试通道已经关闭，只有Win11。——译注

MSCorEE.dll 的 x86，x64 或 ARM 版本。如果是 Windows 的 x86 或 ARM 版本，MSCorEE.dll 的 x86 版本在%SystemRoot%\System32 目录中。如果是 Windows 的 x64 版本，MSCorEE.dll 的 x86 版本在%SystemRoot%\SysWow64 目录中，64 位版本则在%SystemRoot%\System32 目录中(为了向后兼容)。然后，进程的主线程调用 MSCorEE.dll 中定义的一个方法。这个方法初始化 CLR，加载 EXE 程序集，再调用其入口方法(`Main`)。随即，托管应用程序启动并开始运行。[①]

注意：Microsoft C#编译器 1.0 或 1.1 版本生成的程序集包含的是 PE32 头，而且未明确指定 CPU 架构。但在加载时，CLR 认为这些程序集只用于 x86。对于可执行文件，这增强了应用程序与 64 位系统的兼容性，因为可执行文件将在 WoW64 中加载，为进程提供和 Windows 的 32 位 x86 版本非常相似的环境。

如果非托管应用程序调用 Win32 `LoadLibrary` 函数来加载一个托管程序集，那么 Windows 会自动加载并初始化 CLR(如果尚未加载)以处理程序集中的代码。当然，这个时候进程已经启动并运行了，而这可能会限制程序集的可用性。例如，64 位进程完全无法加载使用 `/platform:x86` 开关编译的托管程序集，而用同一个开关编译的可执行文件就能在 64 位 Windows 中用 WoW64 进行加载。

1.4 执行程序集的代码

如前所述，托管程序集同时包含元数据和 IL。IL 是与 CPU 无关的机器语言，是 Microsoft 在请教了外部的几个商业及学术性语言/编译器的作者之后，费尽心思开发出来的。IL 比大多数 CPU 机器语言都高级[②]。IL 能访问和操作对象类型，并提供了指令来创建和初始化对象、在对象上调用虚方法以及直接操作数组元素。甚至提供了抛出和捕捉异常的指令以实现错误处理。可将 IL 视为一种面向对象的机器语言。

开发人员一般用 C#，Visual Basic 或 F#等高级语言进行编程。它们的编译器将生成 IL。然而，和其他任何机器语言一样，IL 也能使用汇编语言编写，Microsoft 甚至专门提供了名为 ILAsm.exe 的 IL 汇编器和名为 ILDasm.exe 的 IL 反汇编器。

注意，高级语言通常只公开了 CLR 全部功能的一个子集。然而，IL 汇编语言允许开发人员访问 CLR 的全部功能。所以，如果你选择的编程语言隐藏了自己迫切需要的一个 CLR 功能，那么可以换用 IL 汇编语言或者提供了所需功能的另一种编程语言来写那部分代码。

---

① 可以在代码中查询 `Environment` 的 `Is64BitOperatingSystem` 属性，判断是否在 64 位 Windows 上运行。还可以查询 `Environment` 的 `Is64BitProcess` 属性，判断是否在 64 位地址空间中运行。

② 在中文语境中，人们往往把“更高级”理解为“更好”。但在软件开发/工程/需求领域中，“更高级”简单地意味着具有更高的抽象级别。机器语言是计算机唯一能理解的、最低级的语言。——译注

重要提示：在我看来，允许在不同编程语言之间方便地切换，同时又保持紧密集成，这是 CLR 很出众的一个特点。遗憾的是，许多开发人员都忽视了这一点。例如，C#和 Visual Basic 等语言能很好地执行 I/O 操作，APL 语言能很好地执行高级工程或金融计算。通过 CLR，应用程序的 I/O 部分可用 C#编写，工程计算部分则换用 APL 编写。CLR 在这些语言之间提供了其他技术无法媲美的集成度，使“混合语言编程”成为许多开发项目一个值得认真考虑的选择。

要知道 CLR 具体提供了哪些功能，唯一的办法就是阅读 CLR 文档。本书致力于讲解 CLR 的功能，以及 C#语言如何公开这些功能。对于 C#没有公开的 CLR 功能，本书也进行了说明。相比之下，其他大多数书籍和文章都是从一种语言的角度讲解 CLR，造成大多数开发人员误以为 CLR 只提供了他们选用的那一种语言所公开的那一部分功能。当然，如果能用一种语言达到目的，那么这种误解也不一定是件坏事。

为了执行方法，首先必须把方法的 IL 转换成本机 CPU 指令。这是 CLR 的 JIT(just-in-time 或者“即时”)编译器的职责。

图 1-4 展示了首次调用一个方法时发生的事情。

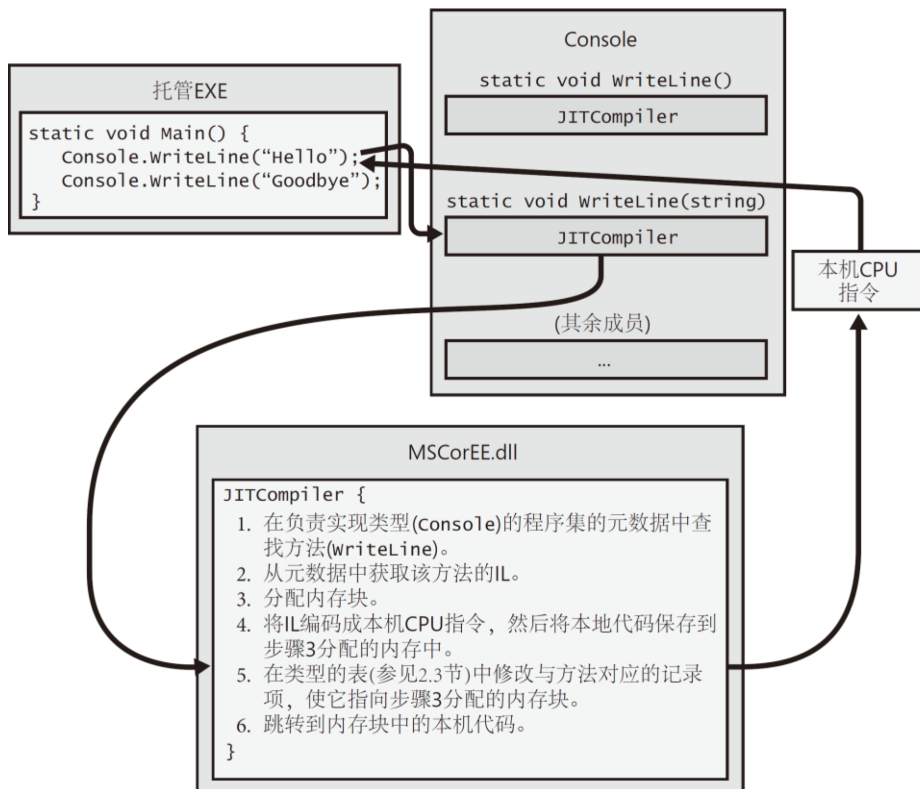


图 1-4 方法的首次调用

就在 `Main` 方法执行之前，CLR 会检测 `Main` 的代码引用的所有类型。这导致 CLR 分配一

个内部数据结构来管理对引用类型的访问。图 1-4 的 Main 方法只引用了一个 Console 类型，导致 CLR 分配一个内部结构。在这个内部数据结构中，Console 类型定义的每个方法都有一个对应的记录项[①]。每个记录项都含有一个地址，根据此地址即可找到方法的实现。对这个结构初始化时，CLR 将每个记录项都设置成(指向)包含在 CLR 内部的一个未编档函数。我把这个函数称为 JITCompiler。

Main 方法首次调用 WriteLine 时，JITCompiler 函数会被调用。JITCompiler 函数负责将方法的 IL 代码编译成本机 CPU 指令。由于 IL 是“即时”(just in time)编译的，所以通常将 CLR 的这个组件称为 JITter 或者 JIT **编译器**。

注意：如果应用程序在 Windows 的 x86 版本或 WoW64 中运行，那么 JIT 编译器将生成 x86 指令。作为 64 位应用程序在 Windows 的 x64 版本中运行，将生成 x64 指令。在 Windows 的 ARM 版本中运行，则将生成 ARM 指令。

JITCompiler 函数被调用时，它知道要调用的是哪个方法，以及具体是什么类型定义了该方法。然后，JITCompiler 会在定义(该类型的)程序集的元数据中查找被调用方法的 IL。接着，JITCompiler 验证 IL 代码，并将 IL 代码编译成本机 CPU 指令。本机 CPU 指令保存到动态分配的内存块中。然后，JITCompiler 回到 CLR 为类型创建的内部数据结构，找到与被调用方法对应的那条记录，修改最初对 JITCompiler 的引用，使其指向内存块(其中包含了刚才编译好的本机 CPU 指令)的地址。最后，JITCompiler 函数跳转到内存块中的代码。这些代码正是 WriteLine 方法(获取单个 String 参数的那个版本)的具体实现。代码执行完毕并返回时，会回到 Main 中的代码，并像往常一样继续执行。

现在，Main 要第二次调用 WriteLine。这一次，由于已对 WriteLine 的代码进行了验证和编译，所以会直接执行内存块中的代码，完全跳过 JITCompiler 函数。WriteLine 方法执行完毕后，会再次回到 Main。图 1-5 展示了第二次调用 WriteLine 时发生的事情。

① 本书将 entry 翻译成“记录项”，其他译法还有条目、入口等等。虽然某些 entry 包含了一个地址，所以相当于一个指针，但并非所有 entry 都是这样的。在其他 entry 中，还可能包含了文件名、类型名、方法名和位标志等信息。——译注

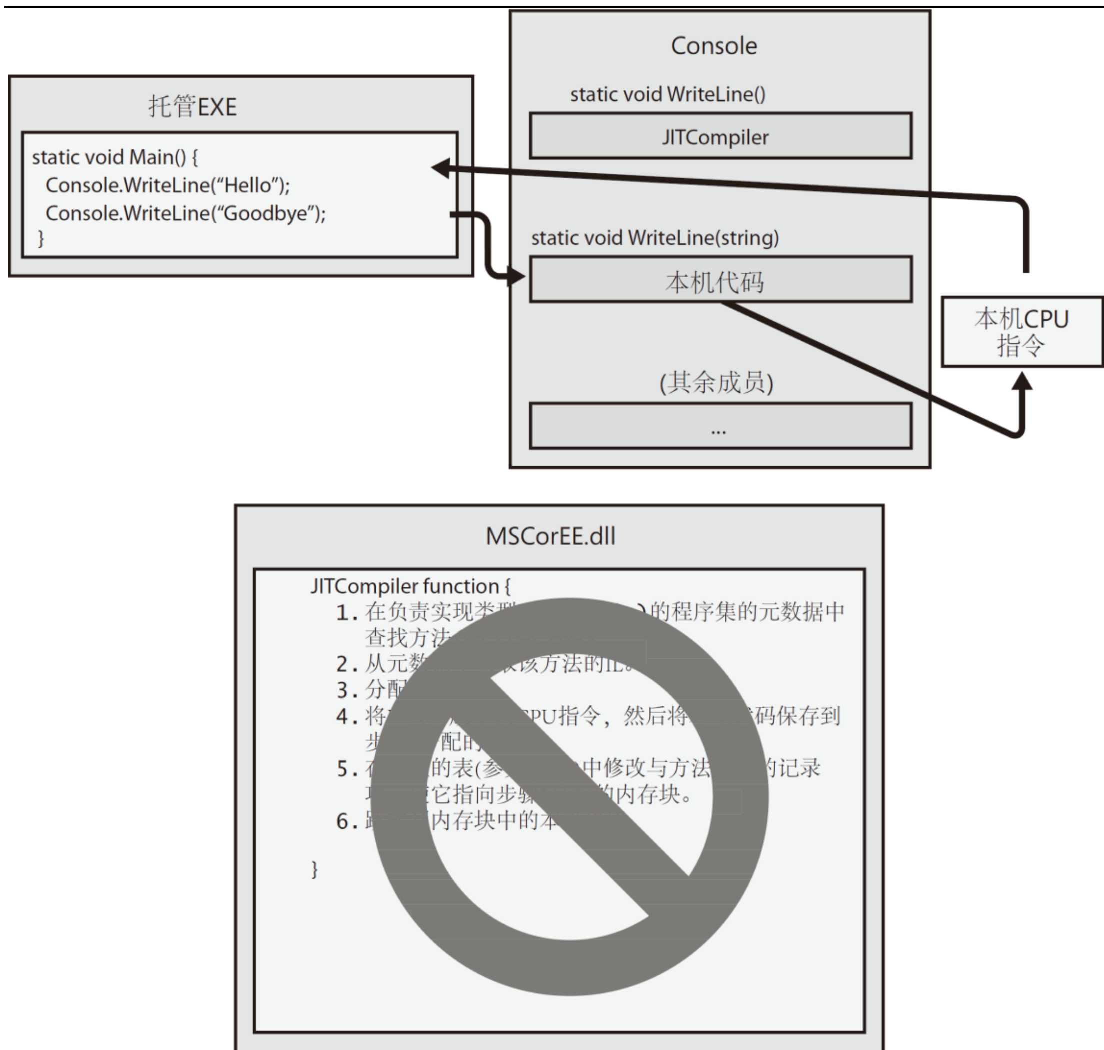


图 1-5 方法的第二次调用

方法仅在首次调用时才会有一些性能损失。以后对该方法的所有调用都以本机代码的形式全速运行，无需重新验证 IL 并把它编译成本机代码。

JIT 编译器将本机 CPU 指令存储到动态内存中。这意味着一旦应用程序终止，编译好的代码也会被丢弃。所以，将来再次运行应用程序，或者同时启动应用程序的两个实例(使用两个不同的操作系统进程)，JIT 编译器必须再次将 IL 编译成本机指令。对于某些应用程序，这可能显著增加内存耗用。相比之下，如果运行的是一个本机(native)应用程序，那么它的只读代码页是可以由应用程序正在运行的所有实例共享的。

对于大多数应用程序，JIT 编译造成的性能损失并不显著。大多数应用程序都反复调用相同的方法。应用程序运行期间，这些方法只会对性能造成一次性的影响。另外，在方法内部花费的时间很有可能比花在调用方法上的时间多得多。

还要注意，CLR 的 JIT 编译器会对本机代码进行优化，这类似于非托管 C++编译器的后端所做的事情。同样，可能要花较多时间来生成优化代码。但和不优化相比，代码优化后的性能更佳。

有两个 C#编译器开关会影响代码优化：/optimize 和/debug。下面总结了这些开关对 C#编译器生成的 IL 代码的质量的影响，以及对 JIT 编译器生成的本机代码的质量的影响。

编译器开关设置	C# IL 代码质量	JIT 本机代码质量
/optimize- /debug-(默认)	未优化	有优化
/optimize- /debug(+/full/pdbonly)	未优化	未优化
/optimize+ /debug(-/+/full/pdbonly)	有优化	有优化

使用/optimize-，在 C#编译器生成的未优化 IL 代码中，将包含许多 NOP(no-operation，空操作)指令，还包含许多跳转到下一行代码的分支指令。Visual Studio 利用这些指令在调试期间提供“编辑并继续”(edit-and-continue)功能。另外，利用这些额外的指令，还可以在控制流程指令(比如 for，while，do，if，else，try，catch 和 finally 语句块)上设置断点，使代码更容易调试。相反，如果生成优化的 IL 代码，C#编译器会删除多余的 NOP 和分支指令。而在控制流程被优化之后，代码就难以在调试器中进行单步调试了。另外，若在调试器中执行，一些函数求值可能无法进行。不过，优化的 IL 代码变得更小，结果 EXE/DLL 文件也更小。另外，如果你像我一样喜欢检查 IL 来理解编译器生成的东西，这种 IL 也更易读。

除此之外，只有指定/debug(+/full/pdbonly)开关，编译器才会生成 Program Database(PDB)文件。PDB 文件帮助调试器查找局部变量并将 IL 指令映射到源代码。/debug:full 开关告诉 JIT 编译器你打算调试程序集，JIT 编译器会记录每条 IL 指令所生成的本机代码。这样一来，就可以利用 Visual Studio 的“即时”(just-in-time)调试功能，将调试器连接到正在运行的进程，并方便地对源代码进行调试。不指定/debug:full 开关，JIT 编译器默认不记录 IL 与本机代码的联系，这使 JIT 编译器运行得稍快，用的内存也稍少。如果进程用 Visual Studio 调试器启动，那么会强迫 JIT 编译器记录 IL 与本机代码的联系(无论/debug 开关的设置是什么)——除非在 Visual Studio 中关闭了“在模块加载时取消 JIT 优化(仅限托管)”选项。[①]

在 Visual Studio 中新建 C#项目时，项目的“调试”(Debug)配置指定的是/optimize-和/debug:full 开关，而“发布”(Release)配置指定的是/optimize+和/debug:pdbonly 开关。

非托管 C 或 C++的开发人员可能担心这一切对于性能的影响。毕竟，非托管代码是针对一种具体 CPU 平台编译的。一旦调用，代码直接就能执行。但是，在现在这种托管环境中，

[①] 选择“工具”|“选项”，然后选择“调试”节点下的“常规”页。——译注

代码的编译是分两个阶段完成的。首先，编译器遍历源代码，做大量工作来生成 IL 代码。但真正要想执行，这些 IL 代码本身必须在运行时编译成本机 CPU 指令，这需要分配更多的非共享内存，而且要花费额外的 CPU 时间。

事实上，我自己也是从 C/C++的背景开始接触 CLR 的，当时也对此持怀疑态度并格外关心这种额外的开销。经过实践，我发现运行时的二次编译确实会影响性能，也确实会分配动态内存。但是，微软进行了大量性能优化工作，将这些额外的开销保持在最低限度之内。

如果仍不放心，就实际生成一些应用程序，亲自测试一下性能。此外，应该运行由 Microsoft 或其他公司生成的一些比较正式的托管应用程序，并测试其性能。相信它们出色的性能表现会让你喜出望外。

虽然你可能很难相信，但许多人(包括我)都认为托管应用程序的性能实际上超越了非托管应用程序。这是出于多方面的原因。例如，当 JIT 编译器在运行时将 IL 代码编译成本机代码时，编译器对执行环境的认识比非托管编译器更深刻。下面列举了托管代码相较于非托管代码的优势。

- JIT 编译器能判断应用程序是否运行在 Intel Pentium 4 CPU 上，并生成相应的本机代码来利用 Pentium 4 支持的任何特殊指令。相反，非托管应用程序通常是针对具有最小功能集合的 CPU 编译的，不会使用能提升性能的特殊指令。
- JIT 编译器能判断一个特定的测试在它运行的机器上是否总是失败。例如，假定一个方法包含以下代码：

  ```
  if (numberOfCPUs > 1) {
    ...
  }
  ```

  如果主机只有一个 CPU，那么 JIT 编译器不会为上述代码生成任何 CPU 指令。在这种情况下，本机代码将针对主机进行优化，最终代码变得更小，执行得更快。
- 应用程序运行时，CLR 可以探查(profile)代码的执行，并将 IL 重新编译成本机代码。重新编译的代码可以重新组织，根据刚才观察到的执行模式，减少不正确的分支预测。虽然目前版本的 CLR 还不能做到这一点，但将来的版本也许就可以了。[①]

除了这些理由，还有另一些理由使我们相信未来的托管代码在执行效率上会比当前的非托管代码更优秀。大多数托管应用程序目前的性能已相当不错，将来还有望进一步提升。

如果你的试验表明，CLR 的 JIT 编译器似乎没有使自己的应用程序达到应有的性能，那么

[①] 现在，可以在 Visual Studio 2022 中选择“调试”|“性能探查器”来使用这个功能。虽然还不能做到全自动，但手动的效果也不错。——译注

可以考虑使用.NET Framework SDK 配套提供的 NGen.exe 工具。该工具将程序集的所有 IL 代码预编译成本机代码，并将这些本机代码保存到一个磁盘文件中。在运行时加载程序集时，CLR 自动判断是否存在该程序集的预编译版本。如果是，CLR 就加载预编译代码。这样一来，就避免了在运行时进行编译。注意，NGen.exe 对最终执行环境的预设是很保守的(不得不如此)。所以，NGen.exe 生成的代码不会像 JIT 编译器生成的代码那样进行高度优化。本章稍后会详细讨论 NGen.exe。

另外，还可以考虑使用 `System.Runtime.ProfileOptimization` 类。该类导致 CLR 检查程序运行时哪些方法被 JIT 编译，结果被记录到一个文件。程序再次启动时，如果是在多 CPU 机器上运行，就用其他线程并发编译这些方法。这使应用程序运行得更快，因为多个方法并发编译，而且是在应用程序初始化时编译，而不是在用户和程序交互时才“即时”编译。

1.4.1 IL 和验证

IL 基于栈(stack)。这意味着它的所有指令都要将操作数压入(push)一个执行栈，并从栈弹出(pop)结果。由于 IL 没有提供操作寄存器的指令，所以人们可以很容易地创建新的语言和编译器，生成面向 CLR 的代码。

IL 指令还是“无类型”(typeless)的。例如，IL 提供了 `add` 指令将压入栈的最后两个操作数加到一起。`add` 指令不分 32 位和 64 位版本。`add` 指令执行时，它判断栈中的操作数的类型，并执行恰当的操作。

我个人认为，IL 最大的优势不在于它对底层 CPU 的抽象，而在于应用程序的健壮性[①]和安全性。将 IL 编译成本机 CPU 指令时，CLR 执行一个名为**验证**(verification)的过程。这个过程会检查高级 IL 代码，确定代码所做的一切都是安全的。例如，会核实调用的每个方法都有正确数量的参数，传给每个方法的每个参数都有正确的类型，每个方法的返回值都得到了正确的使用，每个方法都有一个返回语句，等等。在托管模块的元数据中，包含验证过程要用到的所有方法及类型信息。

Windows 的每个进程都有自己的虚拟地址空间。独立地址空间之所以必要，是因为不能简单地信任一个应用程序的代码。应用程序完全可能读写无效的内存地址(令人遗憾的是，这种情况时有发生)。将每个 Windows 进程都放到独立的地址空间，将获得健壮性与稳定性；一个进程干扰不到另一个进程。

① 这里有必要强调一下健壮性(鲁棒性)和可靠性的区别，两者对应的英文单词分别是 robustness 和 reliability。健壮性主要描述系统对于参数变化的不敏感性，而可靠性主要描述系统的正确性，也就是在你固定提供一个参数时，它应该产生稳定的、能预测的输出。例如一个程序，它的设计目标是获取输入并输出值。假如它能正确完成这个设计目标，就说它是可靠的。但在这个程序执行完毕后，假如没有正确释放内存，或者说系统没有自动帮它释放占用的资源，就认为这个程序及其“运行时”不健壮。另外，在软件需求领域，还有一种说法称为“可信性”(dependability)，它是“健壮性”的同义词，意思就是“我可以依赖你(帮我收尾)”。参见《敏捷软件需求》2024 年精译版，清华大学出版社。——译注

然而，通过验证托管代码，可确保代码不会不正确地访问内存，不会干扰到另一个应用程序的代码。这样就可以放心地将多个托管应用程序放到同一个 Windows 虚拟地址空间运行。

由于 Windows 进程需要大量操作系统资源，所以进程数量太多，会损害性能并制约可用的资源。用一个进程运行多个应用程序，可以减少进程数，从而增强性能，减少所需的资源，健壮性也没有丝毫下降。这是托管代码相较于非托管代码的另一个优势。

事实上，CLR 确实提供了在一个操作系统进程中执行多个托管应用程序的能力。每个托管应用程序都在一个 AppDomain①中执行。每个托管 EXE 文件默认都在它自己的独立地址空间中运行，这个地址空间只有一个 AppDomain。然而，CLR 的宿主进程(比如 IIS 或者 Microsoft SQL Server)可决定在一个进程中运行多个 AppDomain。第 22 章“CLR 寄宿和 AppDomain”会详细讨论 AppDomain。

1.4.2 不安全的代码

Microsoft C#编译器默认生成安全(safe)代码，这种代码的安全性可以验证。然而，Microsoft C#编译器也允许开发人员写不安全的(unsafe)代码。不安全的代码允许直接操作内存地址，并可操作这些地址处的字节。这是非常强大的一个功能，通常只有在与非托管代码进行互操作，或者在提升对效率要求极高的一个算法的性能的时候，才需要这样做。

然而，使用不安全的代码存在重大风险：这种代码可能破坏数据结构，危害安全性，甚至造成新的安全漏洞。有鉴于此，C#编译器要求包含不安全代码的所有方法都用 `unsafe` 关键字标记。除此之外，C#编译器要求使用`/unsafe` 编译器开关来编译源代码。

当 JIT 编译器编译一个 `unsafe` 方法时，会检查该方法所在的程序集是否被授予了 `System.Security.Permissions.SecurityPermission` 权限，而且 `System.Security.Permissions. SecurityPermissionFlag` 的 `SkipVerification` 标志是否设置。如果该标志已经设置，JIT 编译器会编译不安全的代码，并允许代码执行。CLR 信任这些代码，并希望对地址及字节的直接操作不会造成损害。如果标志未设置，那么 JIT 编译器会抛出 `System.InvalidProgramException` 或 `System.Security.VerificationException` 异常，禁止方法执行。事实上，整个应用程序都有可能在这个时候终止，但这至少能防止造成更大的损害。

注意：从本地计算机或“网络共享”加载的程序集默认被授予完全信任，这意味着它们能做任何事情，包括执行不安全代码。但通过 Internet 执行的程序集默认不会被授予执行不安全代码的权限。如果含有不安全的代码，就会抛出上述异常之一。管理员和最终用户可以修改这些默认设置；但在这种情况下，管理员要对代码的行为负全责。

① 本书按照原书的风格保持了 AppDomain 这样的写法，未将其翻译成“应用程序域”。需引用 `AppDomain` 类的时候，会使用代码(等宽)字体。平时引用时，则采用普通字体。——译注

微软提供了一个名为 PEVerify.exe 的实用程序，它检查一个程序集的所有方法，并报告其中含有不安全代码的方法。对想要引用的程序集运行一下 PEVerify.exe，看看应用程序在通过内网或 Internet 运行时是否会出问题。

注意，为了进行验证，需要访问所有依赖的程序集中包含的元数据。所以，当 PEVerify 检查程序集时，它必须能够定位并加载引用的所有程序集。由于 PEVerify 使用 CLR 来定位依赖的程序集，所以会采用和平时执行程序集时一样的绑定(binding)和探测(probing)规则来定位程序集。这些绑定和探测规则将在第 2 章和第 3 章讨论。

IL 和知识产权保护

有人担心 IL 没有为他们的算法提供足够的知识产权保护。换言之，他们认为在生成托管模块后，别人可以使用工具(比如 IL 反汇编器)来进行逆向工程，轻松还原应用程序的代码所做的事情。

我承认，IL 代码确实比其他大多数汇编语言高级，而且对 IL 代码进行逆向工程相对而言比较简单。不过，在实现服务器端代码(比如 Web 服务、Web 窗体或者存储过程)的时候，程序集是放在服务器上的。由于没人能从公司外部拿到程序集，自然就没人能从公司外部使用工具查看 IL。所以，这个时候的知识产权是完全有保障的。

如果担心分发出去的程序集，那么可以从第三方厂商购买某个混淆器(obfuscator)实用程序。这种实用程序能打乱程序集元数据中的所有私有符号的名称。别人很难还原这些名称，从而很难理解每个方法的作用。但要注意，这些混淆器提供的保护是有限的，因为 IL 必须在某个时候提供给 CLR 做 JIT 编译。

如果觉得混淆器不能提供自己需要的知识产权保护等级，那么可以考虑在非托管模块中实现你想保密的算法。这种模块包含的是本机 CPU 指令，而不是 IL 和元数据。然后，可以利用 CLR 的互操作功能(要有足够的权限)来实现应用程序的托管与非托管部分之间的通信。当然，上述方案的前提是不担心别人对非托管代码中的本机 CPU 指令进行逆向工程。

1.5 本机代码生成器：NGen.exe

使用.NET Framework 提供的 NGen.exe 工具，可以在应用程序安装到用户的计算机上时，将 IL 代码编译成本机代码。由于代码在安装时已经编译好，所以 CLR 的 JIT 编译器不再需要在运行时编译 IL 代码，这有助于提升应用程序的性能。NGen.exe 能在以下两种情况下发挥重要作用。

1. 提高应用程序的启动速度

运行 NGen.exe 能提高启动速度，代码已编译成本机代码，所以运行时不再需要花时间编译。

2. 减小应用程序的工作集[①]

如果你认为一个程序集会同时加载到多个进程中，那么对该程序集运行 NGen.exe 可以减小应用程序的工作集。原因是 NGen.exe 会将 IL 编译成本机代码，并将这些代码保存到单独的文件中。该文件可以通过“内存映射”的方式，同时映射到多个进程地址空间中，使代码得到了共享，避免每个进程都需要一份单独的代码拷贝。

安装程序为一个应用程序或程序集调用 NGen.exe 时，该应用程序的所有程序集(或者那个指定的程序集)的 IL 代码会被编译成本机代码。NGen.exe 会新建一个程序集文件，其中只包含这种本机代码，不包含任何 IL。新文件会放到%SystemRoot%\Assembly\NativeImages_v4.0.#####_64 这样的一个目录下的一个文件夹中。在目录名称中，除了包含 CLR 版本号，还会描述本机代码是为 32 位还是 64 位 Windows 编译的。

现在，每当 CLR 加载程序集文件，都会检查是否存在一个对应的、由 NGen 生成的本机文件。如果找不到本机文件，CLR 就和往常一样对 IL 代码进行 JIT 编译。如果有对应的本机文件，CLR 就直接使用本机文件中编译好的代码，文件中的方法不需要在运行时编译。

表面上一切都很完美！一方面，获得了托管代码的所有好处(垃圾回收、验证、类型安全等等)；另一方面，没有托管代码(JIT 编译)的所有性能问题。但是，不要被表面所迷惑。NGen 生成的文件存在以下问题。

- **没有知识产权保护**

  许多人以为，发布 NGen 生成的文件(而不发布包含原始 IL 代码的文件)能保护知识产权。但遗憾的是，这是不可能的。在运行时，CLR 要求访问程序集的元数据(用于反射和序列化等功能)，这就要求发布包含 IL 和元数据的程序集。此外，如果 CLR 因为某些原因不能使用 NGen 生成的文件(如后文所述)，CLR 会自动对程序集的 IL 代码进行 JIT 编译，所以 IL 代码必须处于可用状态。

- **NGen 生成的文件可能失去同步**

  CLR 加载 NGen 生成的文件时，会将预编译代码的许多特征与当前执行环境进行比较。任何特征不匹配，NGen 生成的文件就不能使用。此时要改为使用正常的 JIT 编译器进程。下面列举了必须匹配的部分特征。

  ○ CLR 版本：随补丁或 Service Pack 改变

  ○ CPU 类型：升级处理器就会改变

---

① 所谓工作集(working set)，是指在进程的所有内存中，已映射物理内存的那一部分(即这些内存块全在物理内存中，并且 CPU 能直接访问)；进程还有一部分虚拟内存，它们可能在转换列表中(CPU 不能通过虚地址访问，需要 Windows 映射之后才能访问)；还有一部分内存在磁盘上的分页文件里。——译注

- ○ Windows 操作系统版本：安装新 Service Pack 后改变
- ○ 程序集的标识模块版本 ID(Module Version ID，MVID)：重新编译后改变
- ○ 引用的程序集的版本 ID：重新编译引用的程序集后改变
- ○ 安全性：吊销了之前授予的权限之后，安全性就会发生改变。这些权限包括声明性继承(declarative inheritance)、声明性链接时(declarative link-time)[①]、`SkipVerification` 或者 `UnmanagedCode` 权限。

注意：可以使用更新(update)模式运行 NGen.exe，这会为之前用 NGen 生成的所有程序集再次运行 NGen.exe。用户一旦安装了.NET Framework 的新 Service Pack，这个 Service Pack 的安装程序就会自动用更新模式运行 NGen.exe，使 NGen 生成的文件与新安装的 CLR 版本同步。

- **较差的执行时性能**

  编译代码时，NGen 无法像 JIT 编译器那样对执行环境进行许多假定。这会造成 NGen.exe 生成较差的代码。例如，NGen 不能优化地使用特定 CPU 指令；静态字段只能间接访问，而不能直接访问，因为静态字段的实际地址只能在运行时确定。NGen 到处插入代码来调用类构造器，因为它不知道代码的执行顺序，也不知道一个类构造器是否已经调用(第 8 章“方法”会详细讲述类构造器的问题)。测试表明，相较于 JIT 编译的版本，NGen 生成的某些应用程序在执行时反而要慢 5%左右。所以，如果考虑使用 NGen.exe 来提升应用程序的性能，那么必须仔细比较 NGen 版本和非 NGen 版本，确定 NGen 版本不会变得更慢！对于某些应用程序，由于减小工作集能提升性能，所以使用 NGen 仍有优势。

正是由于这些问题，所以使用 NGen.exe 时必须谨慎。对于服务器端应用程序，NGen.exe 的作用并不明显，有时甚至毫无用处，这是因为只有第一个客户端请求才会感受到性能下降，后续所有客户端请求都能以全速运行。此外，大多数服务器应用程序只需要代码的一个实例，所以缩小工作集不能带来任何好处。

对于客户端应用程序，使用 NGen.exe 也许能提高启动速度，或者能缩小工作集(如果程序集同时由多个应用程序使用)。即便程序集不由多个应用程序使用，用 NGen 来生成也可能会改善工作集。另外，用 NGen.exe 来生成客户端应用程序的所有程序集，CLR 就不需要加载 JIT 编译器了，从而进一步缩小工作集。当然，只要有一个程序集不是用 NGen 生成的，或者程序集的一个由 NGen 生成的文件无法使用，那么还是会加载 JIT 编译器，应用程序的工作集将随之增大。

---

① declarative inheritance 权限是派生出程序集的那个类所要求的；declarative link-time 权限是程序集调用的方法所要求的。另外，虽然文档将 declarative 翻译成“声明性”，但个人更喜欢“宣告式”。——译注

对于启动很慢的大型客户端应用程序，Microsoft 提供了 Managed Profile Guided Optimization 工具(MPGO.exe)。该工具分析程序执行，检查它在启动时需要哪些东西。这些信息反馈给 NGen.exe 来更好地优化本机映像，这使应用程序启动得更快，工作集也缩小了。准备发布应用程序时，用 MPGO 工具启动它，走一遍程序的常规任务。与所执行代码有关的信息会写入一个 profile 并嵌入程序集文件中。NGen.exe 工具利用这些 profile 数据来更好地优化它生成的本机映像。

1.6 Framework 类库简介

.NET Framework 包含 Framework 类库(Framework Class Library，FCL)。FCL 是一组 DLL 程序集的统称，其中含有数千个类型定义，每个类型都公开了一些功能。Microsoft 还发布了其他库，比如 Windows Azure SDK 和 DirectX SDK。这些库提供了更多类型，公开了更多功能。事实上，Microsoft 正在以惊人的速度发布各种各样的库，开发者使用各种 Microsoft 技术变得前所未有的简单。

下面列举了应用程序开发人员可以利用这些程序集创建的一部分应用程序。

- **Web 服务(Web service)**

  利用微软的 ASP.NET XML Web Service 技术或者 Windows Communication Foundation(WCF)技术，可以非常简单地处理通过 Internet 发送的消息。

- **基于 HTML 的 Web 窗体/MVC 应用程序(网站)**

  通常，ASP.NET 应用程序查询数据库并调用 Web 服务，合并和筛选返回的信息，然后使用基于 HTML 的"富"用户界面，在浏览器中显示那些信息。

- **"富"Windows GUI 应用程序**

  也可以不用网页创建 UI，而是用 Windows Store、Windows Presentation Foundation(WPF)或者 Windows Forms 技术提供的更强大、性能更好的功能。GUI 应用程序可以利用控件、菜单以及触摸/鼠标/手写笔/键盘事件，而且可以直接与底层操作系统交换信息。"富"Windows 应用程序同样可以查询数据库和使用 Web 服务。

- **Windows 控制台应用程序**

  如果对 UI 的要求很简单，那么控制台应用程序提供了一种快速、简单的方式来生成应用程序。编译器、实用程序和工具一般都是作为控制台应用程序实现的。

- **Windows 服务**

  是的，完全可以用.NET Framework 生成"服务"应用程序。通过"Windows 服务控制管理器"(Service Control Manager，SCM)控制这些服务。

- **数据库存储过程**

  Microsoft 的 SQL Server、IBM 的 DB2 以及 Oracle 的数据库服务器允许开发人员用.NET Framework 写存储过程。

- **组件库**

  .NET Framework 允许生成独立程序集(组件)，其中包含的类型可以轻松集成到前面提到的任何一种类型的应用程序中。

重要提示：Visual Studio 允许创建“可移植类库”项目[①]。这种项目创建的程序集能用于多种应用程序类型，包括.NET Framework，Silverlight，Windows Phone，Windows Store 应用和 Xbox Series。

由于 FCL 包含的类型数量实在太多，所以有必要将相关的类型放到单独的命名空间。例如，`System` 命名空间(应当是你最熟悉的)包含 `Object` 基类型，其他所有类型最终都从这个基类型派生。此外，`System` 命名空间包含用于整数、字符、字符串、异常处理以及控制台 I/O 的类型。还包含一系列实用工具类型，能在不同数据类型之间进行安全转换、格式化数据类型、生成随机数和执行各种数学运算。所有应用程序都要使用来自 `System` 命名空间的类型。

使用 Framework 的任何功能时，都必须知道这个功能由什么类型提供，以及该类型包含在哪个命名空间中。许多类型都允许自定义其行为，你只需从所需的 FCL 类型派生出自己的类型，再进行自定义即可。.NET Framework 平台本质上是面向对象的，这为软件开发人员提供了一致性的编程模式。此外，开发人员可轻松创建自己的命名空间来包含自己的类型。这些命名空间和类型无缝合并到编程模式中。相较于 Win32 编程模式，这种新方式极大地简化了软件开发。

FCL 的大多数命名空间都提供了各种应用程序通用的类型。表 1-3 总结了部分常规命名空间，并简要描述了其中的类型的用途。这里列出的只是全部可用命名空间中极小的一部分。请参考文档来熟悉 Microsoft 发布的命名空间(它们的数量正在变得越来越多)。

表 1-3 部分常规的 FCL 命名空间

命名空间	内容说明
`System`	包含每个应用程序都要用到的所有基本类型
`System.Data`	包含用于和数据库通信以及处理数据的类型
`System.IO`	包含用于执行流 I/O 以及浏览目录/文件的类型
`System.Net`	包含进行低级网络通信，并与一些常用 Internet 协议协作的类型
`System.Runtime.InteropServices`	包含允许托管代码访问非托管操作系统平台功能(比如 COM 组件以及 Win32 或定制 DLL 中的函数)的类型
`System.Security`	包含用于保护数据和资源的类型
`System.Text`	包含处理各种编码(比如 ASCII 和 Unicode)文本的类型
`System.Threading`	包含用于异步操作和同步资源访问的类型

[①] 在 Visual Studio 的最新版本中(目前是 VS 2022)，可移植类库(PCL)已被弃用。 虽然仍然可以打开、编辑和编译 PCL，但对于新项目，建议改为使用 .NET Framework 类库。——译注

System.Xml	包含用于处理 XML 架构(XML Schema)和数据的类型

本书重点在于 CLR 以及与 CLR 密切交互的常规类型。所以，任何开发人员只要开发的应用程序或组件是面向 CLR 的，就适合阅读本书。还有其他许多不错的参考书描述了具体应用程序类型，包括 Web Services、Web 窗体/MVC 和 Windows Presentation Foundation 等。这些书能指导你快速开始构建自己的应用程序。我认为这些针对具体应用程序的参考书有助于进行“自上而下”的学习，因为它们将重点放在具体应用程序类型上，而非放在开发平台上。相反，本书提供的信息有助于你进行“自下而上”的学习。阅读本书，再找一本针对具体应用程序的书，我想任何类型的应用程序的开发都应该难不倒你了。

1.7 通用类型系统

CLR 一切都围绕类型展开。到目前为止，这一点应该很清楚了。类型向应用程序和其他类型公开了功能。通过类型，用一种编程语言写的代码能与用另一种编程语言写的代码沟通。由于类型是 CLR 的根本，所以微软制定了一个正式的规范来描述类型的定义和行为，这就是“通用类型系统”(Common Type System，CTS)。

注意：微软事实上已将 CTS 和.NET Framework 的其他组件——包括文件格式、元数据、中间语言以及对底层平台的访问(P/Invoke)——提交给 ECMA 以完成标准化工作。最后形成的标准称为“公共语言基础结构”(Common Language Infrastructure，CLI)。除此之外，微软还提交了 Framework 类库的一部分、C#编程语言(ECMA-334)以及 C++/CLI 编程语言。要详细了解这些行业标准，请访问 ECMA 的 Technical Committee 39 专题网站：*http://www.ecma-international.org*。此外，微软还就 ECMA-334 和 ECMA-335 规范做出了社区承诺(Community Promise)，详情请访问 *https://tinyurl.com/2jj2868t*。

CTS 规范规定，一个类型可以包含零个或者多个成员。本书第 II 部分“设计类型”将更详细地讨论这些成员。目前只是简单地介绍一下它们。

- **字段(Field)**

  作为对象状态一部分的数据变量。字段使用名称和类型来加以标识。

- **方法(Method)**

  在对象上执行操作的函数，通常会改变对象状态。方法有一个名称、一个签名以及一个或多个修饰符。签名指定参数数量(及其顺序)；参数类型；方法是否有返回值；如果有返回值，还要指定返回值类型。

- **属性(Property)**

  对于调用者，属性看起来像是字段。但对于类型的实现者，属性看起来像是一个方法(或者两个方法[1])。属性允许在访问值之前校验输入参数和对象状态，以及/或者仅在必要时才计算某个值。属性还允许类型的用户采用简化的语法。最后，属性允许创建只读或只写的“字段”。

- **事件(Event)**

  事件在对象以及其他相关对象之间实现了通知机制。例如，利用按钮提供的一个事件，可以在按钮被单击之后通知其他对象。

① 即 getter 和 setter，或者取值方法和赋值方法。统称为“访问器方法”(accessor)——译注

CTS 还指定了类型可见性规则以及类型成员的访问规则。例如，如果将类型标记为 *public*(在 C#中使用 `public` 修饰符)，那么任何程序集都能看见并访问该类型。但是，如果标记为 *assembly*(在 C#中使用 `internal` 修饰符)，那么只有同一个程序集中的代码才能看见并访问该类型。所以，利用 CTS 制定的规则，程序集为一个类型建立了可视边界，CLR 则强制(贯彻)了这些规则。

调用者虽然能“看见”一个类型，但并不是说就能随心所欲地访问它的成员。可利用以下选项进一步限制调用者对类型中的成员的访问。

- ***private***
  成员只能由同一个类(class)类型中的其他成员访问。

- ***family***
  成员可由派生类型访问，不管那些类型是否在同一个程序集中。注意，许多语言(比如 C++和 C#)都用 `protected` 修饰符来标识 *family*。

- ***family and assembly***

  成员可由派生类型访问，但这些派生类型必须在同一个程序集中定义。许多语言(比如 C#和 Visual Basic)都没有提供这种访问控制。当然，IL 汇编语言不在此列。

- ***assembly***
  成员可由同一个程序集中的任何代码访问。许多语言都用 `internal` 修饰符来标识 *assembly*。

- ***family or assembly***

  成员可由任何程序集中的派生类型访问。成员也可由同一个程序集中的任何类型访问。C#用 `protected internal` 修饰符来标识 *family or assembly*。

- ***public***
  成员可由任何程序集中的任何代码访问。

除此之外，CTS 还为类型继承、虚方法、对象生存期等定义了相应的规则。这些规则在设计之初，便顺应了可以用现代编程语言来表示的语义。事实上，根本不需要专门学习 CTS 规则本身，因为你选择的语言会采用你熟悉的方式公开它自己的语言语法与类型规则。通过编译来生成程序集时，它会将语言特有的语法映射到 IL——也就是 CLR 的“语言”。

接触 CLR 后不久，我便意识到最好区别对待“代码的语言”和“代码的行为”。使用 C++/CLI 可以定义自己的类型，这些类型有它们自己的成员。当然，也可使用 C#或 Visual Basic 来定义相同的类型，并在其中添加相同的成员。使用的语言不同，用于定义类型的语法也不同。但是，无论使用哪一种语言，类型的行为都完全一致，因为最终是由 CLR 的 CTS 来定义类型的行为。

为了更形象地理解这一点，让我们来举一个例子。CTS 规定一个类型只能从一个基类派生

(单继承)。因此，虽然 C++语言允许一个类型继承自多个基类型(多继承)，但 CTS 既不能接受、也不能操作这样的类型。为了帮助开发人员，Microsoft 的 C++/CLI 编译器一旦检测到你试图创建的托管代码含有从多个基类型派生的类型，就会报错。

下面是另一条 CTS 规则：所有类型最终必须从预定义的 `System.Object` 类型继承。可以看出，`Object` 是 `System` 命名空间中定义的一个类型的名称。`Object` 是其他所有类型的根，因而保证了每个类型实例都有一组最基本的行为。具体地讲，`System.Object` 类型允许做下面这些事情。

- 比较两个实例的相等性。
- 获取实例的哈希码。
- 查询一个实例的真正类型。
- 执行实例的浅(按位)拷贝。
- 获取实例对象当前状态的字符串表示。

1.8 公共语言规范

以前在使用 COM 的时候，用不同语言创建的对象可以相互通信。现在，CLR 集成了所有语言，用一种语言创建的对象在另一种语言中，和用后者创建的对象具有相同地位。之所以能实现这样的集成，是因为 CLR 使用了标准类型集、元数据(自描述的类型信息)以及通用执行环境。

语言集成是一个宏伟的目标，最棘手的问题是各种编程语言存在极大区别。例如，有的语言不区分大小写，有的不支持无符号(unsigned)整数、操作符重载或者参数数量可变的方法。

要创建很容易从其他编程语言中访问的类型，只能从自己的语言中挑选其他所有语言都支持的功能。为了在这个方面提供帮助，Microsoft 定义了“公共语言规范”(Common Language Specification，CLS)，它详细定义了一个最小功能集。任何编译器只有支持这个功能集，生成的类型才能兼容由其他符合 CLS、面向 CLR 的语言生成的组件。

CLR/CTS 支持的功能比 CLS 定义的多得多，CLS 定义的只是一个子集。所以，如果不关心语言之间的互操作性，可以开发一套功能很全的类型，它们仅受你选择的那种语言的功能集的限制。具体地说，在开发类型和方法时，如果希望它们对外“可见”，能从符合 CLS 的任何编程语言中访问，就必须遵守 CLS 定义的规则。注意，假如代码只是从定义(这些代码的)程序集的内部访问，CLS 规则就不适用了。图 1-6 形象地展示了这一段想要表达的意思。

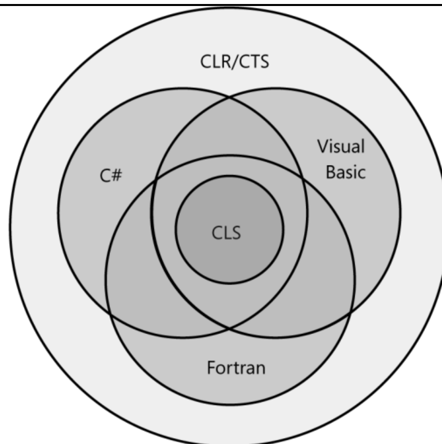


图 1-6 每种语言都提供了 CLR/CTS 的一个子集以及 CLS 的一个超集(但不一定是同一个超集)

如图 1-6 所示，CLR/CTS 提供了一个功能集。有的语言公开了 CLR/CTS 的一个较大的子集。如果开发人员用 IL 汇编语言写程序，那么可以使用 CLR/CTS 提供的全部功能。但是，其他大多数语言(比如 C#、Visual Basic 和 Fortran)只向开发人员公开了 CLR/CTS 的一个功能子集。CLS 定义了所有语言都必须支持的最小功能集。

用一种语言定义类型时，如果希望在另一种语言中使用该类型，就不要在该类型的 public 和 protected 成员中使用位于 CLS 外部的任何功能。否则，其他开发人员使用其他语言写代码时，就可能无法访问该类型的成员。

以下代码使用 C#定义一个符合 CLS 的类型。然而，类型中含有几个不符合 CLS 的构造，造成 C#编译器报错。

```
using System;

// 告诉编译器检查 CLS 相容性
[assembly: CLSCompliant(true)]

namespace SomeLibrary {
    // 因为是 public 类，所以才会显示后续的“警告”
    public sealed class SomeLibraryType {

        // 警告：SomeLibrary.SomeLibraryType.Abc()的返回类型不符合 CLS
        public UInt32 Abc() { return 0; }

        // 警告：仅大小写不同的标识符 SomeLibrary.SomeLibraryType.abc()
        // 不符合 CLS
        public void abc() { }

        // 不显示警告：该方法是私有的
        private UInt32 ABC() { return 0; }
    }
```

```
}
```

上述代码将[assembly:CLSCompliant(true)]这个特性[①]应用于程序集，告诉编译器检查其中的任何公开类型，判断是否存在任何不合适的构造阻止了从其他编程语言中访问该类型。上述代码编译时，C#编译器会显示两条警告消息。第一个警告是因为 Abc 方法返回无符号整数，一些语言是不能操作无符号整数值的。第二个警告是因为该类型公开了两个 public 方法，而这两个方法(Abc 和 abc)只是大小写和返回类型有别。Visual Basic 和其他一些语言无法区分这两个方法。

有趣的是，删除 sealed class SomeLibraryType 之前的 public 字样，然后重新编译，两个警告都会消失。因为这样一来，SomeLibraryType 类型将默认为 internal(而不是 public)，将不再向程序集的外部公开。要获得完整的 CLS 规则列表，请参考“跨语言互操作性”(*http://msdn.microsoft.com/zh-cn/library/730f1wy3.aspx*)。

现在提炼一下 CLS 的规则。在 CLR 中，类型的每个成员要么是字段(数据)，要么是方法(行为)。这意味着每一种编程语言都必须能访问字段和调用方法。字段和方法以特殊或通用的方式使用。为了简化编程，语言往往提供了额外的抽象，从而对这些常见的编程模式进行简化。例如，语言会公开枚举、数组、属性、索引器、委托、事件、构造器、终结器、操作符重载、转换操作符等概念。编译器在源代码中遇到其中任何一样，都必须将其转换为字段和方法，使 CLR 和其他任何编程语言能够访问这些构造。

以下类型定义包含一个构造器、一个终结器、一些重载的操作符、一个属性、一个索引器和一个事件。注意，目的只是让代码能通过编译，不代表类型的正确实现方式。

```
using System;

internal sealed class Test {
    // 构造器
    public Test() {}

    // 终结器
    ~Test() {}

    // 操作符重载
    public static Boolean operator == (Test t1, Test t2) {
        return true;
    }
    public static Boolean operator != (Test t1, Test t2) {
        return false;
    }

    // 操作符重载
    public static Test operator + (Test t1, Test t2) { return null; }
```

① 本书按照文档将 attribute 翻译成“特性”。——译注

```
        // 属性
        public String AProperty {
            get { return null; }
            set { }
        }

        // 索引器
        public String this[Int32 x] {
            get { return null; }
            set { }
        }

        // 事件
        event EventHandler AnEvent;
    }
```

编译上述代码得到含有大量字段和方法的一个类型。可以使用.NET Framework SDK 提供的 IL 反汇编器工具(ILDasm.exe)检查最终生成的托管模块，如图 1-7 所示。

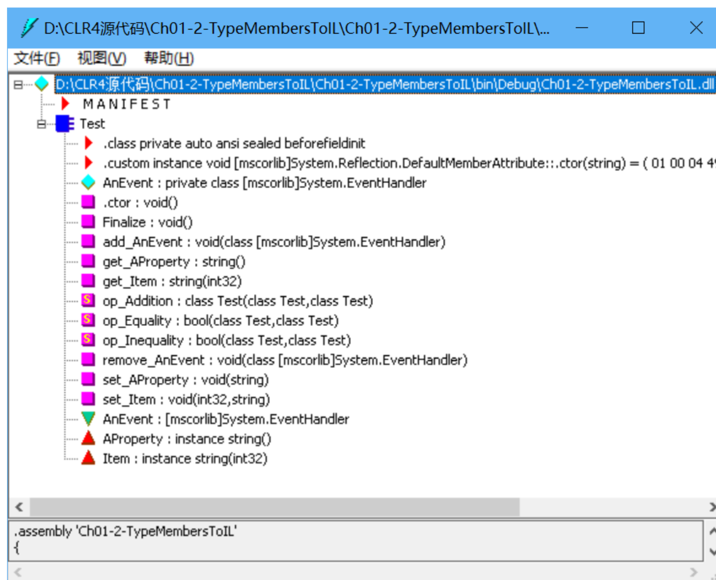


图 1-7 ILDasm 显示了 Test 类型的字段和方法(从元数据中获取)

表 1-4 总结了编程语言的各种构造与 CLR 字段/方法的对应关系。

表 1-4 Test 类型的字段和方法(从元数据中获取)

类型的成员	成员的类型	对应的编程语言构造
AnEvent	字段	事件；字段名是 AnEvent，类型是 System.EventHandler

.ctor	方法	构造器
Finalize	方法	终结器
add_AnEvent	方法	事件的 add 访问器方法
get_AProperty	方法	属性的 get 访问器方法
get_Item	方法	索引器的 get 访问器方法
op_Addition	方法	+操作符
op_Equality	方法	==操作符
op_Inequality	方法	!=操作符
remove_AnEvent	方法	事件的 remove 访问器方法
set_AProperty	方法	属性的 set 访问器方法
set_Item	方法	索引器的 set 访问器方法

Test 类型还有另一些节点未在表 1-4 中列出，包括.class，.custom，AnEvent，AProperty 以及 Item——它们标识了类型的其他元数据。这些节点不映射到字段或方法，只是提供了类型的一些额外信息，供 CLR、编程语言或者工具访问。例如，某个工具可以检测到 Test 类型提供了一个名为 AnEvent 的事件，该事件借由两个方法(add_AnEvent 和 remove_AnEvent)公开。

1.9 与非托管代码的互操作性

.NET Framework 提供了其他开发平台所不具备的许多优势。但是，能下定决心重新设计和重新实现全部现有代码的公司并不多。Microsoft 也知道这个问题，并通过 CLR 来提供了一些机制，允许在应用程序中同时包含托管和非托管代码。具体地说，CLR 支持三种互操作情形。

- **托管代码能调用 DLL 中的非托管函数**

  托管代码通过 P/Invoke(Platform Invoke)机制调用 DLL 中的函数。毕竟，FCL 中定义的许多类型都要在内部调用从 Kernel32.dll、User32.dll 等导出的函数。许多编程语言都提供了机制方便托管代码调用 DLL 中的非托管函数。例如，C#应用程序可调用从 Kernel32.dll 导出的 `CreateSemaphore` 函数。

- **托管代码可以使用现有 COM 组件(服务器)**

  许多公司都已经实现了大量非托管 COM 组件。利用来自这些组件的类型库，可以创建一个托管程序集来描述 COM 组件。托管代码可像访问其他任何托管类型一样访问托管程序集中的类型。这方面的详情可以参考.NET Framework SDK 提供的 TlbImp.exe 工具。有时可能没有类型库，或者想对 TlbImp.exe 生成的内容进行更多控制。这时可在源代码中手动构建一个类型，使 CLR 能用它实现正确的互操作性，例如可从 C#应用程序中使用 DirectX COM 组件。

- **非托管代码可以使用托管类型(服务器)**

  许多现有的非托管代码要求提供 COM 组件来确保代码正确工作。使用托管代码可以更简单地实现这些组件，避免所有代码都不得不和引用计数以及接口打交道。例如，可以使用 C#创建 ActiveX 控件或 shell 扩展。这方面的详情可以参考.NET Framework SDK 提供的 TlbExp.exe 和 RegAsm.exe 工具。

注意：为了方便需要与本机代码交互的开发人员，微软公开了 Type Library Importer 工具和 P/Invoke Interop Assistant 工具的源代码。访问 *https://github.com/clrinterop* 下载这些工具及其源代码。

微软从 Windows 8 开始引入了称为 Windows Runtime(WinRT)的新 Windows API。该 API 内部通过 COM 组件来实现。但是，COM 组件不是使用类型库文件，而是使用.NET Framework 团队创建的元数据 ECMA 标准来描述其 API。好处是用一种.NET 语言写的代码(在很大程度上)能与 WinRT API 无缝对接。CLR 在幕后执行需要的所有 COM 互操作，不要求你使用任何额外的工具。第 25 章“与 WinRT 组件互操作”将进一步详细讲解。

第 2 章 生成、打包、部署和管理应用程序及类型

本章内容：

- .NET Framework 部署目标
- 将类型生成到模块中
- 元数据概述
- 将模块合并成程序集
- 程序集版本资源信息
- 语言文化
- 简单应用程序部署(私有部署的程序集)
- 简单管理控制(配置)

在解释如何为 Microsoft .NET Framework 开发程序之前，首先讨论一下生成、打包和部署应用程序及其类型的步骤。本章重点解释如何生成仅供自己的应用程序使用的程序集。第 3 章“共享程序集和强命名程序集”将讨论更高级的概念，包括如何生成和使用程序集，使其中包含的类型能由多个应用程序共享。这两章会谈及管理员可以采取什么方式来影响应用程序及其类型的执行。

当今的应用程序都由多个类型构成，这些类型通常是由你和 Microsoft 创建的。除此之外，作为一个新兴产业，组件厂商们也纷纷着手构建一些专用类型，并将其出售给各大公司，以缩短软件项目的开发时间。开发这些类型时，如果使用的语言是面向 CLR 的，这些类型就能无缝地协同工作。也就是说，用一种语言写的类型可以将另一个类型作为自己的基类使用，不用关心基类是用什么语言开发的。

本章将解释如何生成这些类型，并将其打包到文件中以进行部署。另外，还会提供一个简短的历史回顾，帮助开发人员理解.NET Framework 希望解决的某些问题。

2.1 .NET Framework 部署目标

Windows 多年来一直因为不稳定和过于复杂而口碑不佳。不管对它的评价对不对，之所以

造成这种状况，要归咎于几方面的原因。首先，所有应用程序都使用来自 Microsoft 或其他厂商的动态链接库(Dynamic-Link Library，DLL)。由于应用程序要执行多个厂商的代码，所以任何一段代码的开发人员都不能百分之百保证别人以什么方式使用这段代码。虽然这种交互可能造成各种各样的麻烦，但实际一般不会出太大的问题，因为应用程序在部署前会进行严格测试和调试。

但对于用户，当一家公司决定更新其软件产品的代码，并将新文件发送给他们时，就可能出问题。新文件理论上应该向后兼容以前的文件，但谁能对此保证呢？事实上，一家厂商更新代码时，经常都不可能重新测试和调试之前发布的所有应用程序，无法保证自己的更改不会造成不希望的结果。

很多人都可能遭遇过这样的问题：安装新应用程序时，它可能莫名其妙破坏了另一个已经安装好的应用程序。这就是所谓的“DLL hell”。这种不稳定会对普通计算机用户带来不小的困扰。最终结果是用户必须慎重考虑是否安装新软件。就我个人来说，有一些重要的应用程序是平时经常都要用到的。为了避免对它们产生不好的影响，我不会冒险去“尝鲜”。

造成 Windows 口碑不佳的第二个原因是安装的复杂性。如今，大多数应用程序在安装时都会影响到系统的全部组件。例如，安装一个应用程序会将文件复制到多个目录，更新注册表设置，并在桌面和“开始”菜单上安装快捷方式。问题是，应用程序不是一个孤立的实体。应用程序备份不易，因为必须复制应用程序的全部文件以及注册表中的相关部分。除此之外，也不能轻松地将应用程序从一台机器移动到另一台。只有再次运行安装程序，才能确保所有文件和注册表设置的正确性。最后，即使卸载或移除了应用程序，也免不了担心它的一部分内容仍潜伏在我们的机器中。

第三个原因涉及安全性。应用程序安装时会带来各种文件，其中许多是由不同的公司开发的。此外，Web 应用程序经常会悄悄下载一些代码(比如 ActiveX 控件)，用户根本注意不到自己的机器上安装了这些代码。如今，这种代码能够执行任何操作，包括删除文件或者发送电子邮件。用户完全有理由害怕安装新的应用程序，因为它们可能造成各种各样的危害。考虑到用户的感受，安全性必须集成到系统中，使用户能够明确允许或禁止各个公司开发的代码访问自己的系统资源。

阅读本章和下一章可以知道，.NET Framework 正在尝试彻底解决 DLL hell 的问题。另外，.NET Framework 还在很大程度上解决了应用程序状态在用户硬盘中四处分散的问题。例如，和 COM 不同，类型不再需要注册表中的设置。但遗憾的是，应用程序还是需要快捷方式。安全性方面，.NET Framework 包含称为“代码访问安全性”(Code Access Security)的安全模型。Windows 安全性基于用户身份，而代码访问安全性允许宿主设置权限，控制加载的组件能做的事情。像 Microsoft SQL Server 这样的宿主应用程序只能将少许权限授予代码，而本地安装的(自寄宿)应用程序可以获得完全信任(全部权限)。以后会讲到，.NET Framework 允许用户灵活地控制哪些东西能够安装，哪些东西能够运行。他们对自己机器的控制上升到一个前所未有的高度。

2.2 将类型生成到模块中

本节讨论如何将包含多个类型的源代码文件转变为可以部署的文件。先看下面这个简单的应用程序：

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

该应用程序定义了 Program 类型，其中有名为 Main 的 public static 方法。Main 中引用了另一个类型 System.Console。System.Console 是 Microsoft 实现好的类型，用于实现这个类型的各个方法的 IL 代码存储在 MSCorLib.dll 文件中。总之，该应用程序定义了一个类型，还使用了其他公司提供的类型。

为了生成这个示例应用程序，请将上述代码放到一个源代码文件中(假定为 Program.cs)，然后在命令行执行以下命令：

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

这个命令行指示 C# 编译器生成名为 Program.exe 的可执行文件(/out:Program.exe)。生成的文件是 Win32 控制台应用程序类型(/t[arget]:exe)。

C#编译器处理源文件时，发现代码引用了 System.Console 类型的 WriteLine 方法。此时，编译器要核实该类型确实存在，它确实有 WriteLine 方法，而且传递的实参与方法形参匹配。由于该类型在 C#源代码中没有定义，所以要顺利通过编译，必须向 C#编译器提供一组程序集，使它能解析对外部类型的引用。在上述命令行中，我添加了 /r[eference]:MSCorLib.dll 开关，告诉编译器在 MSCorLib.dll 程序集中查找外部类型。

MSCorLib.dll 是特殊文件，它包含所有核心类型，包括 Byte，Char，String，Int32 等等。事实上，由于这些类型使用得如此频繁，以至于 C#编译器会自动引用 MSCorLib.dll 程序集。换言之，命令行其实可以简化成下面这样(省略/r 开关)：

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

此外，由于/out:Program.exe 和/t:exe 开关是 C#编译器的默认设定，所以能继续简化成以下形式：

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

如果因为某个原因不想让 C#编译器自动引用 MSCorLib.dll 程序集，可以使用/nostdlib 开关。Microsoft 生成 MSCorLib.dll 程序集自身时便使用了这个开关。例如，用以下命令行编译 Program.cs 会报错，因为它使用的 System.Console 类型是在 MSCorLib.dll 中定义的：

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

现在更深入地思考一下 C#编译器生成的 Program.exe 文件。这个文件到底是什么？首先，它是标准 PE(可移植执行体，Portable Executable)文件。这意味着运行 32 位或 64 位 Windows 的计算机能加载它，并能通过它执行某些操作。Windows 支持三种应用程序。生成控制台用户界面(Console User Interface，CUI)应用程序使用`/t:exe` 开关；生成图形用户界面(Graphical User Interface，GUI)应用程序使用`/t:winexe` 开关；生成 Windows Store 应用使用`/t:appcontainerexe` 开关。

响应文件

结束对编译器开关的讨论之前，让我们花点时间了解一下**响应文件**(response file)。响应文件是包含一组编译器命令行开关的文本文件。执行 CSC.exe 时，编译器打开响应文件，并使用其中包含的所有开关，感觉就像是这些开关直接在命令行上传递给 CSC.exe。要告诉编译器使用响应文件，在命令行中，请在@符号之后指定响应文件的名称。例如，假定响应文件 MyProject.rsp 包含以下文本：

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

为了让 CSC.exe 使用这些设置，可以像下面这样调用它：

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

这就告诉了 C#编译器输出文件的名称和要创建哪种类型的应用程序。可以看出，响应文件能带来一些便利，不必每次编译项目时都手动指定命令行参数。

C#编译器支持多个响应文件。除了在命令行上显式指定的文件，编译器还会自动查找名为 CSC.rsp 的文件。CSC.exe 运行时，会在 CSC.exe 所在的目录查找全局 CSC.rsp 文件。应该将你想应用于自己的所有项目的设置放到其中。编译器汇总并使用所有响应文件中的设置。本地和全局响应文件中的某个设置发生冲突，将以本地设置为准。类似地，命令行上显式指定的设置将覆盖本地响应文件中的设置。

.NET Framework 安装时会在%SystemRoot%\Microsoft.NET\Framework(*64*)\v*X.X.X* 目录中安装默认全局 CSC.rsp 文件(*X.X.X* 是你安装的.NET Framework 的版本号)。这个文件的最新版本包含以下开关：

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

# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.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.Data.OracleClient.dll
/r:System.Deployment.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
/r:System.Messaging.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceModel.dll
/r:System.ServiceModel.Web.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.dll
/r:System.Web.Extensions.Design.dll
/r:System.Web.Extensions.dll
/r:System.Web.Mobile.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.Dll
/r:System.Workflow.Activities.dll
/r:System.Workflow.ComponentModel.dll
/r:System.Workflow.Runtime.dll
/r:System.Xml.dll
/r:System.Xml.Linq.dll
```

由于全局CSC.rsp文件引用了列出的所有程序集，所以不必使用C#编译器的/reference开关显式引用这些程序集。这个响应文件为开发人员带来了极大的方便，因为可以直接使用Microsoft发布的各个程序集中定义的类型和命名空间，不必每次编译时都指定/reference编译器开关。

引用所有这些程序集对编译器的速度有一点影响。但是，如果源代码没有引用上述任何程序集定义的类型或成员，就不会影响最终的程序集文件，也不会影响程序的执行性能。

当然，要进一步简化操作，还可以在全局 CSC.rsp 文件中添加自己的开关。但这样一来，在其他机器上重现代码的生成环境就比较困难了：在每台用于生成的机器上，都必须以相同方式更新 CSC.rsp。另外，指定/noconfig 命令行开关，编译器将忽略本地和全局CSC.rsp文件。

注意：用/reference 编译器开关引用程序集时，可以指定目标文件的完整路径。然

而，如果不指定路径，编译器会在以下位置查找文件(按所列顺序)

1. 工作目录。
2. CSC.exe 所在的目录。MSCorLib.dll 总是在该目录中。目录路径形如%SystemRoot%\Microsoft.NET\Framework\v4.0.#####。
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。

2.3 元数据概述

现在，我们知道了创建的是什么类型的 PE 文件。但是，Program.exe 文件中到底有什么？托管 PE 文件由 4 部分构成：PE32(+)头、CLR 头、元数据以及 IL。PE32(+)头是 Windows 要求的标准信息。CLR 头是一个小的信息块，这些信息是需要 CLR 的模块(托管模块)所特有的。这个头包含模块生成时所面向的 CLR 的 major(主)和 minor(次)版本号；一些标志(flag)；一个`MethodDef` token(稍后详述)，该 token 指定了模块的入口方法(前提是该模块是 CUI、GUI 或 Windows Store 执行体)；一个可选的强名称数字签名(将在第 3 章讨论)。最后，CLR 头还包含模块内部的一些元数据表的大小和偏移量。可以查看 CorHdr.h 头文件定义的`IMAGE_COR20_HEADER`来了解 CLR 头的具体格式。

元数据是由几个表构成的二进制数据块。有三种表，分别是定义表(definition table)、引用表(reference table)和清单表(manifest table)。表 2-1 总结了模块元数据块中常用的定义表。

表 2-1 常用的元数据定义表

元数据定义表名称	说明
`ModuleDef`	总是包含对模块进行标识的一个记录项。该记录项包含模块文件名和扩展名(不含路径)，以及模块版本 ID(形式为编译器创建的 GUID)。这样可在保留原始名称记录的前提下自由重命名文件。但强烈反对重命名文件，因为可能妨碍 CLR 在运行时正确定位程序集
`TypeDef`	模块定义的每个类型在这个表中都有一个记录项。每个记录项都包含类型的名称、基类型、一些标志(`public`，`private` 等)以及一些索引，这些索引指向 MethodDef 表中该类型的方法、FieldDef 表中该类型的字段、PropertyDef 表中该类型的属性以及 EventDef 表中该类型的事件
`MethodDef`	模块定义的每个方法在这个表中都有一个记录项。每个记录项都包含方法的名称、一些标志(`private`，`public`，`virtual`，`abstract`，`static`，`final` 等)、签名以及方法的 IL 代码在模块中的偏移量。每个记录项还引用了 ParamDef 表中的一个记录项，后者包括与方法参数有关的更多信息

FieldDef	模块定义的每个字段在这个表中都有一个记录项。每个记录项都包含标志(private，public 等)、类型和名称
ParamDef	模块定义的每个参数在这个表中都有一个记录项。每个记录项都包含标志(in，out，retval 等)、类型和名称
PropertyDef	模块定义的每个属性在这个表中都有一个记录项。每个记录项都包含标志、类型和名称
EventDef	模块定义的每个事件在这个表中都有一个记录项。每个记录项都包含标志和名称

编译器编译源代码时，代码定义的任何东西都导致在表 2-1 列出的某个表中创建一个记录项。此外，编译器还会检测源代码引用的类型、字段、方法、属性和事件，并创建相应的元数据表记录项。在创建的元数据中包含一组引用表，它们记录了所引用的内容。表 2-2 总结了常用的引用元数据表。

表 2-2 常用的引用元数据表

引用元数据表名称	说明
AssemblyRef	模块引用的每个程序集在这个表中都有一个记录项。每个记录项都包含绑定[①]该程序集所需的信息：程序集名称(不含路径和扩展名)、版本号、语言文化(culture)以及公钥 token(根据发布者的公钥生成的一个小的哈希值，标识了所引用程序集的发布者)。每个记录项还包含一些标志(flag)和一个哈希值。该哈希值本应作为所引用程序集的二进制数据的校验和(checksum)来使用。但是，目前 CLR 完全忽略该哈希值，未来的 CLR 可能同样如此
ModuleRef	实现该模块所引用的类型的每个 PE 模块在这个表中都有一个记录项。每个记录项都包含模块的文件名和扩展名(不含路径)。可能是别的模块实现了你需要的类型，这个表的作用便是建立同那些类型的绑定关系
TypeRef	模块引用的每个类型在这个表中都有一个记录项。每个记录项都包含类型的名称和一个引用(指向类型的位置)。如果类型在另一个类型中实现，引用指向一个 TypeRef 记录项。如果类型在同一个模块中实现，引用指向一个 ModuleDef 记录项。如果类型在调用程序集内的另一个模块中实现，引用指向一个 ModuleRef 记录项。如果类型在不同的程序集中实现，引用指向一个 AssemblyRef 记录项
MemberRef	模块引用的每个成员(字段和方法，以及属性方法和事件方法)在这个表中都有一个记录项。每个记录项都包含成员的名称和签名，并指向对成员进行定义的那个类型的 TypeRef 记录项

① bind 在文档中有时翻译成“联编”，binder 有时翻译成“联编程序”。本书采用“绑定”和“绑定器”。——译注

除了表 2-1 和表 2-2 所列的，还有其他许多定义表和引用表。但是，我的目的只是让你体会一下编译器在生成的元数据中添加的各种信息。前面还提到了清单(manifest)元数据表，本章稍后会讨论这种表。

可以使用多种工具检查托管 PE 文件中的元数据。我个人喜欢使用 ILDasm.exe，即 IL Disassembler(IL 反汇编器)。要查看元数据表，请执行以下命令行：

```
ILDasm Program.exe
```

ILDasm.exe 将运行并加载 Program.exe 程序集。要采用一种美观的、容易阅读的方式查看元数据，请选择“视图”|“元信息”|“显示!”菜单项(或直接按 Ctrl+M 组合键)。随后会显示以下信息(基于.NET Framework 4.8，用 Release 模式生成，做了一定删减和排版)：

```
============================================================
ScopeName : Program.exe
MVID      : {1E2A35B5-C81A-4418-B361-3E66954A70F1}
============================================================
Global functions
-------------------------------------------------------

Global fields
-------------------------------------------------------

Global MemberRefs
-------------------------------------------------------

TypeDef #1 (02000002)
-------------------------------------------------------
    TypDefName: Program  (02000002)
    Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
                [BeforeFieldInit]  (00100101)
    Extends   : 01000010 [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.Runtime.CompilerServices.CompilationRelaxationsAttribute
    MemberRef #1 (0a000001)
    -------------------------------------------------------
        Member: (0a000001) .ctor:
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        1 Arguments
            Argument #1:  I4

TypeRef #2 (01000002)
-------------------------------------------------------
Token:             0x01000002
ResolutionScope:   0x23000001
TypeRefName:       System.Runtime.CompilerServices.RuntimeCompatibilityAttribute
    MemberRef #1 (0a000002)
    -------------------------------------------------------
        Member: (0a000002) .ctor:
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        No arguments.

...

TypeRef #16 (01000010)
-------------------------------------------------------
Token:             0x01000010
ResolutionScope:   0x23000001
TypeRefName:       System.Object
    MemberRef #1 (0a000010)
    -------------------------------------------------------
        Member: (0a000010) .ctor:
        CallCnvntn: [DEFAULT]
        hasThis
        ReturnType: Void
        No arguments.

TypeRef #17 (01000011)
-------------------------------------------------------
Token:             0x01000011
```

```
ResolutionScope:   0x23000001
TypeRefName:       System.Console
    MemberRef #1 (0a00000f)
    -------------------------------------------------------
        Member: (0a00000f) WriteLine:
        CallCnvntn: [DEFAULT]
        ReturnType: Void
        1 Arguments
            Argument #1:  String

Assembly
-------------------------------------------------------
    Token: 0x20000001
    Name : Program
    Public Key    :
    Hash Algorithm : 0x00008004
    Version: 1.0.0.0
    Major Version: 0x00000001
    Minor Version: 0x00000000
    Build Number: 0x00000000
    Revision Number: 0x00000000
    Locale: <null>
    Flags : [none] (00000000)
    CustomAttribute #1 (0c000001)
    -------------------------------------------------------
        CustomAttribute Type: 0a000001
        CustomAttributeName:
          System.Runtime.CompilerServices.CompilationRelaxationsAttribute ::
            instance void .ctor(int32)
        Length: 8
        Value : 01 00 08 00 00 00 00 00                          >                <
        ctor args: (8)

    CustomAttribute #2 (0c000002)
    -------------------------------------------------------
        CustomAttribute Type: 0a000002
        CustomAttributeName:
          System.Runtime.CompilerServices.RuntimeCompatibilityAttribute ::
            instance void .ctor()
        Length: 30
        Value : 01 00 01 00 54 02 16 57  72 61 70 4e 6f 6e 45 78 >    T  WrapNonEx<
              : 63 65 70 74 69 6f 6e 54  68 72 6f 77 73 01    >ceptionThrows   <
        ctor args: ()

...

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)


User Strings
-------------------------------------------------------
70000001 : ( 2) L"Hi"


Coff symbol name overhead:  0
===========================================================
===========================================================
===========================================================
```

幸好ILDasm处理了元数据表，恰当合并了信息，避免我们跑去分析原始的表信息。例如，可以看到，当ILDasm显示一个TypeDef记录项时，会在第一个TypeRef项之前显示对应的成员定义信息。

不用完全理解上面显示的一切。重点是Program.exe包含名为`Program`的TypeDef。`Program`是公共密封类，从`System.Object`派生(`System.Object`是引用的另一个程序集中的类型)。`Program`类型还定义了两个方法：`Main`和`.ctor`(构造器)。

`Main`是公共静态方法，用IL代码实现(有的方法可能用本机CPU代码实现，比如x86代码)。`Main`的返回类型是`void`，无参。构造器(名称始终是`.ctor`)是公共方法，也用IL代码实现。构造器的返回类型是`void`，无参，有一个`this`指针(指向调用方法时要构造的对象内存)。

强烈建议多试验一下 ILDasm。它提供了丰富的信息。你对自己看到的东西理解得越多，对CLR及其功能的理解就越好。本书后面会大量地用到ILDasm。

为了增加趣味性，来看看Program.exe程序集的统计信息。在ILDasm中选择“视图”|“统计”，会显示以下信息：

```
File size            : 4608
 PE header size       : 512 (496 used)    (11.11%)
 PE additional info   : 1663              (36.09%)
 Num.of PE sections   : 3
 CLR header size     : 72                 ( 1.56%)
 CLR meta-data size  : 1428               (30.99%)
 CLR additional info : 0                  ( 0.00%)
 CLR method headers  : 2                  ( 0.04%)
 Managed code         : 18                ( 0.39%)
 Data                 : 2048              (44.44%)
```

```
Unaccounted          : -1135            (-24.63%)

Num.of PE sections   : 3
  .text    - 2048
  .rsrc    - 1536
  .reloc   - 512

CLR meta-data size  : 1428
  Module        -    1 (10 bytes)
  TypeDef       -    2 (28 bytes)      0 interfaces, 0 explicit layout
  TypeRef       -   17 (102 bytes)
  MethodDef     -    2 (28 bytes)      0 abstract, 0 native, 2 bodies
  MemberRef     -   16 (96 bytes)
  CustomAttribute-   14 (84 bytes)
  Assembly      -    1 (22 bytes)
  AssemblyRef   -    1 (20 bytes)
  Strings       -   580 bytes
  Blobs         -   268 bytes
  UserStrings   -     8 bytes
  Guids         -    16 bytes
  Uncategorized -   166 bytes

CLR method headers : 2
  Num.of method bodies  - 2
  Num.of fat headers    - 0
  Num.of tiny headers   - 2

Managed code : 18
  Ave method size - 9
```

从中可以看出文件大小(字节数)以及文件各部分大小(字节数和百分比)。对于这个如此小的Program.exe应用程序，PE头和元数据占了相当大的比重。事实上，IL代码只有区区18字节。当然，随着应用程序规模的增大，它会重用大多数类型以及对其他类型和程序集的引用，元数据和头信息在整个文件中的比重越来越小。

注意：顺便说一下，ILDasm.exe 的一个 bug 会影响显示的文件长度。尤其不要相信 Unaccounted 信息。

2.4 将模块合并成程序集

上一节讨论的 Program.exe 并非只是含有元数据的 PE 文件，它还是**程序集**(assembly)。程序集是一个或多个类型定义文件及资源文件的集合。在程序集的所有文件中，有一个文件容纳了**清单**(manifest)。清单也是一个元数据表集合，表中主要包含作为程序集组成部分的那些文件的名称。此外，还描述了程序集的版本、语言文化、发布者、公开导出的类型以及构成程序集的所有文件。

CLR 操作的是程序集。换言之，CLR 总是首先加载包含“清单”元数据表的文件，再根据“清单”来获取程序集中的其他文件的名称。下面是你应该记住的程序集的一些重要特征。

- 程序集定义了可重用的类型。
- 程序集用一个版本号标记。
- 程序集可以关联安全信息。

除了包含清单元数据表的文件，程序集中其他单独的文件都不包含这些特性值。

类型为了顺利地进行打包、版本控制、安全保护以及使用，必须放在作为程序集一部分的模块中。程序集大多数时候只有一个文件，就像前面的 Program.exe 那样。然而，程序集还可以由多个文件构成：一些是含有元数据的 PE 文件，另一些是.gif 或.jpg 这样的资源文件。为便于理解，可将程序集视为一个逻辑 EXE 或 DLL。

微软为什么引入“程序集”的概念？这是因为使用程序集，可重用类型的逻辑表示与物理表示就可以分开。例如，程序集可能包含多个类型。可以将常用类型放到一个文件中，不常用类型放到另一个文件中。如果程序集要从 Internet 下载并部署，那么对于含有不常用类型的文件，假如客户端永远不使用那些类型，该文件就永远不会下载到客户端。例如，擅长制作 UI 控件的一家独立软件开发商(Independent Software Vendor，ISV)可选择在单独的模块中实现 Active Accessibility 类型(以满足 Microsoft 徽标认证授权要求)[①]。这样一来，只有需要额外“无障碍访问”功能的用户才需要下载该模块。

为了配置应用程序去下载程序集文件，可以在应用程序配置文件中指定 `codeBase` 元素(详见第 3 章)。在 `codeBase` 元素定义的 URL 所指向的位置，可以找到程序集的所有文件。试图加载程序集的一个文件时，CLR 获取 `codeBase` 元素的 URL，检查机器的下载缓存，判断文件是否存在。如果是，直接加载文件。如果不是，CLR 去 URL 指向的位置将文件下载到缓存。如果还是找不到文件，CLR 在运行时抛出 `FileNotFoundException` 异常。

---

① Microsoft Active Accessibility®是一种基于 COM 的技术，能够为应用程序和 Active Accessibility 客户端提供标准、一致的机制来交换信息。宗旨是帮助残障人士更有效地使用计算机。——译注

我想指出使用多文件程序集的三点理由。

- 可以使用单独的文件对类型进行划分，使文件能以“增量”方式下载(就像前面在 Internet 下载的例子中描述的那样)。另外，将类型划分到不同的文件中，可以对购买和安装的应用程序进行部分或分批打包/部署。也就是说，可以只更新或替换特定文件，而不必重新部署整个应用程序。

- 可在程序集中添加资源或数据文件。例如，假定一个类型的作用是计算保险信息，需要访问精算表才能完成计算。在这种情况下，不必在自己的源代码中嵌入精算表。相反，可以使用一个工具(比如稍后要讨论的程序集链接器 AL.exe)，使数据文件成为程序集的一部分。顺便说一句，数据文件可以为任意格式——包括文本文件、Microsoft Excel 电子表格文件以及 Microsoft Word 表格等——只要应用程序知道如何解析。

- 程序集包含的各个类型可以用不同的编程语言来实现。例如，一些类型可以用 C#实现，一些用 Visual Basic 实现，其他则用其他语言实现。编译用 C#写的类型时，编译器会生成一个模块。编译用 Visual Basic 写的类型时，编译器会生成另一个模块。然后，可以用工具将所有模块合并成单个程序集。其他开发人员在使用这个程序集时，只知道这个程序集包含了一系列类型，根本不知道、也不用知道这些类型分别是用什么语言写的。顺便说一句，如果愿意，可以对每个模块都运行 ILDasm.exe，获得相应的 IL 源代码文件。然后运行 ILAsm.exe，将所有 IL 源代码文件都传给它。随后，ILAsm.exe 会生成包含全部类型的单个文件。该技术的前提是源代码编译器能生成纯 IL 代码。

重要提示：总之，程序集是进行重用、版本控制和应用安全性设置的基本单元。它允许将类型和资源文件划分到单独的文件中。这样一来，无论你自己，还是你的程序集的用户，都可以决定打包和部署哪些文件。一旦 CLR 加载含有清单的文件，就可以确定在程序集的其他文件中，具体是哪些文件了包含应用程序所引用的类型和资源。程序集的用户(其他开发人员)只需知道含有清单的那个文件的名称。这样一来，文件的具体划分方式在程序集的用户那里就是完全透明的。你以后可以自由更改，不会干扰应用程序的行为。

如果多个类型能共享相同的版本号和安全性设置，那么建议将所有这些类型放到同一个文件中，而不是分散到多个文件中，更不要分散到多个程序集中。这是出于对性能的考虑。每次加载文件或程序集，CLR 和 Windows 都要花费一定的时间来查找、加载并初始化程序集。需要加载的文件/程序集的数量越少，性能越好，因为加载较少的程序集有助于减小工作集(working set)，并缓解进程地址空间的碎片化。最后，NGen.exe 在处理较大的文件时可以进行更好的优化。

在生成程序集时，要么选择现有的 PE 文件作为“清单”的宿主，要么创建单独的 PE 文件

并只在其中包含清单。表 2-3 展示了将托管模块转换成程序集时要用到的各种清单元数据表。

表 2-3 清单元数据表

清单元数据表名称	说明
AssemblyDef	如果模块标识的是程序集，这个元数据表就包含单一记录项来列出程序集名称(不含路径和扩展名)、版本(major，minor，build 和 revision)、语言文化(culture)、一些标志(flag)、哈希算法以及发布者公钥(可为 null)
FileDef	作为程序集一部分的每个 PE 文件和资源文件在这个表中都有一个记录项(清单本身所在的文件除外，该文件在 AssemblyDef 表的单一记录项中列出)。在每个记录项中，都包含文件名和扩展名(不含路径)、哈希值和一些标志(flags)。如果程序集只包含它自己的文件，FileDef 表将无记录①
ManifestResourceDef	作为程序集一部分的每个资源在这个表中都有一个记录项。记录项中包含资源名称、一些标志(如果在程序集外部可见，就为 public；否则为 private)以及 FileDef 表的一个索引(指出资源或流包含在哪个文件中)。如果资源不是独立文件(比如.jpg 或者.gif 文件)，那么资源就是包含在 PE 文件中的流。对于嵌入资源，记录项还包含一个偏移量，指出资源流在 PE 文件中的起始位置
ExportedTypesDef	从程序集的所有 PE 模块中导出的每个 public 类型在这个表中都有一个记录项。记录项中包含类型名称、FileDef 表的一个索引(指出类型由程序集的哪个文件实现)以及 TypeDef 表的一个索引。注意，为节省空间，从清单所在文件导出的类型不再重复，因为可通过元数据的 TypeDef 表获取类型信息

正是因为有了清单的存在，所以程序集的用户不必关心程序集的划分细节。另外，清单也使程序集具有了自描述性(self-describing)。另外，在包含清单的文件中，一些元数据信息描述了哪些文件是程序集的一部分。但是，那些文件本身并不包含元数据来指出它们是程序集的一部分。

注意：包含清单的程序集文件还有一个 AssemblyRef 表。由程序集的全部文件所引用的每个程序集在这个表中都有一个记录项。这样一来，工具只需打开程序集的清单，就可以知道它引用的所有程序集，而不必打开程序集的其他文件。同样地，AssemblyRef 表的存在加强了程序集的自描述性。

① 所谓“如果程序集只包含它自己的文件”，是指程序集只包含它的主模块，不包含其他非主模块和资源文件。1.2 节说过，程序集是一个抽象概念，是一个或者多个模块文件和资源文件组成的逻辑单元，其中包含一个且只能有一个作为主模块的.exe 或.dll 文件。——译注

指定以下任何命令行开关，C#编译器都会生成程序集：`/t[arget]:exe`，`/t[arget]:winexe`，`/t[arget]:appcontainerexe`，`/t[arget]:library` 或者 `/t[arget]:winmdobj`[①]。所有这些开关都会造成编译器生成含有清单元数据表的 PE 文件。这些开关分别生成 CUI 执行体、GUI 执行体、Windows Store 执行体、类库或者 WINMD 库。

除了这些开关，C#编译器还支持`/t[arget]:module` 开关。这个开关指示编译器生成一个不包含清单元数据表的 PE 文件。这样生成的肯定是一个 DLL PE 文件。CLR 要想访问其中的任何类型，必须先将该文件添加到一个程序集中。使用`/t:module` 开关时，C#编译器默认为输出文件使用.netmodule 扩展名。

重要提示：遗憾的是，不能直接从 Microsoft Visual Studio 集成开发环境(IDE)中创建多文件程序集。只能用命令行工具创建多文件程序集。

可以通过多种方式将模块添加到程序集。如果用 C#编译器生成含清单的 PE 文件，那么可以使用`/addmodule` 开关。为了理解如何生成多文件程序集，假定有两个源代码文件。

- RUT.cs，其中包含不常用类型(R 代表很少)。
- FUT.cs，其中包含常用类型(F 代表经常)。

下面将不常用类型编译到一个单独的模块。这样一来，如果程序集的用户永远不使用不常用类型，就不需要部署这个模块。

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

上述命令行造成 C#编译器创建名为 RUT.netmodule 的文件。这是一个标准的 DLL PE 文件，但是，CLR 不能单独加载它。

接着将常用类型编译到另一个模块。该模块将成为程序集清单的宿主，因为这些类型会经常用到。事实上，由于该模块现在代表整个程序集，所以我将输出文件的名称改为 MultiFileLibrary.dll，而不是默认的 FUT.dll。

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

上述命令行指示 C#编译器编译 FUT.cs 来生成 MultiFileLibrary.dll。由于指定了`/t:library` 开关，所以生成的是含有清单元数据表的 DLL PE 文件。`/addmodule:RUT.netmodule` 开关告诉编译器 RUT.netmodule 文件是程序集的一部分。具体地说，`/addmodule` 开关告诉编译器将文件添加到 FileDef 清单元数据表，并将 RUT.netmodule 的公开导出类型添加到 ExportedTypesDef 清单元数据表。

① 如果使用`/t[arget]:winmdobj`，那么生成的.winmdobj 文件必须传给 WinMDExp.exe 工具进行处理，以便将程序集的公共 CLR 类型作为 Windows Runtime 类型公开。WinMDExp.exe 工具根本不会碰 IL 代码。

编译器最终创建如图 2-1 所示的两个文件。清单在右边的模块中。

RUT.netmodule 文件包含编译 RUT.cs 所生成的 IL 代码。该文件还包含一些定义元数据表，描述了 RUT.cs 定义的类型、方法、字段、属性、事件等。还包含一些引用元数据表，描述了 RUT.cs 引用的类型、方法等。MultiFileLibrary.dll 是一个单独的文件。与 RUT.netmodule 相似，MultiFileLibrary.dll 包含编译 FUT.cs 所生成的 IL 代码以及类似的定义与引用元数据表。然而，MultiFileLibrary.dll 还包含额外的清单元数据表，这使 MultiFileLibrary.dll 成为了程序集。清单元数据表描述了程序集的所有文件(MultiFileLibrary.dll 本身和 RUT.netmodule)。清单元数据表还包含从 MultiFileLibrary.dll 和 RUT.netmodule 导出的所有公共类型。

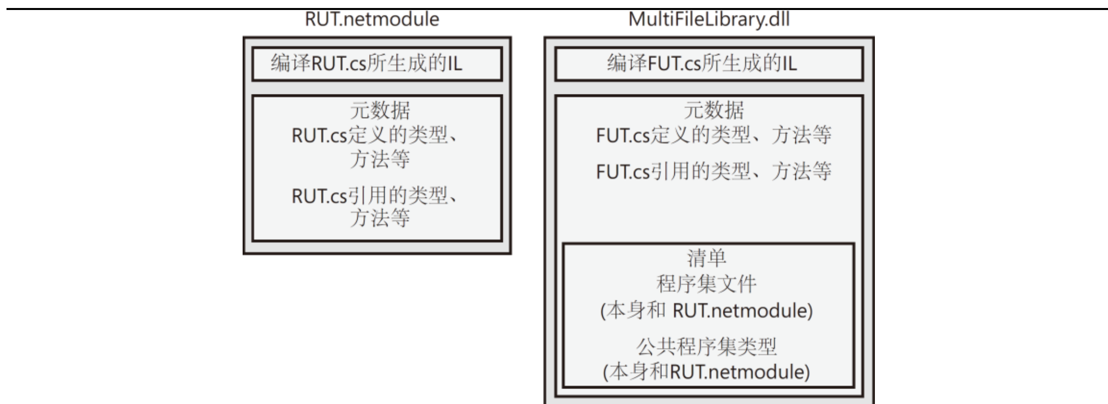


图 2-1 含有两个托管模块的多文件程序集，清单在其中一个模块中

注意：清单元数据表实际并不包含从清单所在的 PE 文件导出的类型。这是一项优化措施，旨在减少 PE 文件中的清单信息量。因此，上述说法“清单元数据表还包含从 MultiFileLibrary.dll 和 RUT.netmodule 导出的所有公共类型”并非百分之百准确。不过，这种说法确实精准地反映了清单在逻辑意义上公开的内容。

生成 MultiFileLibrary.dll 程序集之后，接着可以使用 ILDasm.exe 检查元数据的清单表，验证程序集文件确实包含了对 RUT.netmodule 文件的类型的引用。FileDef 和 ExportedTypesDef 元数据表的内容如下所示。

```
File #1 (26000001)
-------------------------------------------------------
    Token: 0x26000001
    Name : RUT.netmodule
    HashValue Blob : e6 e6 df 62 2c a1 2c 59 97 65 0f 21 44 10 15 96 f2 7e db c2
    Flags : [ContainsMetaData]  (00000000)


ExportedType #1 (27000001)
-------------------------------------------------------
    Token: 0x27000001
    Name: ARarelyUsedType
    Implementation token: 0x26000001
    TypeDef token: 0x02000002
    Flags     : [Public] [AutoLayout] [Class] [Sealed] [AnsiClass]
                [BeforeFieldInit]  (00100101)
```

可以看出，RUT.netmodule 文件已被视为程序集的一部分，它的 token 是 0x26000001。在 ExportedTypesDef 表中可以看到一个公开导出的类型，名为 `ARarelyUsedType`。该类型的

实现 token 是 0x26000001，表明类型的 IL 代码包含在 RUT.netmodule 文件中。

客户端代码必须使用`/r[eference]:MultiFileLibrary.dll` 编译器开关生成，才能使用 MultiFileLibrary.dll 程序集的类型。该开关指示编译器在搜索外部类型时加载 MultiFileLibrary.dll 程序集以及 FileDef 表中列出的所有文件。要求程序集的所有文件都已安装，而且能够访问。删除 RUT.netmodule 文件会导致 C#编译器会报告以下错误：

```
fatal error 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。”
```

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。

注意：以下内容仅供技术宅参考。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。

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。

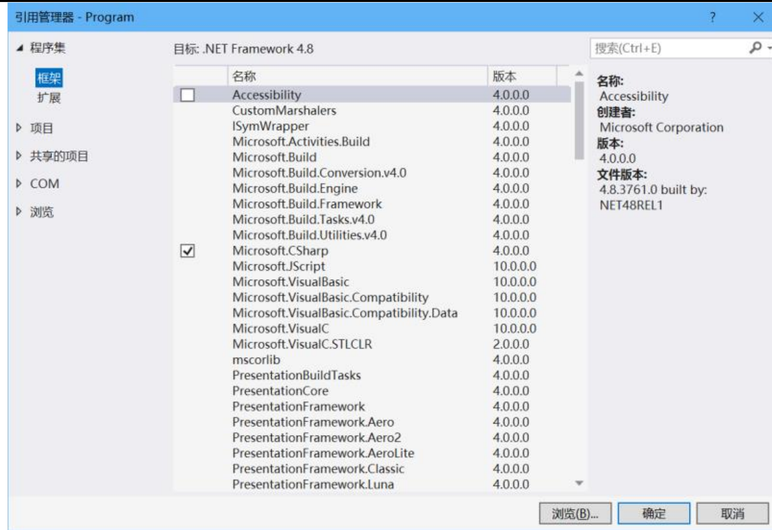


图 2-2 Visual Studio 的“引用管理器”

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“项目”节点，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。利用“COM”节点，可以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“浏览”节点，可以选择最近添加到其他项目的程序集。

按照 *https://tinyurl.com/4kpebea3* 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，程序集链接器就相当有用了。还可以使用 AL.exe 来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。

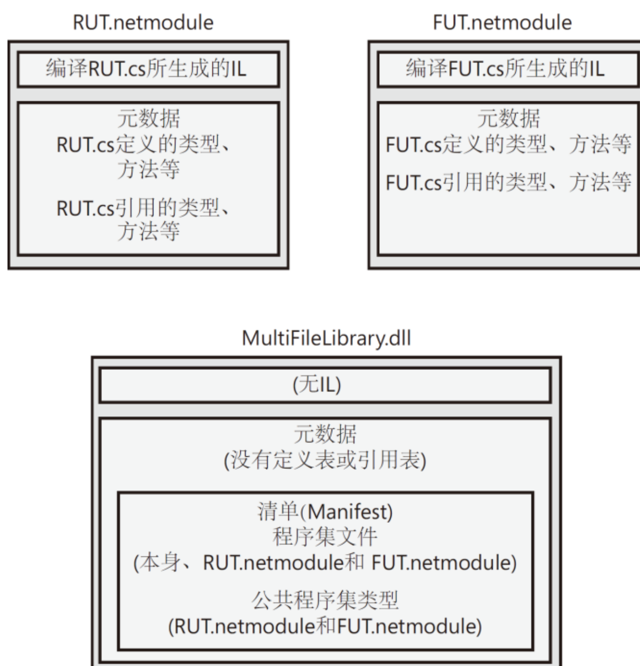


图 2-3 由三个托管模块构成的多文件程序集，其中一个含有清单

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 实用程序还能生成 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，可以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，所以 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，那么何必还要为它创建程序集呢？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。可以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，可以使用/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，最后注意可以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”标签页中添加“资源文件”。嵌入图标的目的一般是在 Windows 文件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#编译器默认会生成这种清单信息，但是可以使用`/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”标签页。

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("")]
```

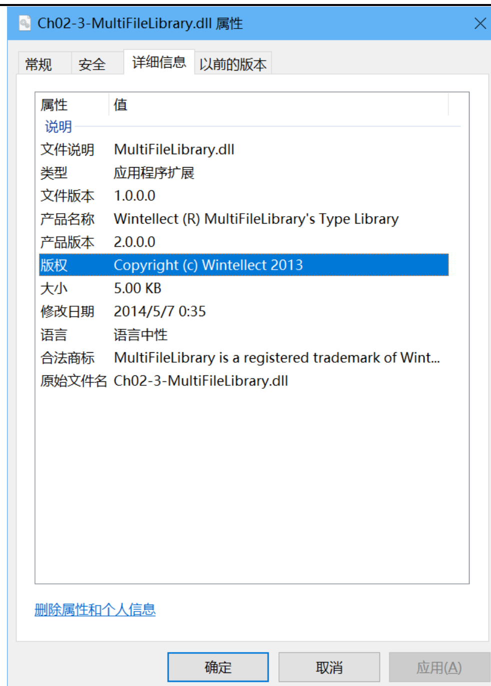


图 2-4 “Ch02-3-MultiFileLibrary.dll 属性”对话框的“详细信息”标签页

重要提示：Windows 文件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，那么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。所以，在这种情况下，最好是用定制特性设置相应的信息。

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
Version	/version	System.Reflection.AssemblyVersionAttribute
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”标签页中单击“程序集信息”。随后会看到如图 2-5 所示的对话框。

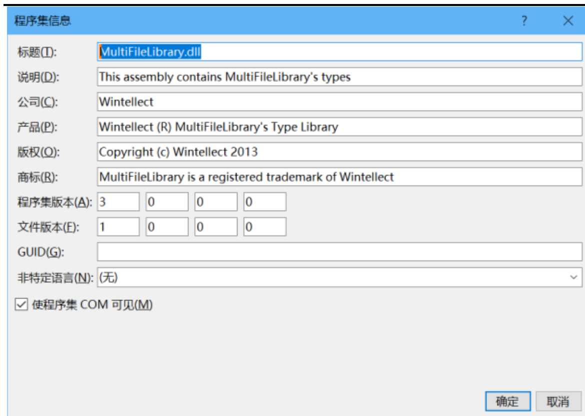


图 2-5 Visual Studio 的“程序集信息”对话框

版本号

上一节指出，可以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 文件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

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

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

① 文档翻译为“区域性”。——译注

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。

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 开关来指定语言文化。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-load①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！

① 不经应用商店而将软件拷贝到设备上，就称为 side-load。——译注

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。

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，也可以使用 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，可以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。可以为 `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。

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.#####)。

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，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。

探测程序集文件

CLR 在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。请访问 *https://tinyurl.com/3ruyrvyc* 了解详情(中文版)。

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。

程序集可以采用两种方式部署：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，两家公司就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：可以利用辅助类 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。弱命名程序集可以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，还用发布者的私钥进行了签名。

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，请像下面这样运行 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):
002400000c80000094000000060200000024000052534131000400000100010 0e3e32597daeb0a
0d7acf5a6230a3f36d0c406571ae39b1feb5ec0bc93088145011dab48253762b44abf97838380c
29592c58af4eeecafe3dfe20f029e5b70a18d58214c40563fd7bec41c090c94931df579c2a6bb8
```

---

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*。

```
019303084444c2945eb488232ff4c43cee91f0977af0e1da2e1dce555803d518c6dfaf2f51c022
775ecf9d

公钥标记为 d4ef0f81895d6eed
```

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#编译器在看到这个开关时，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，可以显示项目属性，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，可以使用 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。

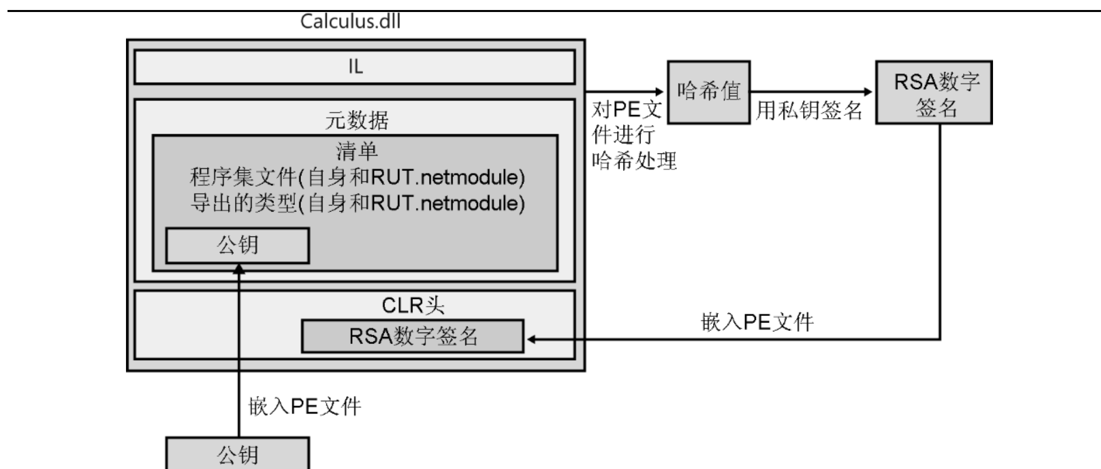


图 3-1 对程序集进行签名

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=4.0.0.0, 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”。

如果检查 DLL 程序集的 AssemblyDef 元数据表，那么会看到以下内容：

```
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 4.0.30319.0
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 的用户没有管理员权限，那么 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，那么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，程序集中还可以引用由 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 章已经介绍了如何使用 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，如果指定的是不含路径的文件名，那么 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，那么可以在执行 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，另一套则安装到 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。对于这些程序集来说，只有它们中的元数据才是最重要的。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，由于代码可以针对特定 CPU 架构进行优化，所以 GAC 允许存在一个程序集的多个拷贝。每种 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。如果能找到被引用程序集，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产生额外的性能开销。

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。密钥可以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。这项措施能防止私钥泄露，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。如果使用 C#编译器，就指定`/delaysign`编译器开关。如果使用 Visual Studio，就打开项目属性页，在“签名”标签页中勾选“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`开关会将程序集的标识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。(如果安装的是.NET Framework 4.8，将上述路径中的 8.0 改为 10.0，将 4.0 改成 4.8。)

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，并且可以随意测试程序集。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后，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
   ```

本节开头说过，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一个磁盘文件中。“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，就会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，那么必须为 CSC.exe、AL.exe 和 SN.exe 程序指定不同的开关。编译时(CSC.exe)要指定/keycontainer 开关而不是/keyfile 开关；链接时(AL.exe)要指定/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，要指定-Rc 开关而不是-R 开关。SN.exe 还提供了其他几个开关让开发人员与 CSP 交互。

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 可以引用用户机器上的任何目录，也可以引用 Web 地址。如果引用 Web 地址，那么 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，那么 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 token① 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。

- **同一文件**

① 元数据 token 的详情请参见 2.4 节。——译注

编译时便能发现对同一文件中的类型的访问，这称为**早期绑定**(early binding)[①]。类型直接从文件中加载，然后继续执行。

- **不同文件，相同程序集**

  “运行时”确认被引用的文件在当前程序集的清单的 FileDef 表中。然后，“运行时”检查程序集清单文件的加载目录。文件被加载，检查哈希值以确保文件完整性。发现类型的成员，然后继续执行。

- **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，然后继续执行。

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`，`ReflectionOnlyAssemblyResolve` 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：从严格意义上说，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

---

① 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

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”代码栈(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。

IL 引用一个类型

IL 引用一个成员

AssemblyRef：
类型在不同程序
集的不同文件中

TypeRef 记录项
指向什么？

AssemblyRef 记录
项指向什么？

强命名程序集

先在 GAC 中搜
索程序集，再在
AppBase 中搜索

ModuleRef：
类型在同一个
程序集的另一
个文件中

弱命名
程序集

检查 ModuleRef
表并加载恰当
的文件

ModuleDef：
类型在同一个
程序集的同一
个文件中

在 AppBase 中搜
索程序集

加载含有清单
的文件

清单文件中的类型

ExportedTypesDef
记录项指向什么？

不在清单文件
中的类型

创建内部类型结构

加载文件

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用于定义类型的新程序集。

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用于定义类型的程序集。

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。

这个发布者策略程序集一旦生成，就可以随同新的 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 比它修复的 bug 还要多，那么管理员可以指示 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"/>
```

该元素可以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，不要随带一个发布者策略程序集。另外，也不必测试这类程序集的向后兼容性。

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 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
- 在运行时的相互关系

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。还要讨论类型安全性、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，那么类型应重写该方法。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”
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 理论上应察觉到与调用线程关联的 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 对象可与各种反射类配合，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(overriding)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，并设置新对象的实例字段，使它们与 this 对象的实例字段完全一致。返回对新实例的引用[②]
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)

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

② 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，而静态成员属于类。——译注

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，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。额外成员的字节数需计入对象大小。

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)。

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一种类型。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 极其重要的一个特性。

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。这样修改后，本例在编译时就能报错，而非等到运行时才报错。这里之所以使用 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.
```

如果对象引用为 null，那么 is 操作符总是返回 false，因为没有可供检查其类型的对象。is 操作符一般像下面这样使用：

```
if (o is Employee) {
    Employee e = (Employee) o;
    // 在 if 语句剩余的部分使用 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。否则，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();         // 新建一个 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成功编译和执行(OK)，造成编译时错误(CTE)，还是造成运行时错误(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，开发人员可以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一系列类型，而 `System.IO` 命名空间定义了执行 I/O 操作的一系列类型。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# 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# 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 自己其实对“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，保证实参具有正确的类型，正确地使用方法返回值，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 编译器开关来实现。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(比如 Object，Int32，String 等)的定义。

你可能已经猜到，编译器对待命名空间的方式存在潜在的问题：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。但有的时候，非不为也，是不能也。“运行时”鼓励组件重用。例如，应用程序可能同时使用了 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，所以应该在引用时用完全限定名称区分它们。也就是说，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#编译器将报告错误消息：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。[①]

在自己库中设计要由第三方使用的类型时，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。在查阅文档时，可以清楚地看到 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(用于实现类型的文件)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。

[①] https://tinyurl.com/mr2xvbba

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，`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。在编译引用了 `ResXFileRef` 类型的代码时，需要在源代码中添加 `using System.Resources;` 指令，而且要使用 `/r:System.Windows.Forms.dll` 编译器开关。[①]

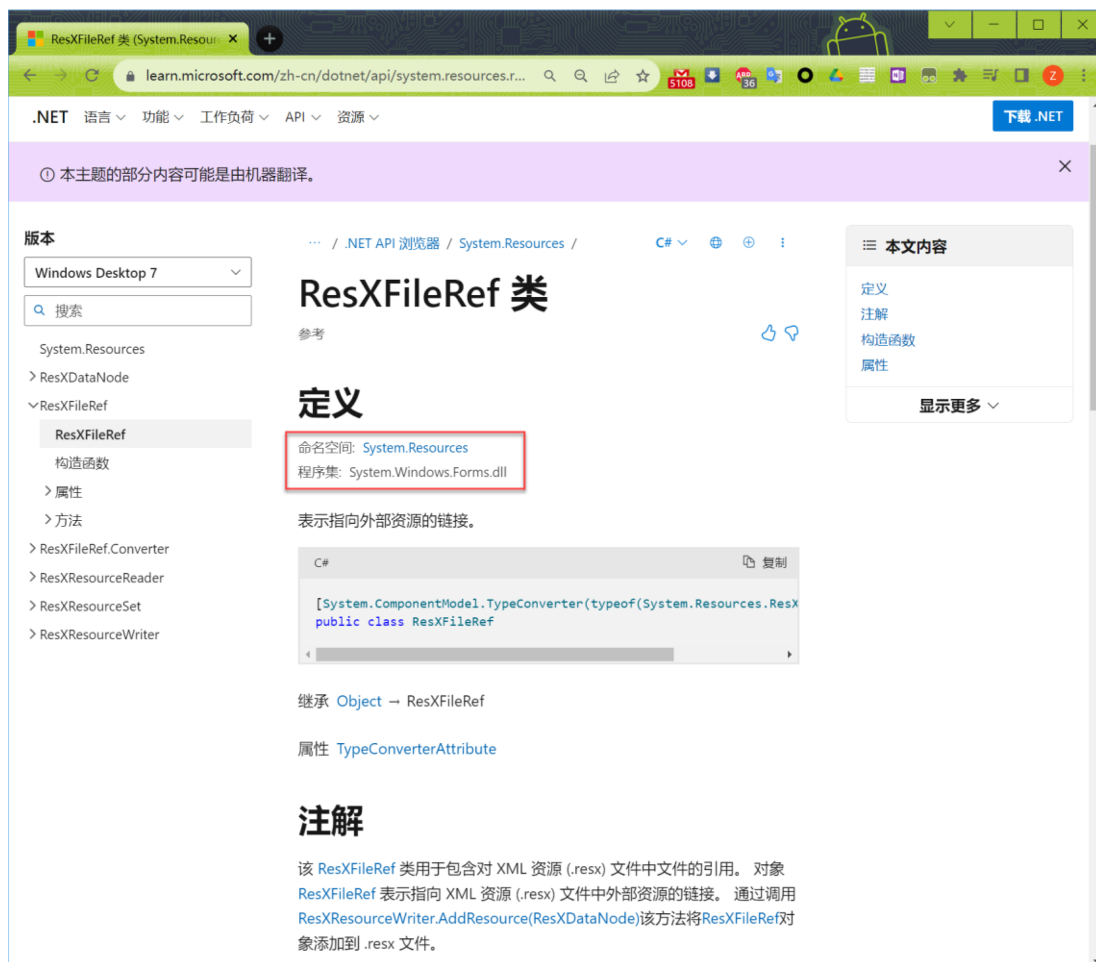


图 4-1 文档显示了类型的命名空间和程序集信息

4.4 在运行时的相互关系

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

[①] 本例网址是 https://tinyurl.com/ys3n7mbz ——译注

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(stack)。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。

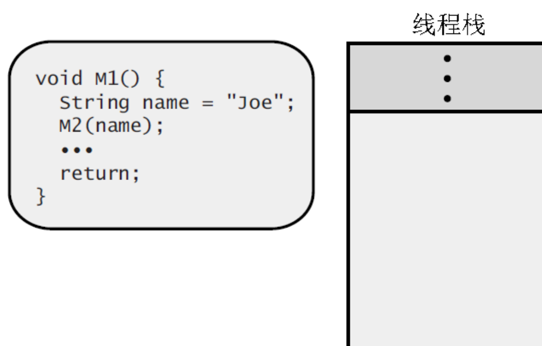


图 4-2 一个线程的栈，正准备调用 M1 方法

除了最简单的方法之外，大多数代码都包含一些“序幕”(prologue)代码，用于初始化方法，然后方法才能开始“干活儿”；还包含一些“尾声”(epilogue)代码，在方法“干完活儿”后对其进行清理，然后方法才能返回至调用者。M1 方法开始执行时，其“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。

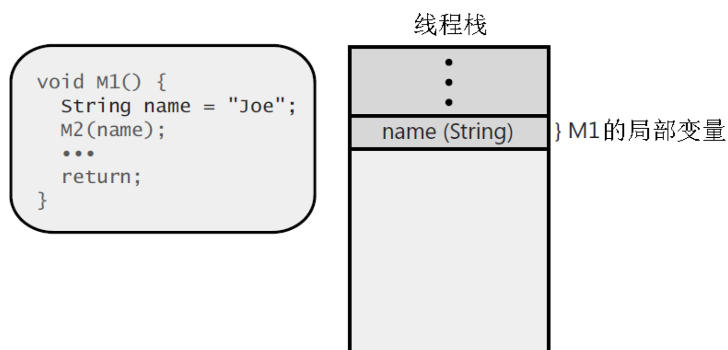


图 4-3 在线程栈上分配 M1 的局部变量

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)。

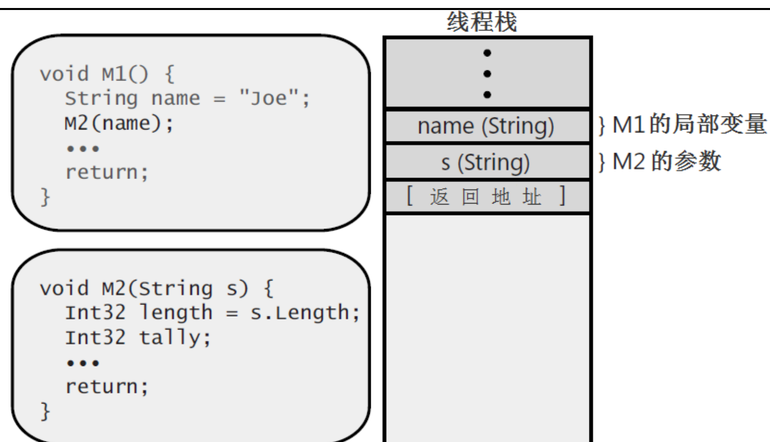


图 4-4 M1 调用 M2 时，将实参和返回地址压入线程栈

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。

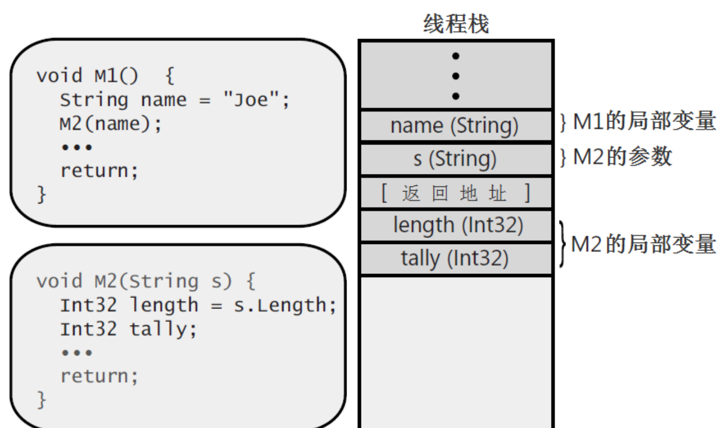


图 4-5 在线程栈上分配 M2 的局部变量

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

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

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，而且已经创建了一个线程(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。

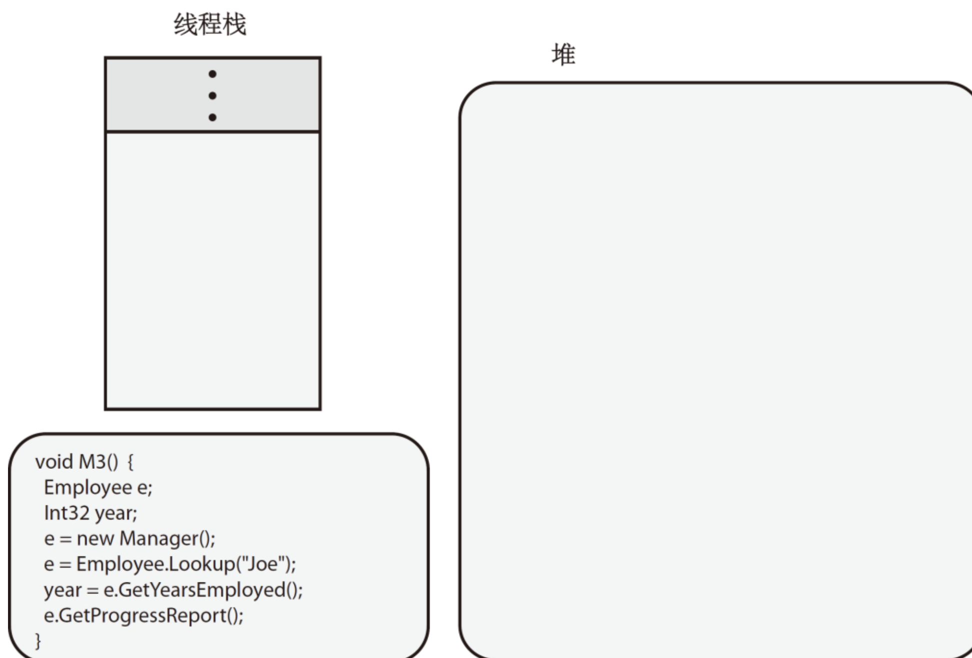


图 4-6 CLR 已加载到进程中，堆已初始化，线程栈已创建，正要调用 M3 方法

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。

让我们稍微花点时间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，可以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 类型定义了三个方法(GetYearsEmployed，GetProgressReport 和 Lookup)，所以 Employee 的方法表有三个记录项。Manager 类型只定义了一个方法(GetProgressReport 的重写版本)，所以 Manager 的方法表只有一个记录项。

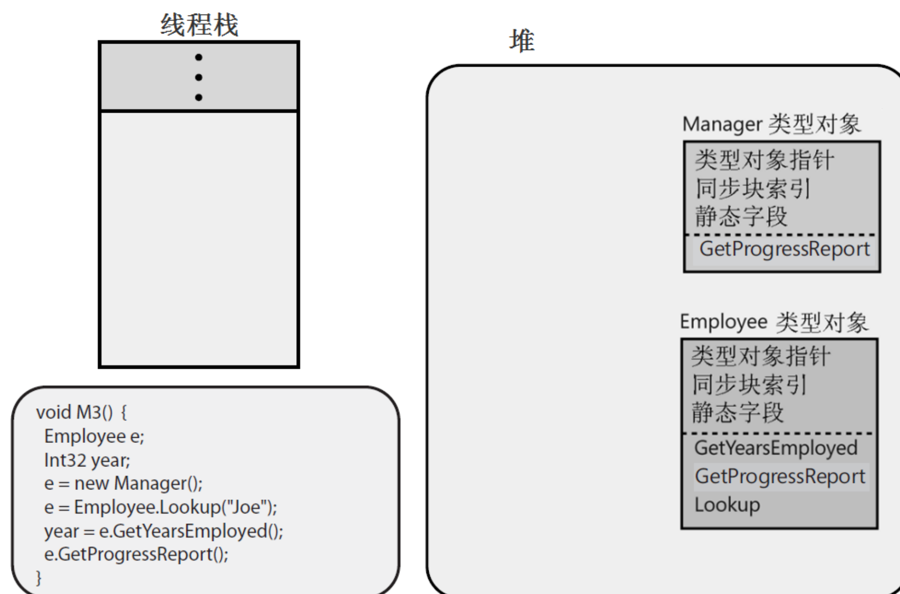


图 4-7 Employee 和 Manager 类型对象在 M3 被调用时创建

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 执行代码来构造一个 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，另外还要容纳由 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”成员来引用与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)。

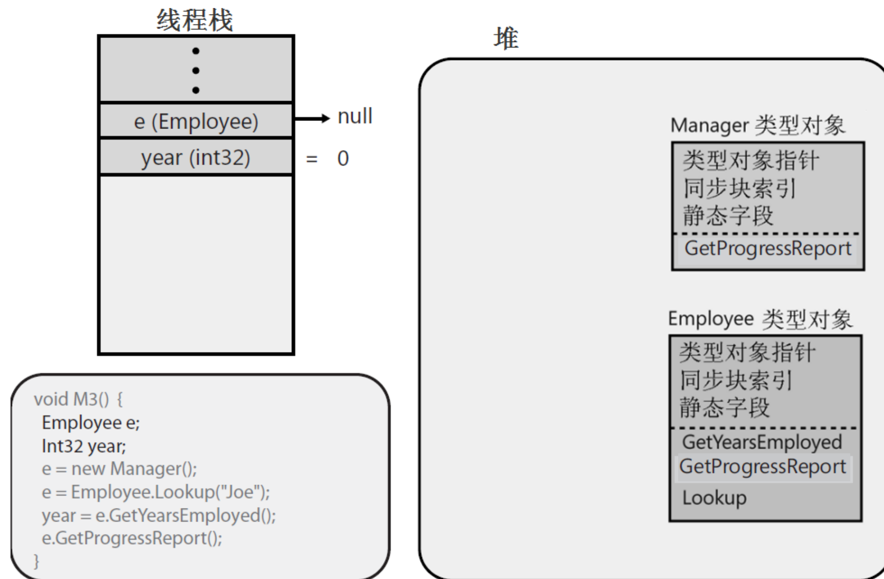


图 4-8 在线程栈上分配 M3 的局部变量

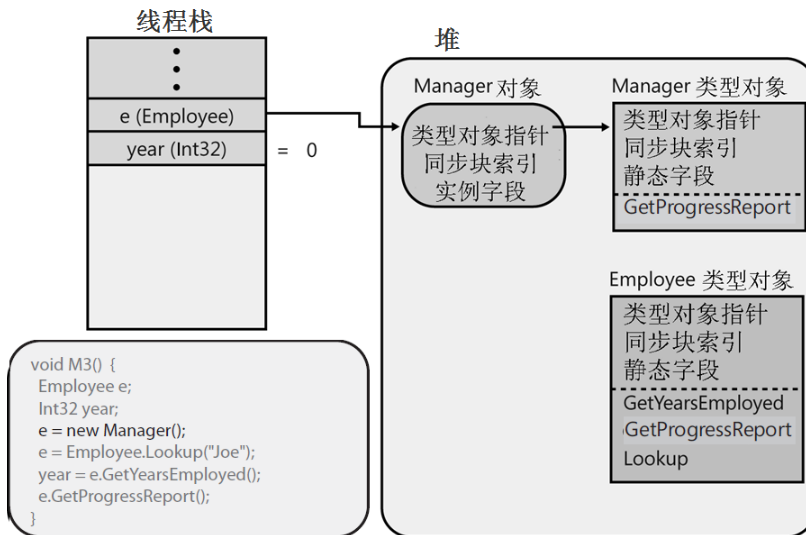


图 4-9 分配并初始化 Manager 对象

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。

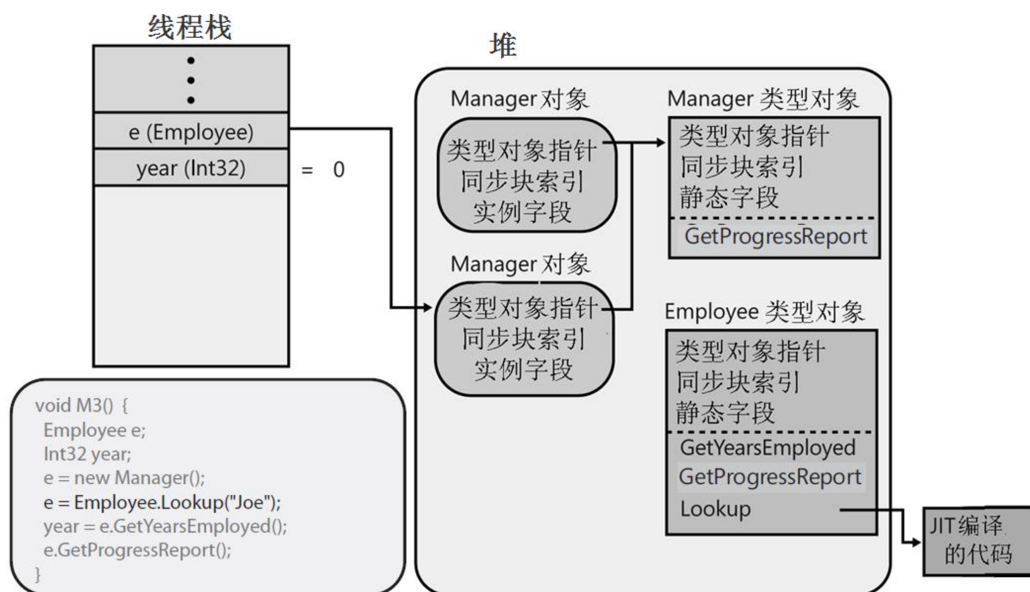


图 4-10 Employee 的静态方法 Lookup 为 Joe 分配并初始化 Manager 对象

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。

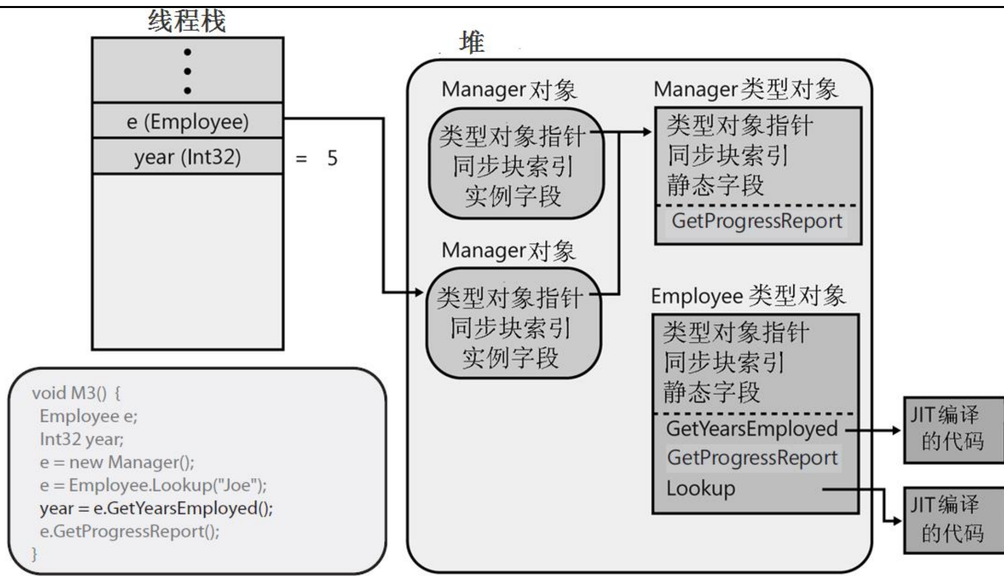


图 4-11 **Employee 的非虚实例方法 GetYearsEmployed 调用后返回 5**

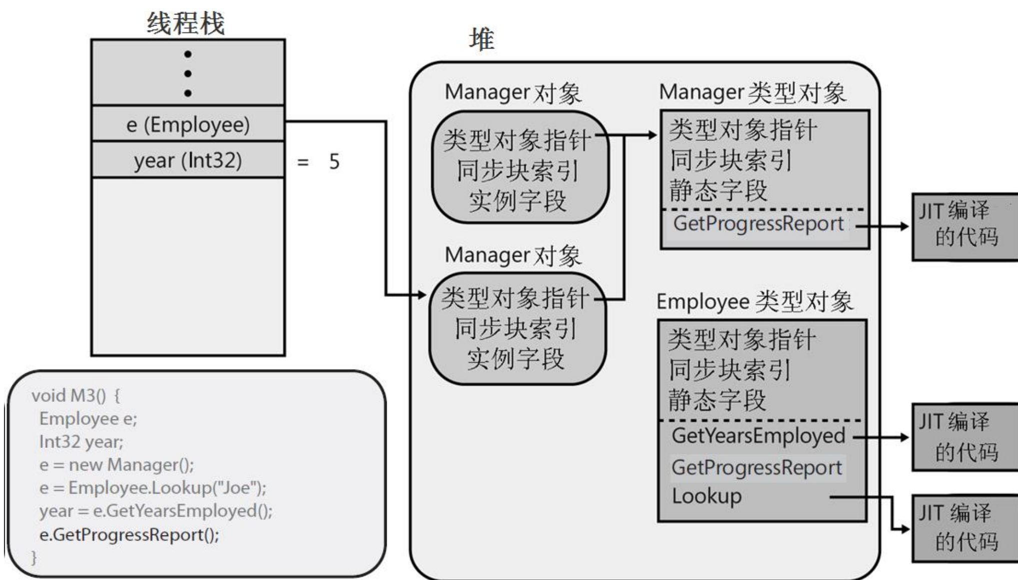


图 4-12 调用 Employee 的虚实例方法 GetProgressReport，最终执行 Manager 重写的版本

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。

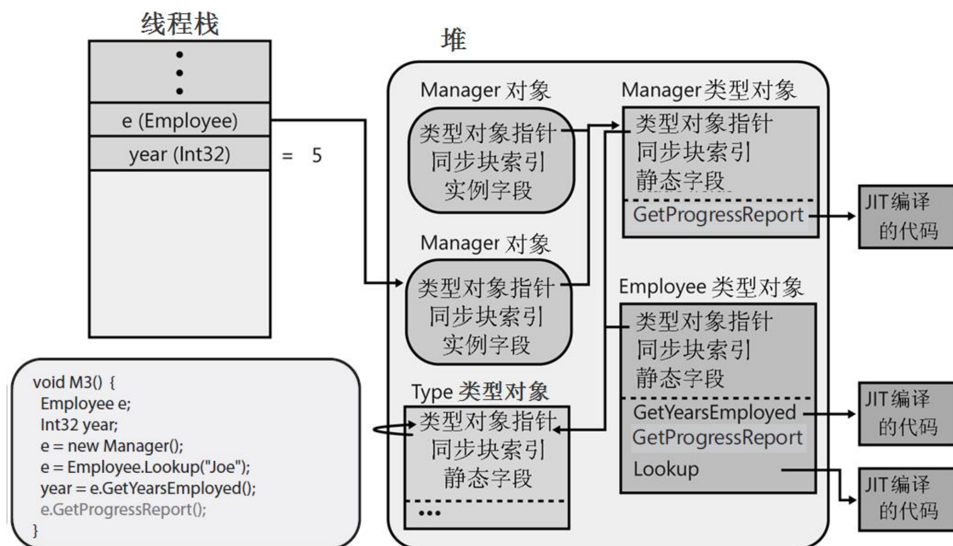


图 4-13 **Employee 和 Manager 类型对象是 System.Type 类型的实例**

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样的语法来声明并初始化整数，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：

```
int                 a = 0;                  // 最方便的语法
```

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

```
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(dynamic dispatch)。详情参见本章最后的 5.5 节“dynamic 基元类型”

从另一个角度，可以认为 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 几乎完全是用 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
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(*https://tinyurl.com/3s8p8vkj*)。

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，所以可以像下面这样为实例(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，那么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：

---

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

```
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。

溢出大多数时候非我们所愿。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 会抛出 OverflowException 异常。

除了全局性地打开或关闭溢出检查，程序员还可以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
```

[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

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

事实上，如果使用了 checked 语句块，就可以将+=操作符用于 Byte，从而稍微简化一下代码：

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

重要提示：由于 checked 操作符和 checked 语句唯一的作用就是决定生成哪个版本的加、减、乘和数据转换 IL 指令，所以在 checked 操作符或语句中调用方法，不会对该方法造成任何影响，如下例所示：

```
checked {
   // 假定 SomeMethod 试图把 400 加载到一个 Byte 中。
   SomeMethod(400);
   // SomeMethod 可能会、也可能不会抛出 OverflowException 异常。
   // 如果 SomeMethod 使用 checked 指令编译，就可能会抛出异常，
   // 但这和当前的 checked 语句无关。
}
```

根据我的经验，许多计算都会产生令人吃惊的结果。这一般是由于无效的用户输入，但也可能是由于系统的某个部分返回了程序员没有预料到的值。所以我对程序员有以下建议。

1. 尽量使用有符号数值类型(比如 Int32 和 Int64)而不是无符号数值类型(比如 UInt32 和 UInt64)。这允许编译器检测更多的上溢/下溢错误。除此之外，类库多个部分(比如 Array 和 String 的 Length 属性)被硬编码为返回有符号的值。这样在代码中四处移动这些值时，需要进行的强制类型转换就少了。较少的强制类型转换使代码更整洁，更容易维护。除此之外，无符号数值类型不符合 CLS。

2. 写代码时，如果代码可能发生你不希望的溢出(可能是因为无效的输入，比如要求使用最终用户或客户机提供的数据)，就把这些代码放到 checked 块中。同时捕捉 OverflowException，得体地从错误中恢复。

3. 写代码时，将允许发生溢出的代码显式放到 unchecked 块中，比如在计算校验和时。

4. 如果代码没有使用 checked 或 unchecked，那么相当于你希望在发生溢出时抛出一个异常。例如，在输入已知的前提下计算一些东西(比如质数)时，发生的溢出应被视为 bug。

开发应用程序时，打开编译器的/checked+开关进行调试性生成。这样系统会对没有显式标记 checked 或 unchecked 的代码进行溢出检查，所以应用程序运行起来会慢一些。此时

一旦发生异常，就可以轻松检测到，而且能及时修正代码中的 bug。但是，为了正式发布而生成应用程序时，应使用编译器的`/checked-`开关，确保代码能更快运行，不会产生溢出异常。要在 Microsoft Visual Studio 中更改 Checked 设置，请打开项目的属性页，点击“生成”标签，单击“高级”，再勾选“检查算术溢出”，如图 5-1 所示。

图 5-1 在 Visual Studio 的“高级生成设置”对话框中指定编译器是否检查溢出

如果应用程序能容忍总是执行 checked 运算而带来的轻微性能损失，建议即使是为了发布而生成应用程序，也用/checked 命令行开关进行编译，这样可防止应用程序在包含已损坏的数据(甚至可能是安全漏洞)的前提下继续运行。例如，通过乘法运算来计算数组索引时，相较于因为数学运算的“回滚”[①]而访问到不正确的数组元素，抛出 OverflowException 异常才是更好的做法。

重要提示：System.Decimal 是非常特殊的类型。虽然许多编程语言(包括 C#和 Visual Basic)将 Decimal 视为基元类型，但 CLR 不然。这意味着 CLR 没有知道如何处理 Decimal 值的 IL 指令。在文档中查看 Decimal 类型，可以看到它提供了一系列 public static 方法，包括 Add，Subtract，Multiply，Divide 等。此外，Decimal 类型还为+，-，*，/等提供了操作符重载方法。

编译使用了 Decimal 值的程序时，编译器会生成代码来调用 Decimal 的成员，并通过这些成员来执行实际运算。这意味着 Decimal 值的处理速度慢于 CLR 基元类型的值。另外，由于没有相应的 IL 指令来处理 Decimal 值，所以 checked 和 unchecked 操作符、语句以及编译器开关都失去了作用。对 Decimal 值执行不安全的运算时，肯定会抛出 OverflowException 异常。

类似地，System.Numerics.BigInteger 类型也很特殊，因为它在内部使用 UInt32 数组来表示任意大的整数，它的值没有上限和下限。因此，对 BigInteger 执行的运算永远不会造成 OverflowException 异常。但如果值太大，没有足够多的内存来改变数组大小，那么对 BigInteger 的运算可能抛出 OutOfMemoryException 异常。

① 乘法运算可能产生一个较大的值，超出数组的索引范围。参见上一条关于“wrap”的注释。——译注

5.2 引用类型和值类型

CLR 支持两种类型：**引用类型**和**值类型**。虽然 FCL 的大多数类型都是引用类型，但程序员用得最多的还是值类型。引用类型总是从托管堆分配，C#的 `new` 操作符返回对象内存地址——即指向对象数据的内存地址。使用引用类型是，必须留意性能问题。首先要认清楚以下四个事实。

1. 内存必须从托管堆分配。
2. 堆上分配的每个对象都有一些额外成员，这些成员必须初始化。
3. 对象中的其他字节(为字段而设)总是设为零。
4. 从托管堆分配对象时，可能强制执行一次垃圾回收。

如果所有类型都是引用类型，应用程序的性能将显著下降。设想每次使用 `Int32` 值时都进行一次内存分配，性能会受到多么大的影响！为了提升简单和常用的类型的性能，CLR 提供了名为“值类型”的轻量级类型。值类型的实例一般在线程栈上分配(虽然也可作为字段嵌入引用类型的对象中)。在代表值类型实例的变量中不包含指向实例的指针。相反，变量中包含了实例本身的字段。由于变量已包含了实例的字段，所以操作实例中的字段不需要提领(dereference)指针。值类型的实例不受垃圾回收器的控制。因此，值类型的使用缓解了托管堆的压力，并减少了应用程序生存期内的垃圾回收次数。

文档清楚指出哪些类型是引用类型，哪些是值类型。在文档中查看类型时，任何称为“类”的类型都是引用类型。例如，`System.Exception` 类、`System.IO.FileStream` 类以及 `System.Random` 类都是引用类型。相反，所有值类型都称为结构或枚举。例如，`System.Int32` 结构、`System.Boolean` 结构、`System.Decimal` 结构、`System.TimeSpan` 结构、`System.DayOfWeek` 枚举、`System.IO.FileAttributes` 枚举以及 `System.Drawing.FontStyle` 枚举都是值类型。

进一步研究文档，会发现所有结构都是抽象类型 `System.ValueType` 的直接派生类。`System.ValueType` 本身又直接从 `System.Object` 派生。根据定义，所有值类型都必须从 `System.ValueType` 派生。所有枚举都从 `System.Enum` 抽象类型派生，后者又从 `System.ValueType` 派生。CLR 和所有编程语言都给予枚举特殊待遇[①]。欲知枚举类型的详情，请参见第 15 章“枚举类型和位标志”。

虽然不能在定义值类型时为它选择基类型，但如果愿意，值类型可实现一个或多个接口。除此之外，所有值类型都隐式密封，目的是防止将值类型用作其他引用类型或值类型的基类型。例如，无法将 `Boolean`，`Char`，`Int32`，`UInt64`，`Single`，`Double`，`Decimal` 等作为

---

① 将其视为“一等公民”，直接支持各种强大的操作。在非托管环境中，枚举就没这么“好命”了。——译注

基类型来定义任何新类型。

重要提示：对于许多开发人员(比如非托管 C/C++开发人员)，最初接触引用类型和值类型时都觉得有些不解。在非托管 C/C++中声明类型后，使用该类型的代码会决定是在线程栈上还是在应用程序的堆中分配类型的实例。但在托管代码中，要由定义类型的开发人员决定在什么地方分配类型的实例，使用类型的人对此并无控制权。

以下代码和图 5-2 演示了引用类型和值类型的区别：

```
// 引用类型(因为'class')
class SomeRef { public Int32 x; }

// 值类型(因为'struct')
struct SomeVal { public Int32 x; }

static void ValueTypeDemo() {
    SomeRef r1 = new SomeRef();       // 在堆上分配
    SomeVal v1 = new SomeVal();       // 在栈上分配
    r1.x = 5;                         // 提领指针
    v1.x = 5;                         // 在栈上修改
    Console.WriteLine(r1.x);          // 显示"5"
    Console.WriteLine(v1.x);          // 同样显示"5"
    // 图 5-2 的左半部分反映了执行以上代码之后的情况

    SomeRef r2 = r1;                  // 只复制引用(指针)
    SomeVal v2 = v1;                  // 在栈上分配并复制成员
    r1.x = 8;                         // r1.x 和 r2.x 都会更改
    v1.x = 9;                         // v1.x 会更改，v2.x 不变
    Console.WriteLine(r1.x);          // 显示"8"
    Console.WriteLine(r2.x);          // 显示"8"
    Console.WriteLine(v1.x);          // 显示"9"
    Console.WriteLine(v2.x);          // 显示"5"
    // 图 5-2 的右半部分反映了执行以上所有代码之后的情况
}
```

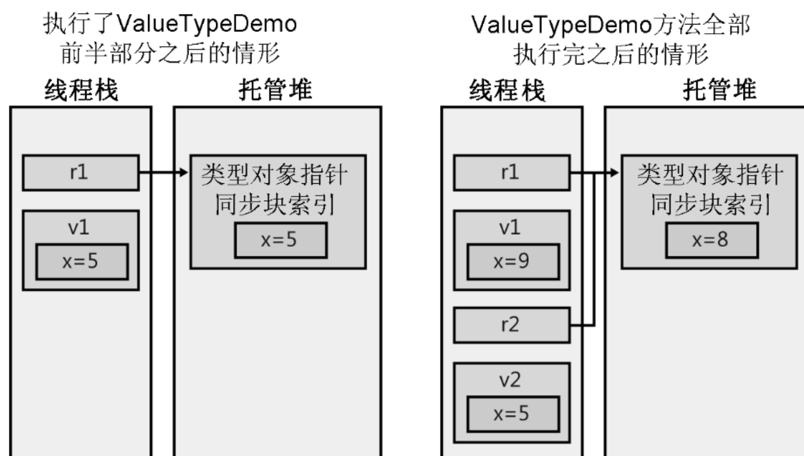


图 5-2 图解代码执行时的内存分配情况

在上述代码中，SomeVal 类型用 struct 声明，而不是用更常见的 class。在 C#中，用 struct 声明的类型是值类型，用 class 声明的类型是引用类型。可以看出，引用类型和值类型的区别相当大。在代码中使用类型时，必须注意是引用类型还是值类型，因为这会极大地影响在代码中表达自己意图的方式。

上述代码中有这样一行：

```
SomeVal v1 = new SomeVal();      // 在栈上分配
```

只看这行代码的写法，似乎是要在托管堆上分配一个 SomeVal 实例。但是，C#编译器知道 SomeVal 是值类型，所以会生成正确的 IL 代码，在线程栈上分配一个 SomeVal 实例。C#还会确保值类型中的所有字段都初始化为零。

上述代码还可以像下面这样写：

```
SomeVal v1;   // 在栈上分配
```

这一行生成的 IL 代码也会在线程栈上分配实例，并将字段初始化为零。唯一的区别在于，如果使用 new 操作符，C#会认为实例已初始化。以下代码更清楚地进行了说明：

```
// 这两行代码能通过编译，因为 C#认为
// v1 的字段已初始化为 0
SomeVal v1 = new SomeVal();
Int32 a = v1.x;

// 这两行代码不能通过编译，因为 C#不认为
// v1 的字段已初始化为 0
SomeVal v1;
Int32 a = v1.x; // error CS0170：使用了可能未赋值的字段"x"
```

设计自己的类型时，要仔细考虑类型是否应该定义成值类型而不是引用类型。值类型有时能提供更好的性能。具体地说，除非满足以下全部条件，否则不应将类型声明为值类型。

- 类型具有基元类型的行为。也就是说，是十分简单的类型，没有成员会修改类型的任何实例字段。如果类型没有提供会更改其字段的成员，就说该类型是**不可变**(immutable)类型。事实上，对于许多值类型，我们都建议将全部字段标记为 readonly(详情参见第 7 章“常量和字段”)。
- 类型不需要从其他任何类型继承。
- 类型也不派生出其他任何类型。

类型实例大小也应在考虑之列，因为实参默认以传值方式传递，造成对值类型实例中的字段进行复制，对性能造成损害。同样地，被定义为返回一个值类型的方法在返回时，实例中的字段会复制到调用者分配的内存中，对性能造成损害。所以，要将类型声明为值类型，

除了要满足以上全部条件，还必须满足以下任意条件。

- 类型的实例较小(16 字节或更小)。
- 类型的实例较大(大于 16 字节)，但不作为方法实参传递，也不从方法返回。

值类型的主要优势是不作为对象在托管堆上分配。当然，与引用类型相比，值类型也存在自身的一些局限。下面列出了值类型和引用类型的一些区别。

- 值类型对象有两种表示形式：**未装箱**和**已装箱**，详情参见下一节。相反，引用类型总是处于已装箱形式。
- 值类型从 `System.ValueType` 派生。该类型提供了与 `System.Object` 相同的方法。但 `System.ValueType` 重写了 `Equals` 方法，能在两个对象的字段值完全匹配的前提下返回 `true`。此外，`System.ValueType` 重写了 `GetHashCode` 方法。生成哈希码时，这个重写方法所用的算法会将对象的实例字段中的值考虑在内。由于这个默认实现存在性能问题，所以定义自己的值类型时应重写 `Equals` 和 `GetHashCode` 方法，并提供它们的显式实现。本章末尾会讨论 `Equals` 和 `GetHashCode` 方法。
- 由于不能将值类型作为基类型来定义新的值类型或者新的引用类型，所以不应在值类型中引入任何新的虚方法。所有方法都不能是抽象的，所有方法都隐式密封(不可重写)。
- 引用类型的变量包含堆中对象的地址。引用类型的变量创建时默认初始化为`null`，表明当前不指向有效对象。试图使用 `null` 引用类型的变量会抛出 `NullReferenceException` 异常。相反，值类型的变量总是包含其基础类型的一个值，而且值类型的所有成员都初始化为 `0`。值类型变量不是指针，访问值类型不可能抛出 `NullReferenceException` 异常。CLR 确实允许为值类型添加“可空”(nullability)标识。可空类型将在第 19 章“可空值类型”详细讨论。
- 将值类型变量赋给另一个值类型变量，会执行逐字段的复制。将引用类型的变量赋给另一个引用类型的变量，只复制内存地址。
- 基于上一条，两个或多个引用类型变量能引用堆中同一个对象，所以对一个变量执行的操作可能影响到另一个变量引用的对象。相反，值类型变量自成一体，对值类型变量执行的操作不可能影响另一个值类型变量。
- 由于未装箱的值类型不在堆上分配，一旦定义了该类型的一个实例的方法不再活动，为它们分配的存储就会被释放，而不是等着进行垃圾回收。[①]

[①] 这段话的意思是，因为(未装箱)值类型实例是在栈上分配的，所以它们的内存分配和释放直接与方法的活动状态相关联。一旦包含值类型实例的方法不再活动(即方法执行完毕)，与这些值类型实例相关的内存就会立即被释放，而不需要等待垃圾回收。这正是“局部变量”的概念。——译注

CLR 如何控制类型中的字段布局

为了提高性能，CLR 能按照它所选择的任何方式排列类型的字段。例如，CLR 可以在内存中重新安排字段的顺序，将对象引用分为一组，同时正确排列和填充数据字段。但在定义类型时，针对类型的各个字段，你可以告诉 CLR 是严格按照自己指定的顺序排列，还是按照 CLR 自己认为合适的方式重新排列。

为了告诉 CLR 应该怎样做，要为自己定义的类或结构应用 `System.Runtime.InteropServices.StructLayoutAttribute` 特性。可向该特性的构造器传递 `LayoutKind.Auto`，让 CLR 自动排列字段；也可传递 `LayoutKind.Sequential`，让 CLR 保持你的字段布局；也可传递 `LayoutKind.Explicit`，利用偏移量在内存中显式排列字段。如果不为自己定义的类型显式指定 `StructLayoutAttribute`，编译器会选择它自认为最好的布局。

注意，Microsoft C#编译器默认为引用类型(类)选择 `LayoutKind.Auto`，为值类型(结构)选择 `LayoutKind.Sequential`。显然，C#编译器团队认为，和非托管代码互操作时会经常用到结构。为此，字段必须保持程序员定义的顺序。然而，假如创建的值类型不与非托管代码互操作，就应考虑覆盖 C#编译器的默认设定。下面是一个例子：

```
using System;
using System.Runtime.InteropServices;

// 让 CLR 自动排列字段以增强这个值类型的性能
[StructLayout(LayoutKind.Auto)]
internal struct SomeValType {
   private readonly Byte m_b;
   private readonly Int16 m_x;
   ...
}
```

`StructLayoutAttribute` 还允许显式指定每个字段的偏移量，这要求向其构造器传递 `LayoutKind.Explicit`。然后向值类型中的每个字段都应用 `System.Runtime.InteropServices.FieldOffsetAttribute` 特性的实例，向该特性的构造器传递 `Int32` 值来指出字段第一个字节距离实例起始处的偏移量(以字节为单位)。显式布

局常用于模拟非托管 C/C++中的 union[①]，因为多个字段可起始于内存的相同偏移位置。下面是一个例子：

```
using System;
using System.Runtime.InteropServices;

// 开发人员显式排列这个值类型的字段
[StructLayout(LayoutKind.Explicit)]
internal struct SomeValType {
    [FieldOffset(0)]
    private readonly Byte m_b;//m_b 和 m_x 字段在该类型的实例中相互重叠

    [FieldOffset(0)]
    private readonly Int16 m_x;//m_b 和 m_x 字段在该类型的实例中相互重叠
}
```

注意，在类型中，一个引用类型和一个值类型相互重叠是不合法的。虽然允许多个引用类型在同一个起始偏移位置相互重叠，但这无法验证(unverifiable)。定义类型，在其中让多个值类型相互重叠则是合法的。但是，为了使这样的类型能够验证(verifiable)，所有重叠字节都必须能通过公共字段访问。

5.3 值类型的装箱和拆箱

值类型比引用类型“轻”，原因是它们不作为对象在托管堆中分配，不被垃圾回收，也不通过指针进行引用。但许多时候都需要获取对值类型实例的引用。例如，假定要创建一个 ArrayList(在 System.Collections 命名空间中定义的一个类型)对象来容纳一组 Point 结构，代码如下：

```
// 声明值类型
struct Point {
    public Int32 x, y;
}
```

[①]在 C 和 C++中，union(联合体或共同体)是一种特殊的类，union 中的数据成员在内存中的存储相互重叠。每个数据成员都从相同内存地址开始。分配给 union 的存储区数量是包含它最大数据成员所需的内存数。同一时刻只有一个成员可以被赋值。作为一个内存区域，union 能随着时间推移包含各种类型的对象。由于 union 的成员共享相同的存储空间，所以 union 一次最多只能包含一个对象，并需要足够的内存来容纳其最大的成员。通常将 C++ std::variant 对象称为类型安全的 union。摘自《学习 C++20》，清华大学出版社，2023 年。——译注

```
public sealed class Program {
    public static void Main() {
        ArrayList a = new ArrayList();
        Point p;                // 分配一个 Point(不在堆中分配)
        for (Int32 i = 0; i < 10; i++) {
            p.x = p.y = i;      // 初始化值类型中的成员
            a.Add(p);           // 对值类型装箱，将引用添加到 Arraylist 中
        }
        ...
    }
}
```

每次循环迭代都会初始化一个 Point 的值类型字段，并将该 Point 存储到 ArrayList 中。但思考一下 ArrayList 中究竟存储了什么？是 Point 结构，Point 结构的地址，还是其他完全不同的东西？要知道正确答案，必须研究 ArrayList 的 Add 方法，了解它的参数被定义成什么类型。本例的 Add 方法原型如下：

```
public virtual Int32 Add(Object value);
```

可以看出，Add 获取的是一个 Object 参数。也就是说，Add 获取对托管堆上的一个对象的引用(或指针)来作为参数。但之前的代码传递的是 p，也就是一个 Point，是值类型。为了使代码正确工作，Point 值类型必须转换成真正的、在堆中托管的对象，而且必须获取对该对象的引用。

将值类型转换成引用类型要使用**装箱**机制。下面总结了对值类型的实例进行装箱时发生的事情。

1. 在托管堆中分配内存。分配的内存量是值类型各字段所需的内存量，还要加上托管堆所有对象都有的两个额外成员(类型对象指针和同步块索引)所需的内存量。

2. 值类型的字段复制到新分配的堆内存。

3. 返回对象地址。现在该地址是对象引用；值类型成了引用类型。

C#编译器自动生成对值类型实例进行装箱所需的 IL 代码。但是，我们仍需理解内部发生的事情，对代码长度和性能做到心中有数。

C#编译器检测到上述代码是向要求引用类型的方法传递值类型，所以自动生成代码对对象进行装箱。因此在运行时，当前存在于 Point 值类型实例 p 中的字段会复制到新分配的 Point 对象中。然后，将已装箱 Point 对象(现在是引用类型)的地址返回并传给 Add 方法。Point 对象会一直存在于堆中，直至被垃圾回收。Point 值类型变量 p 可被重用，因为 ArrayList 不知道关于它的任何事情。所以，在这种情况下，已装箱值类型的生存期超过了未装箱值类型的生存期。

注意：FCL 现在包含一组新的泛型集合类，非泛型集合类已成为“昨日黄花”。例如，应该使用 System.Collections.Generic.List<T>类而不是 System.Collections.ArrayList

类。泛型集合类对非泛型集合类进行了大量改进。例如，API 得到简化和增强，集合类的性能也得到显著提升。但最大的改进在于，泛型集合类允许开发人员在操作值类型的集合时不需要对集合中的项进行装箱/拆箱。单这一项改进，就使性能提升了不少。这是因为托管堆中需要创建的对象减少了，进而减少了应用程序需要执行的垃圾回收的次数。另外，开发人员还获得了编译时的类型安全性，源代码也因为强制类型转换的次数减少而变得更清晰。所有这一切都将在第 12 章“泛型”详细解释。

知道装箱如何进行后，接着谈谈拆箱。假定要用以下代码获取 ArrayList 的第一个元素：

```
Point p = (Point) a[0];
```

它获取 ArrayList 的元素 0 包含的引用(或指针)，试图将其放到 Point 值类型的实例 p 中。为此，已装箱 Point 对象中的所有字段都必须复制到值类型变量 p 中，后者在线程栈上。CLR 分两步完成复制。第一步获取已装箱 Point 对象中的各个 Point 字段的地址。这个过程称为**拆箱**(unboxing)。第二步将字段包含的值从堆复制到基于栈的值类型实例中。

拆箱不是直接将装箱过程倒过来。拆箱的代价比装箱低得多。拆箱其实就是获取指针的过程，该指针指向包含在一个对象中的原始值类型(数据字段)。其实，指针指向的是已装箱实例中的未装箱部分。所以和装箱不同，拆箱不涉及在内存中复制任何字节。知道这个重要区别之后，还应知道的一个重点是，往往紧接着拆箱会发生一次字段复制。

装箱和拆箱/复制显然会对应用程序的速度和内存消耗产生不利影响，所以应留意编译器在什么时候生成代码来自动进行这些操作。并尝试手动编写代码，尽量减少这种情况的发生。

已装箱值类型实例在拆箱时，内部发生下面这些事情。

1. 如果包含“对已装箱值类型实例的引用”的变量为 null，那么会抛出 NullReferenceException 异常。
2. 如果引用的对象不是所需值类型的已装箱实例，那么会抛出 InvalidCastException 异常。[①]

第二条意味着以下代码的工作方式可能跟你想的不一样：

```
public static void Main() {
    Int32 x = 5;
    Object o = x;           // 对 x 装箱，o 引用已装箱对象
    Int16 y = (Int16) o;    // 抛出 InvalidCastException 异常
}
```

从逻辑上说，完全能获取 o 引用的已装箱 Int32，将其强制转型为 Int16。但在对对象进行拆箱时，只能转型为最初未装箱的值类型——本例是 Int32。以下是上述代码的正确写

① CLR 还允许将值类型拆箱为相同值类型的可空版本。详情将在第 19 章讨论。

法：

```
public static void Main() {
    Int32 x = 5;
    Object o = x;                // 对 x 进行装箱，o 引用已装箱对象
    Int16 y = (Int16)(Int32) o;  // 先拆箱为正确类型，再转型
}
```

前面说过，在一次拆箱操作后，经常紧接着执行一次字段复制。以下 C#代码演示了拆箱和复制：

```
public static void Main() {
    Point p;
    p.x = p.y = 1;
    Object o = p;       // 对 p 装箱；o 引用已装箱实例

    p = (Point) o;      // 对 o 拆箱，将字段从已装箱实例复制到栈变量中
}
```

最后一行，C#编译器生成一条 IL 指令对 o 拆箱(获取已装箱实例中的字段的地址)，并生成另一条 IL 指令将这些字段从堆复制到基于栈的变量 p 中。

再来看看以下代码：

```
public static void Main() {
    Point p;
    p.x = p.y = 1;
    Object o = p;           // 对 p 装箱；o 引用已装箱实例

    // 将 Point 的 x 字段变成 2
    p = (Point) o;          // 对 o 拆箱，并将字段从已装箱的实例复制到栈变量中
    p.x = 2;                // 更改栈变量的状态
    o = p;                  // 对 p 装箱；o 引用新的已装箱实例
}
```

最后三行代码唯一的目的就是将 Point 的 x 字段从 1 变成 2。为此，首先要执行一次拆箱，再执行一次字段复制，再更改字段(在栈上)，最后执行一次装箱(在托管堆上创建全新的已装箱实例)。想必你已体会到了装箱和拆箱/复制对应用程序性能的影响。

有的语言(比如 C++/CLI)允许在不复制字段的前提下对已装箱的值类型进行拆箱。拆箱返回已装箱对象中的未装箱部分的地址(忽略对象的“类型对象指针”和“同步块索引”这两个额外的成员)。接着，可以利用这个指针来操纵未装箱实例的字段(这些字段恰好在堆上的已装箱对象中)。例如，上述代码用 C++/CLI 来写，效率会高很多，因为可以直接在已装箱 Point 实例中修改 Point 的 x 字段的值。这就避免了在堆上分配新对象和复制所有字段两次！

重要提示：如果关心应用程序的性能，就应该搞清楚编译器何时生成代码执行这些操作。遗憾的是，许多编译器都隐式生成代码来装箱对象，所以有时并不知道自己的代码

会造成装箱。如果关心特定算法的性能，可以使用 ILDasm.exe 这样的工具查看方法的 IL 代码，观察 IL 指令 box 都在哪些地方出现。

再来看几个装箱和拆箱的例子：

```
public static void Main() {
    Int32 v = 5;         // 创建未装箱值类型变量
    Object o = v;        // o 引用已装箱的、包含值 5 的 Int32
    v = 123;             // 将未装箱的值修改成 123

    Console.WriteLine(v + ", " + (Int32) o); // 显示"123, 5"
}
```

能从上述代码中看出发生了多少次装箱吗？如果说 3 次，会不会觉得意外？让我们仔细分析一下代码，理解具体发生的事情。为了帮助理解，下面列出为这个 Main 方法生成的 IL 代码。我为这些代码加上了注释，方便你看清楚发生的每个操作：

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // 代码大小       45 (0x2d)
  .maxstack 3
  .locals init ([0]int32 v,
           [1] object o)
  // 将 5 加载到 v 中
  IL_0000: ldc.i4.5
  IL_0001: stloc.0

  // 对 v 装箱，将引用指针存储到 o 中
  IL_0002: ldloc.0
  IL_0003: box         [mscorlib]System.Int32
  IL_0008: stloc.1

  // 将 123 加载到 v 中
  IL_0009: ldc.i4.s 123
  IL_000b: stloc.0

  // 对 v 装箱，将指针保留在栈上以进行 Concat(连接)操作
  IL_000c: ldloc.0
  IL_000d: box         [mscorlib]System.Int32

  // 将字符串加载到栈上以执行 Concat 操作
  IL_0012: ldstr   ", "

  // 对 o 拆箱：获取一个指针，它指向栈上的 Int32 字段
  IL_0017: ldloc.1
  IL_0018: unbox.any [mscorlib]System.Int32

  // 对 Int32 装箱，将指针保留在栈上以进行 Concat 操作
  IL_001d: box         [mscorlib]System.Int32
```

```
    // 调用 Concat
    IL_0022: call string [mscorlib]System.String::Conct(  object,
                                                           object,
                                                           object)
    // 将从 Concat 返回的字符串传给 WriteLine
    IL_0027: call void [mscorlib]System.Console::WriteLine(string)

    // 从 Main 返回，终止应用程序
    IL_002c: ret
  } // end of method App::Main
```

首先在栈上创建一个 Int32 未装箱值类型实例(v)，将其初始化为 5。再创建 Object 类型的变量(o)并初始化，让它指向 v。但由于引用类型的变量始终指向堆中的对象，所以 C#生成正确的 IL 代码对 v 进行装箱，将 v 的已装箱拷贝的地址存储到 o 中。接着，值 123 被放到未装箱值类型实例 v 中，但这个操作不会影响已装箱的 Int32，后者的值依然为 5。

接着调用 WriteLine 方法，WriteLine 要求获取一个 String 对象，但当前没有 String 对象。相反，现在有三个数据项：一个未装箱的 Int32 值类型实例(v)，一个 String(它是引用类型)，以及对已装箱 Int32 值类型实例的引用(o)，它要转型为未装箱的 Int32。必须以某种方式合并这些数据项来创建一个 String。

为了创建一个 String，C#编译器生成代码来调用 String 的静态方法 Concat。该方法有几个重载版本，所有版本执行的操作都一样，只是参数的数量不同。由于需要连接[①]三个数据项来创建字符串，所以编译器选择 Concat 方法的以下版本：

```
public static String Concat(Object arg0, Object arg1, Object arg2);
```

为第一个参数 arg0 传递的是 v。但 v 是未装箱的值参数，而 arg0 是 Object，所以必须对 v 进行装箱，并将已装箱的 v 的地址传给 arg0。对于 arg1 参数，", "这个字符串作为一个 String 对象引用传递。对于 arg2 参数，o(一个 Object 引用)会转型为 Int32。这要求执行拆箱(但不紧接着执行复制)，从而获取包含在已装箱 Int32 中的未装箱 Int32 的地址。这个未装箱的 Int32 实例必须再次装箱，并将新的已装箱实例的内存地址传给 Concat 的 arg2 参数。

Concat 方法调用指定的每个对象的 ToString 方法，将每个对象的字符串形式连接起来。从 Concat 返回的 String 对象传给 WriteLine 方法以显示最终结果。

应该指出，如果像下面这样写 WriteLine 调用，生成的 IL 代码将具有更高的执行效率：

```
Console.WriteLine(v + ", " + o);  // 显示"123, 5"
```

[①] 字符串“连接”的另一种说法是字符串“拼接”，是对 concatenation 的不同翻译。文档中采用“连接”。——译注

这和前面的版本几乎完全一致，只是移除了变量 o 之前的(Int32)强制转型。之所以效率更高，是因为 o 已经是指向一个 Object 的引用类型，它的地址可以直接传给 Concat 方法。所以，移除强制转型避免了两次操作：一次拆箱和一次装箱。不妨重新生成应用程序，观察 IL 代码来体会避免的额外操作：

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // 代码大小       35 (0x23)
  .maxstack 3
  .locals init ([0] int32 v,
           [1] object o)


  // 将 5 加载到 v 中
  IL_0000: ldc.i4.5
  IL_0001: stloc.0

  // 对 v 装箱，并将引用指针存储到 o 中
  IL_0002: ldloc.0
  IL_0003: box         [mscorlib]System.Int32
  IL_0008: stloc.1

  // 将 123 加载到 v 中
  IL_0009: ldc.i4.s 123
  IL_000b: stloc.0

  // 对 v 装箱，并将指针保留在栈上以进行 Concat(连接)操作
  IL_000c: ldloc.0
  IL_000d: box         [mscorlib]System.Int32

  // 将字符串加载到栈上以执行 Concat 操作
  IL_0012: ldstr   ", "

  // 将已装箱 Int32 的地址加载到栈上以进行 Concat 操作
  IL_0017: ldloc.1

  // 调用 Concat
  IL_0018: call string [mscorlib]System.String::Conct(  object,
                                                        object,
                                                        object)

  // 将从 Concat 返回的字符串传给 WriteLine
  IL_001d: call void [mscorlib]System.Console::WriteLine(string)

  // 从 Main 返回，终止这个应用程序
  IL_0022: ret
} // end of method App::Main
```

简单对比一下两个版本的 Main 方法的 IL 代码，会发现没有(Int32)转型的版本比有转型的

版本小了 10 字节。第一个版本额外的拆箱/装箱步骤显然会生成更多的代码。更大的问题是，额外的装箱步骤会从托管堆中分配一个额外的对象，将来必须对其进行垃圾回收。这两个版本的结果一样，速度上的差别也并不明显。但是，假如在循环中发生额外的、不必要的装箱操作，就会严重影响应用程序的性能和内存消耗。

甚至可以这样调用 WriteLine，进一步提升上述代码的性能：

```
Console.WriteLine(v.ToString() + ", " + o);    // 显示"123, 5"
```

这会为未装箱的值类型实例 v 调用 ToString 方法，它返回一个 String。String 对象已经是引用类型，所以能直接传给 Concat 方法，不需要任何装箱操作。

下面是演示装箱和拆箱的另一个例子：

```
public static void Main() {
  Int32 v = 5;                    // 创建未装箱的值类型变量
  Object o = v;                   // o 引用 v 的已装箱版本

  v = 123;                        // 将未装箱的值类型修改成 123
  Console.WriteLine(v);           // 显示"123"
  v = (Int32) o;                  // 拆箱并将 o 复制到 v
  Console.WriteLine(v);           // 显示"5"
}
```

上述代码发生了多少次装箱？答案是一次。之所以只发生一次装箱，是因为 System.Console 类已定义了获取单个 Int32 参数的 WriteLine 方法：

```
public static void WriteLine(Int32 value);
```

在前面对 WriteLine 的两次调用中，变量 v(Int32 未装箱值类型实例)以传值方式传给方法。虽然 WriteLine 方法也许会在它自己内部对 Int32 装箱，但这已经不在我们的控制范围之内了。最重要的是，我们已尽可能地在*自己*的代码中减少了装箱。

仔细研究一下 FCL，会发现许多方法都针对不同的值类型参数进行了重载。例如，System.Console 类型提供了 WriteLine 方法的几个重载版本：

```
public static void WriteLine(Boolean);
public static void WriteLine(Char);
public static void WriteLine(Char[]);
public static void WriteLine(Int32);
public static void WriteLine(UInt32);
public static void WriteLine(Int64);
public static void WriteLine(UInt64);
public static void WriteLine(Single);
public static void WriteLine(Double);
public static void WriteLine(Decimal);
public static void WriteLine(Object);
public static void WriteLine(String);
```

以下几个方法也有一组类似的重载版本：System.Console 的 Write 方法，

System.IO.BinaryWriter 的 Write 方法，System.IO.TextWriter 的 Write 和 WriteLine 方法，System.Runtime.Serialization.SerializationInfo 的 AddValue 方法，System.Text.StringBuilder 的 Append 和 Insert 方法。大多数方法之所以要进行重载，唯一的目的就是减少常用值类型的装箱次数。

但是，这些 FCL 类的方法不可能接受你自己定义的值类型。另外，即使是 FCL 中定义好的值类型，这些方法也可能没有提供对应的重载版本。调用方法并传递值类型时，如果不存在与值类型对应的重载版本，那么调用的肯定是获取一个 Object 参数的重载版本。将值类型实例作为 Object 传递会造成装箱，从而对性能造成不利影响。定义自己的类时，可以将类中的方法定义为泛型(通过类型约束将类型参数限制为值类型)。这样方法就可获取任何值类型而不必装箱。泛型主题将在第 12 章讨论。

关于装箱最后要注意一点：如果知道自己的代码会造成编译器反复对一个值类型装箱，请改成用手动方式对值类型进行装箱。这样代码会变得更小、更快。下面是一个例子：

```
using System;

public sealed class Program {
  public static void Main() {
    Int32 v = 5; // 创建未装箱的值类型变量

#if INEFFICIENT
    // 编译下面这一行，v 被装箱 3 次，浪费时间和内存
    Console.WriteLine("{0}, {1}, {2}", v, v, v);
#else
    // 下面的代码结果一样，但无论执行速度，
    // 还是内存利用，都较前面的代码更胜一筹
    Object o = v;  // 对 v 进行手动装箱(仅 1 次)

    // 编译下面这一行不发生装箱
    Console.WriteLine("{0}, {1}, {2}", o, o, o);
#endif
  }
}
```

在定义了 INEFFICIENT 符号的前提下编译，编译器会生成代码对 v 装箱 3 次，造成在堆上分配 3 个对象！这太浪费了，因为每个对象都是完全相同的内容：5。在没有定义 INEFFICIENT 符号的前提下编译，v 只装箱一次，所以只在堆上分配一个对象。随后，在对 Console.WriteLine 方法的调用中，对同一个已装箱对象的引用被传递 3 次。第二个版本执行起来快得多，在堆上分配的内存也要少得多。

通过这些例子，很容易判断在什么时候一个值类型的实例需要装箱。简单地说，要获取对值类型实例的引用，实例就必须装箱。将值类型实例传给需要获取引用类型的方法，就会发生这种情况。但这并不是要对值类型实例装箱的唯一情况。

前面说过，未装箱值类型比引用类型更“轻”。这要归结于以下两个原因。

● 不在托管堆上分配。

● 没有堆上的每个对象都有的额外成员：“类型对象指针”和“同步块索引”。

由于未装箱值类型没有同步块索引，所以不能使用System.Threading.Monitor类型的方法(或者 C# lock 语句)让多个线程同步对实例的访问。

虽然未装箱值类型没有类型对象指针，但仍可调用由类型继承或重写的虚方法(比如Equals，GetHashCode 或者 ToString)。如果值类型重写了其中任何虚方法，那么 CLR 可以非虚地调用该方法，因为值类型隐式密封，不可能有类型从它们派生，而且调用虚方法的值类型实例没有装箱。然而，如果重写的虚方法要调用方法在基类中的实现，那么在调用基类的实现时，值类型实例会装箱，以便能够通过 this 指针将对一个堆对象的引用传给基方法。

但在调用非虚的、继承的方法时(比如GetType或MemberwiseClone)，无论如何都要对值类型进行装箱。因为这些方法由System.Object定义，要求this实参是指向堆对象的指针。

此外，将值类型的未装箱实例转型为类型的某个接口时要对实例进行装箱。这是因为接口变量必须包含对堆对象的引用(接口主题将在第 13 章“接口”中讨论)。以下代码对此进行了演示：

```
using System;

internal struct Point : IComparable {
    private Int32 m_x, m_y;

    // 构造器负责初始化字段
    public Point(Int32 x, Int32 y)
    {
        m_x = x;
        m_y = y;
    }

    // 重写从System.ValueType继承的ToString方法
    public override String ToString()
    {
        // 将point作为字符串返回。注意：调用ToString以避免装箱
        return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
    }

    // 实现类型安全的CompareTo方法
    public Int32 CompareTo(Point other)
    {
        // 利用勾股定理计算哪个point距离原点(0, 0)更远
        return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
          - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
    }
```

```
        // 实现 IComparable 的 CompareTo 方法
        public Int32 CompareTo(Object o) {
            if (GetType() != o.GetType()) {
                throw new ArgumentException("o is not a Point");
            }
            // 调用类型安全的 CompareTo 方法
            return CompareTo((Point) o);
        }
    }

    public static class Program
    {
        public static void Main()
        {
            // 在栈上创建两个 Point 实例
            Point p1 = new Point(10, 10);
            Point p2 = new Point(20, 20);

            // 调用 ToString(虚方法)不装箱 p1
            Console.WriteLine(p1.ToString()); // 显示"(10, 10)"

            // 调用 GetType(非虚方法)时，要对 p1 进行装箱
            Console.WriteLine(p1.GetType());  // 显示"Point"

            // 调用 CompareTo 不装箱 p1
            // 由于调用的是 CompareTo(Point)，所以 p2 不装箱
            Console.WriteLine(p1.CompareTo(p2));   // 显示"-1"

            // p1 要装箱，引用放到 c 中
            IComparable c = p1;
            Console.WriteLine(c.GetType());    // 显示"Point"

            // 调用 CompareTo 不装箱 p1
            // 由于向 CompareTo 传递的不是 Point 变量，
            // 所以调用的是 CompareTo(Object)，它要求获取对已装箱 Point 的引用
            // c 不装箱是因为它本来就引用已装箱 Point
            Console.WriteLine(p1.CompareTo(c));    // 显示"0"

            // c 不装箱，因为它本来就引用已装箱 Point
            // p2 要装箱，因为调用的是 CompareTo(Object)
            Console.WriteLine(c.CompareTo(p2));    // 显示"-1"

            // 对 c 拆箱，字段复制到 p2 中
            p2 = (Point) c;

            // 证明字段已复制到 p2 中
            Console.WriteLine(p2.ToString()); // 显示"(10, 10)"
        }
    }
```

上述代码演示了涉及装箱和拆箱的几种情形。

1. **调用 ToString**

调用 ToString 时 p1 不必装箱。表面看 p1 似乎必须装箱，因为 ToString 是从基类 System.ValueType 继承的虚方法。通常，为了调用虚方法，CLR 需要判断对象的类型来定位类型的方法表。由于 p1 是未装箱的值类型，所以不存在“类型对象指针”。但 JIT 编译器发现 Point 重写了 ToString 方法，所以会生成代码来直接(非虚地)调用 ToString 方法，而不必进行任何装箱操作。编译器知道这里不存在多态性问题，因为 Point 是值类型，没有类型能从它派生以提供虚方法的另一个实现。但假如 Point 的 ToString 方法在内部调用 base.ToString()，那么在调用 System.ValueType 的 ToString 方法时，值类型的实例会被装箱。

2. **调用 GetType**

调用非虚方法 GetType 时 p1 必须装箱。Point 的 GetType 方法是从 System.Object 继承的。所以，为了调用 GetType，CLR 必须使用指向类型对象的指针，而这个指针只能通过装箱 p1 来获得。

3. **调用 CompareTo(第一次)**

第一次调用 CompareTo 时 p1 不必装箱，因为 Point 实现了 CompareTo 方法，编译器能直接调用它。注意向 CompareTo 传递的是一个 Point 变量(p2)，所以编译器调用的是获取一个 Point 参数的 CompareTo 重载版本。这意味着 p2 以传值方式传给 CompareTo，无需装箱。

4. **转型为 IComparable**

p1 转型为接口类型的变量 c 时必须装箱，因为接口被定义为引用类型。装箱 p1 后，指向已装箱对象的指针存储到变量 c 中。后面对 GetType 的调用证明 c 确实引用堆上的已装箱 Point。

5. **调用 CompareTo(第二次)**

第二次调用 CompareTo 时 p1 不必装箱，因为 Point 实现了 CompareTo 方法，编译器能直接调用。注意向 CompareTo 传递的是 IComparable 类型的变量 c，所以编译器调用的是获取一个 Object 参数的 CompareTo 重载版本。这意味着传递的实参必须是指针，必须引用堆上一个对象。幸好，c 确实引用一个已装箱 Point，所以 c 中的内存地址直接传给 CompareTo，无需额外装箱。

6. **调用 CompareTo(第三次)**

第三次调用 CompareTo 时，c 本来就引用堆上的已装箱 Point 对象，所以不装箱。由于 c 是 IComparable 接口类型，所以只能调用接口的获取一个 Object 参数的 CompareTo 方法。这意味着传递的实参必须是引用了堆上对象的指针。所以 p2 要装箱，指向这个已装箱对象的指针将传给 CompareTo。

7. 转型为 Point

将 c 转型为 Point 时，c 引用的堆上对象被拆箱，其字段从堆复制到 p2。p2 是栈上的 Point 类型实例。

我知道，对于引用类型、值类型和装箱的所有这些讨论很容易让人产生挫折感。但是，任何.NET Framework 开发人员只有在切实理解了这些概念之后，才能保证自己的长期成功。相信我，只有在深刻理解了之后，才能更快、更轻松地构建高效率的应用程序。

5.3.1 使用接口更改已装箱值类型中的字段(以及为什么不应该这样做)

下面通过一些例子来验证自己对值类型、装箱和拆箱的理解程度。请研究以下代码，判断它会在控制台上显示什么：

```
using System;

// Point 是值类型
internal struct Point {
  private Int32 m_x, m_y;

  public Point(Int32 x, Int32 y) {
    m_x = x;
    m_y = y;
  }

  public void Change(Int32 x, Int32 y) {
    m_x = x; m_y = y;
  }

  public override String ToString() {
    return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
  }
}

public sealed class Program {
  public static void Main() {
    Point p = new Point(1, 1);

    Console.WriteLine(p);

    p.Change(2, 2);
    Console.WriteLine(p);

    Object o = p;
    Console.WriteLine(o);

    ((Point) o).Change(3, 3);
    Console.WriteLine(o);
```

```
    }
  }
```

程序其实很简单。Main 在栈上创建 Point 值类型的实例(p)，将它的 m_x 和 m_y 字段设为 1。然后，第一次调用 WriteLine 之前 p 要装箱。WriteLine 在已装箱 Point 上调用 ToString，并像预期的那样显示(1, 1)。然后用 p 调用 Change 方法，该方法将 p 在栈上的 m_x 和 m_y 字段值都更改为 2。第二次调用 WriteLine 时，再次对 p 进行装箱，像预料之中的那样显示(2, 2)。

现在，p 进行第 3 次装箱，o 引用已装箱的 Point 对象。第 3 次调用 WriteLine 再次显示(2, 2)，这同样是预料之中的。最后，我们希望调用 Change 方法来更新已装箱的 Point 对象中的字段。然而，Object(变量 o 的类型)对 Change 方法一无所知，所以首先必须将 o 转型为 Point。将 o 转型为 Point 要求对 o 进行拆箱，并将已装箱 Point 中的字段复制到线程栈上的一个临时 Point 中！这个临时 Point 的 m_x 和 m_y 字段会变成 3 和 3，但已装箱的 Point 不受这个 Change 调用的影响。第 4 次调用 WriteLine 方法，会再次显示(2, 2)。这是出乎许多开发人员预料的。

有的语言(比如 C++/CLI)允许更改已装箱值类型中的字段，但 C#不允许。不过，可以用接口欺骗 C#，让它允许这个操作。下面是上例的修改版本：

```
using System;

// 接口定义了 Change 方法
internal interface IChangeBoxedPoint {
   void Change(Int32 x, Int32 y);
}

// Point 是值类型
internal struct Point : IChangeBoxedPoint {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
       m_x = x;
       m_y = y;
    }

  public void Change(Int32 x, Int32 y) {
      m_x = x; m_y = y;
  }

  public override String ToString() {
    return String.Format("({0}, {1})", m_x.ToString(), m_y.ToString());
  }
}

public sealed class Program {
  public static void Main() {
    Point p = new Point(1, 1);
```

```
            Console.WriteLine(p);

            p.Change(2, 2);
            Console.WriteLine(p);

            Object o = p;
            Console.WriteLine(o);

            ((Point) o).Change(3, 3);
            Console.WriteLine(o);

            // 对 p 进行装箱，更改已装箱的对象，然后丢弃它
            ((IChangeBoxedPoint) p).Change(4, 4);
            Console.WriteLine(p);

            // 更改已装箱的对象，并显示它
            ((IChangeBoxedPoint) o).Change(5, 5);
            Console.WriteLine(o);
        }
    }
```

上述代码和上一个版本几乎完全一致，主要区别是 `Change` 方法由 `IChangeBoxedPoint` 接口定义，`Point` 类型现在实现了该接口。`Main` 中的前 4 个 `WriteLine` 调用和前面的例子相同，生成的结果也一样 (这是我们预期的)。然而，`Main` 最后新增了两个例子。

在第一个例子中，未装箱的 `Point p` 转型为一个 `IChangeBoxedPoint`。这个转型造成对 `p` 中的值进行装箱。然后在已装箱值上调用 `Change`，这确实会将其 `m_x` 和 `m_y` 字段分别变成 4 和 4。但在 `Change` 返回之后，已装箱对象立即准备好进行垃圾回收。所以，对 `WriteLine` 的第 5 个调用会显示`(2, 2)`。这同样出乎许多开发人员的预料。

在最后一个例子中，`o` 引用的已装箱 `Point` 转型为一个 `IChangeBoxedPoint`。这不需要装箱，因为 `o` 本来就是已装箱的 `Point`。然后调用 `Change`，它能正确修改已装箱 `Point` 的 `m_x` 和 `m_y` 字段。接口方法 `Change` 使我能够更改已装箱 `Point` 对象中的字段！现在调用 `WriteLine`，会像预期的那样显示`(5, 5)`。本例旨在演示接口方法如何修改已装箱值类型中的字段。在 C#中，不用接口方法便无法做到。

重要提示：本章前面提到，值类型应该“不可变”(immutable)。也就是说，不应定义任何会修改实例字段的成员。事实上，我建议将值类型的字段都标记为 `readonly`。这样，一旦不留神写了一个试图更改字段的方法，编译时就会报错。前面的例子清楚揭示了为什么应该这样做。假如方法试图修改值类型的实例字段，调用这个方法就会产生非预期的行为。构造好值类型后，如果不调用任何会修改其状态的方法(或者如果根本不存在这样的方法)，就用不着操心什么时候发生装箱和拆箱/字段复制。在值类型不可变的情况下，我们可以“无脑”地复制相同的状态，不必担心有方法会修改这些状态，代码的任何行为都在自己的掌控之中。

有许多开发人员审阅了本书内容。在阅读我的部分示例代码之后(比如前面的代码)，他们告诉我以后再也不敢使用值类型了。我必须声明，值类型的这些玄妙之处着实花了我好几天功夫进行调试，痛定思痛之余，我必须在这里着重强调，提醒大家注意，希望大家记住我描述的问题。这样，当代码真正出现这些问题的时候，就能够做到心中有数。虽然如此，但也不要因噎废食而惧怕值类型。它们很有用，有自己的适用场景。毕竟，程序偶尔还是需要 Int32 的。只是要注意，值类型和引用类型的行为会因为使用方式的不同而有明显差异。事实上，前例将 Point 声明为 class 而不是 struct，即可获得令人满意的结果。最后还要告诉你一个好消息，FCL 的核心值类型(Byte，Int32，UInt32，Int64，UInt64，Single，Double，Decimal，BigInteger，Complex 以及所有枚举)都是“不可变”的，所以在使用这些类型时，不会发生任何稀奇古怪的事情。

5.3.2 对象相等性和同一性

开发人员经常写代码比较对象。例如，有时要将对象放到集合，写代码对集合中的对象排序、搜索或比较。本节将讨论相等性和同一性，还将讨论如何定义正确实现了对象相等性的类型。

System.Object 类型提供了名为 Equals 的虚方法，作用是在两个对象包含相同值的前提下返回 true。Object 的 Equals 方法是像下面这样实现的：

```
public class Object {
  public virtual Boolean Equals(Object obj) {

    // 如果两个引用指向同一个对象，它们肯定包含相同的值
    if (this == obj) return true;

    // 假定对象不包含相同的值
    return false;
  }
}
```

乍一看，这似乎就是 Equals 的合理实现：假如 this 和 obj 实参引用同一个对象，就返回 true。似乎合理是因为 Equals 知道对象肯定包含和它自身一样的值。但假如实参引用不同对象，Equals 就无法肯定对象包含的是一样的值，所以返回 false。换言之，对于 Object 的 Equals 方法的默认实现，它实现的实际是**同一性**(identity)，而非**相等性**(equality)。

遗憾的是，Object 的 Equals 方法的默认实现并不合理，而且永远都不应该像这样实现。研究一下类的继承层次结构，并思考如何正确重写(override)Equals 方法，马上会发现问题出在哪里。下面展示了 Equals 方法应该如何正确地实现。

1. 如果 obj 实参为 null，就返回 false，因为调用非静态 Equals 方法时，this 所标识的当前对象显然不为 null。

2. 如果 this 和 obj 实参引用同一个对象，就返回 true。在比较包含大量字段的对象时，这一步能显著提升性能。

3. 如果 this 和 obj 实参引用不同类型的对象，就返回 false。一个 String 对象显然不等于一个 FileStream 对象。

4. 针对类型定义的每个实例字段，将 this 对象中的值与 obj 对象中的值进行比较。任何字段不相等，就返回 false。

5. 调用基类的 Equals 方法来比较它定义的任何字段。如果基类的 Equals 方法返回 false，就返回 false；否则返回 true。

所以，微软本应像下面这样实现 Object 的 Equals 方法：

```
public class Object {
  public virtual Boolean Equals(Object obj) {
    // 要比较的对象不能为 null
    if (obj == null) return false;

    // 如果对象属于不同的类型，则肯定不相等
    if (this.GetType() != obj.GetType()) return false;

    // 如果对象属于相同的类型，那么在它们的所有字段都匹配的前提下返回 true
    // 由于 System.Object 没有定义任何字段，所以字段是匹配的
    return true;
  }
}
```

但是，由于微软没有像这样实现 Object 类的 Equals，所以在你的类中实现 Equals 时，规则远比想象的复杂。类型在重写 Equals 方法时应调用其基类的 Equals 实现(除非基类就是 Object)。另外，由于类型能重写 Object 的 Equals 方法，所以不能再用它测试同一性。为了解决这个问题，Object 提供了静态方法 ReferenceEquals，其原型如下：

```
public class Object {
   public static Boolean ReferenceEquals(Object objA, Object objB) {
       return (objA == objB);
   }
}
```

要想检查同一性(判断两个引用是否指向同一个对象)，务必调用 ReferenceEquals，而不要使用 C#的==操作符(除非先把两个操作数都转型为 Object)。这是因为某个操作数的类型可能重载了==操作符，为其赋予了不同于“同一性”的语义。

可以看出，在涉及对象相等性和同一性的时候，.NET Framework 的设计很容易使人混淆。顺便说一下，System.ValueType(所有值类型的基类)就重写了 Object 的 Equals 方法，并进行了正确的实现来执行值的相等性检查(而不是同一性检查)。ValueType 的 Equals 内部是这样实现的。

1. 如果 obj 实参为 null，就返回 false。

2. 如果 this 和 obj 实参引用不同类型的对象，就返回 false。

3. 针对类型定义的每个实例字段，都将 this 对象中的值与 obj 对象中的值进行比较(通过调用字段的 Equals 方法)。任何字段不相等，就返回 false。

4. 返回 true。ValueType 的 Equals 方法不调用 Object 的 Equals 方法。

在内部，ValueType 的 Equals 方法利用反射(详情将在第 23 章“程序集加载和反射”讲述)完成上述步骤 3。由于 CLR 反射机制慢，定义自己的值类型时应重写 Equals 方法来提供自己的实现，从而提高用自己类型的实例进行值相等性比较的性能。当然，自己的实现不调用 base.Equals。

定义自己的类型时，你重写的 Equals 要符合相等性的 4 个特征。

- Equals 必须自反；x.Equals(x)肯定返回 true。
- Equals 必须对称；x.Equals(y)和 y.Equals(x)返回相同的值。
- Equals 必须可传递；如果 x.Equals(y)返回 true，y.Equals(z)返回 true，那么 x.Equals(z)肯定返回 true。
- Equals 必须一致：比较的两个值不变，Equals 的返回值(true 或 false)也不能变。

如果实现的 Equals 不符合上述任何特征，应用程序就会行为失常。重写 Equals 方法时，可能还需要做下面几件事情。

- **让类型实现 System.IEquatable<T>接口的 Equals 方法**

  这个泛型接口允许定义类型安全的 Equals 方法。通常，你实现的 Equals 方法应获取一个 Object 参数，以便在内部调用类型安全的 Equals 方法。

- **重载==和!=操作符方法**

  通常应实现这些操作符方法，在内部调用类型安全的 Equals。

此外，如果以后要出于排序目的而比较类型的实例，那么类型还应实现 System.IComparable 的 CompareTo 方法和 System.IComparable<T>的类型安全的 CompareTo 方法。如果实现了这些方法，还可考虑重载(overload)各种比较操作符方法(<，<=，>，>=)，在这些方法内部调用类型安全的 CompareTo 方法。

5.4 对象哈希码

FCL 的设计者认为，如果能将任何对象的任何实例放到哈希表集合中，那么会带来很多好处。为此，System.Object 提供了虚方法 GetHashCode，它能获取任意对象的 Int32 哈希码。

如果你定义的类型重写了 Equals 方法，那么还应重写 GetHashCode 方法。事实上，如果类型重写 Equals 的同时没有重写 GetHashCode，Microsoft C#编译器会生成一条警告。例如，编译以下类型会显示警告消息：warning CS0659: "Program"重写 Object.Equals(object o)但不重写 Object.GetHashCode()。

```
public sealed class Program {
  public override Boolean Equals(Object obj) { ... }
}
```

类型定义 Equals 之所以还要定义 GetHashCode，是由于在 System.Collections.Hashtable 类型、System.Collections.Generic.Dictionary 类型以及其他一些集合的实现中，要求两个对象必须具有相同哈希码才被视为相等。所以，重写 Equals 就必须重写 GetHashCode，确保相等性算法和对象哈希码算法一致。

简单地说，向集合添加键/值(key/value)对，首先要获取键对象的哈希码。该哈希码指出键/值对要存储到哪个哈希桶(bucket)中。集合需要查找键时，会获取指定键对象的哈希码。该哈希码标识了现在要以顺序方式搜索的哈希桶，将在其中查找与指定键对象相等的键对象。采用这个算法来存储和查找键，意味着一旦修改了集合中的一个键对象，集合就再也找不到该对象。所以，需要修改哈希表中的键对象时，正确做法是移除原来的键/值对，修改键对象，再将新的键/值对添加回哈希表。

自定义 GetHashCode 方法或许不是一件难事。但取决于数据类型和数据分布情况，可能并不容易设计出能返回良好分布值的哈希算法。下面是一个简单的哈希算法，它用于 Point 对象时也许还不错：

```
internal sealed class Point {
  private readonly Int32 m_x, m_y;
  public override Int32 GetHashCode() {
    return m_x ^ m_y; // 返回 m_x 和 m_y 的 XOR 结果
  }
  …
}
```

选择算法来计算类型实例的哈希码时，请遵守以下规则。

- 这个算法要提供良好的随机分布，使哈希表获得最佳性能。
- 可以在算法中调用基类的 GetHashCode 方法，并包含它的返回值。但一般不要调用 Object 或 ValueType 的 GetHashCode 方法，因为两者的实现都与高性能哈希算法“不沾边”。
- 算法至少使用一个实例字段。
- 理想情况下，算法使用的字段应该不可变(immutable)；也就是说，字段应在对象构造时初始化，在对象生存期“永不言变”。
- 算法执行速度尽量快。
- 包含相同值的不同对象应返回相同哈希码。例如，包含相同文本的两个 String 对象应返回相同哈希码。

System.Object 实现的 GetHashCode 方法对派生类型和其中的字段一无所知，所以返回一个在对象生存期保证不变的编号。

重要提示：假如因为某些原因要实现自己的哈希表集合，或者要在实现的代码中调用 GetHashCode，记住千万不要对哈希码进行持久化，因为哈希码很容易改变。例如，一个类型未来的版本可能使用不同的算法计算对象哈希码。

有个公司没有把这个警告放在心上。在他们的网站上，用户可选择用户名和密码来创建账号。然后，网站获取密码 String，调用 GetHashCode，将哈希码持久性存储到数据库。用户重新登录网站，输入自己的密码。网站再次调用 GetHashCode，并将哈希码与数据库中存储的值比较，匹配就允许访问。不幸的是，公司升级到新版本 CLR 后，String 的 GetHashCode 方法发生了改变，现在返回的是不同的哈希码。结果是所有用户都无法登录！

5.5 dynamic 基元类型

C#是类型安全的编程语言。这意味着所有表达式都解析成类型的实例，编译器生成的代码只执行对该类型有效的操作。和非类型安全的语言相比，类型安全的语言的优势在于：程序员会犯的许多错误都能在编译时检测到，确保代码在尝试执行前是正确的。此外，还能编译出更小、更快的代码，因为能在编译时做更多预设，并在生成的 IL 和元数据中落实预设。

但是，程序许多时候仍需处理一些运行时才会知晓的信息。虽然可以使用类型安全的语言(比如 C#)和这些信息交互，但语法就会比较笨拙，尤其是在涉及大量字符串处理的时候。另外，性能也会有所损失。如果写的是纯 C#应用程序，只有在使用反射(详情参见第 23 章“程序集加载和反射”)的时候，才需要和运行时才能确定的信息打交道。但是，许多开发者在使用 C#时，都要和一些不是用 C#实现的组件进行通信。有的组件是.NET 动态语言，比如 Python 或 Ruby，有的是支持 IDispatch 接口的 COM 对象(可能用原生 C 或 C++实现)，也有的是 HTML 文档对象模型(Document Object Model，DOM)对象(可以用多种语言和技术实现)。构建 Microsoft Silverlight 应用程序时，与 HTML DOM 对象的通信尤其重要。

为了方便开发人员使用反射或者与其他组件通信，C#编译器允许将表达式的类型标记为 dynamic。还可将表达式的结果放到变量中，并将变量类型标记为 dynamic。然后，可以使用这个 dynamic 表达式/变量来调用一个成员，比如字段、属性/索引器、方法、委托以及一元/二元/转换操作符。代码使用 dynamic 表达式/变量调用成员时，编译器将生成特殊 IL 代码来描述所需的操作。这种特殊的代码称为 payload(有效载荷)。在运行时，payload 代码根据 dynamic 表达式/变量引用的对象的实际类型来决定具体执行的操作。

以下代码进行了演示。

```
internal static class DynamicDemo {
   public static void Main() {
      dynamic value;
      for (Int32 demo = 0; demo < 2; demo++) {
         value = (demo == 0) ? (dynamic) 5 : (dynamic) "A";
         value = value + value;
         M(value);
      }
   }

   private static void M(Int32 n) { Console.WriteLine("M(Int32): " + n); }
   private static void M(String s) { Console.WriteLine("M(String): " + s); }
}
```

执行 Main 会得到以下输出：

```
M(Int32): 10
M(String): AA
```

要理解发生的事情，首先就得搞清楚+操作符。它的两个操作数的类型是 dynamic。由于

value 是 dynamic，所以 C#编译器生成 payload 代码在运行时检查 value 的实际类型，决定+操作符实际要做什么。

第一次对+操作符求值，value 包含 5(一个 Int32)，所以结果是 10(也是 Int32)。结果存回 value 变量。然后调用 M 方法，将 value 传给它。编译器针对 M 调用生成 payload 代码，以便在运行时检查传给 M 的实参的实际类型，并决定应该调用 M 方法的哪个重载版本。由于 value 包含一个 Int32，所以调用的是获取 Int32 参数的那个版本。

第二次对+操作符求值，value 包含"A"(一个 String)，所以结果是"AA"("A"和它自己连接)。然后再次调用 M 方法，将 value 传给它。这次 payload 代码判断传给 M 的是一个 String，所以调用获取 String 参数的版本。

如果字段、方法参数或方法返回值的类型是 dynamic，编译器会将该类型转换为 System.Object，并在元数据中向字段、参数或返回类型应用 System.Runtime.CompilerServices.DynamicAttribute 的实例。如果局部变量被指定为 dynamic，则变量类型也会成为 Object，但不会向局部变量应用 DynamicAttribute，因为它限制在方法内部使用。由于 dynamic 其实就是 Object，所以方法签名不能仅靠 dynamic 和 Object 的变化来区分。

泛型类(引用类型)、结构(值类型)、接口、委托或方法的泛型类型实参也可以是 dynamic 类型。编译器将 dynamic 转换成 Object，并向必要的各种元数据应用 DynamicAttribute。注意，使用的泛型代码是已经编译好的，会将类型视为 Object；编译器不在泛型代码中生成 payload 代码，所以不会执行动态调度(dynamic dispatch)。

所有表达式都能隐式转型为 dynamic，因为所有表达式最终都生成从 Object 派生的类型[①]。正常情况下，编译器不允许写代码将表达式从 Object 隐式转型为其他类型；必须显式转型。但是，编译器允许使用隐式转型语法将表达式从 dynamic 转型为其他类型：

```
Object o1 = 123;          // OK：从 Int32 隐式转型为 Object(装箱)
Int32 n1 = o1;            // Error: 不允许从 Object 到 Int32 的隐式转型
Int32 n2 = (Int32) o1;    // OK: 从 Object 显式转型为 Int32(拆箱)

dynamic d1 = 123;         // OK: 从 Int32 隐式转型为 dynamic(装箱)
Int32 n3 = d1;            // OK: 从 dynamic 隐式转型为 Int32(拆箱)
```

从 dynamic 转型为其他类型时，虽然编译器允许省略显式转型，但 CLR 会在运行时验证转型来确保类型安全性。如果对象类型不兼容要转换成的类型，CLR 会抛出 InvalidCastException 异常。

注意，dynamic 表达式的求值结果是一个动态表达式。例如以下代码：

```
dynamic d = 123;
```

① 值类型当然要装箱。

```
var result = M(d);          // 注意：'var result'等同于'dynamic result'
```

代码之所以能通过编译，是因为编译时不知道调用哪个 M 方法，从而不知道 M 的返回类型，所以编译器假定 result 变量具有 dynamic 类型。为了对此进行验证，可以在 Visual Studio 中将鼠标指针放在 var 上。随后，“智能感知”窗口会显示“dynamic:表示将在运行时解析其操作的对象”。如果运行时调用的 M 方法的返回类型是 void，那么将抛出 Microsoft.CSharp.RuntimeBinder.RuntimeBinderException 异常。

重要提示：不要混淆 dynamic 和 var。用 var 声明局部变量只是一种简化语法(语法糖)，它要求编译器根据表达式推断具体数据类型。var 关键字只能在方法内部声明局部变量，而 dynamic 关键字可用于局部变量、字段和参数。表达式不能转型为 var，但能转型为 dynamic。必须显式初始化用 var 声明的变量，但无需初始化用 dynamic 声明的变量。欲知 C#的 var 关键字的详情，请参见 9.2 节“隐式类型的局部变量”。

然而，从 dynamic 转换成另一个静态类型时，结果类型当然是静态类型。类似地，向类型的构造器传递一个或多个 dynamic 实参，结果是所要构造的对象的类型：

```
dynamic d = 123;
var x = (Int32) d;             // 转换: 'var x'等同于'Int32 x'
var dt = new DateTime(d);      // 构造: 'var dt'等同于'DateTime dt'
```

如果dynamic表达式被指定为foreach语句中的集合，或者被指定为using语句中的资源，编译器会生成代码，分别将表达式转型为非泛型 System.IEnumerable 接口或 System.IDisposable 接口。转型成功，就使用表达式，代码正常运行。转型失败，就抛出 Microsoft.CSharp.RuntimeBinder.RuntimeBinderException 异常。

重要提示：dynamic 表达式其实是和 System.Object 一样的类型。编译器假定你在表达式上进行的任何操作都是合法的，所以不会生成任何警告或错误。但如果试图在运行时执行无效的操作，就会抛出异常。此外，Visual Studio 无法提供任何“智能感知”支持来帮助你写针对 dynamic 表达式的代码。虽然能定义对 Object 进行扩展的扩展方法(详情参见第 8 章“方法”)，但不能定义对 dynamic 进行扩展的扩展方法。另外，不能将 lambda 表达式或匿名方法(都在第 17 章“委托”中讨论)作为实参传给 dynamic 方法调用，因为编译器推断不了要使用的类型。

以下示例 C#代码使用 COM IDispatch 创建 Microsoft Office Excel 工作簿，将一个字符串放到单元格 A1 中：

```
using Microsoft.Office.Interop.Excel;
...
public static void Main() {
    Application excel = new Application();
    excel.Visible = true;
    excel.Workbooks.Add(Type.Missing);
    ((Range)excel.Cells[1, 1]).Value
```

```
            = "Text in cell A1"; // 把这个字符串放到单元格 A1 中
}
```

没有 dynamic 类型，excel.Cells[1, 1]的返回值就是 Object 类型，必须先转型为 Range 类型才能访问其 Value 属性。但在为 COM 对象生成可由“运行时”调用的包装器(wrapper)程序集时，COM 方法中使用的任何 VARIANT 实际都转换成 dynamic；这称为**动态化**(dynamification)。因此，由于 excel.Cells[1, 1]是 dynamic 类型，所以不必显式转型为 Range 类型就能访问其 Value 属性。动态化显著简化了与 COM 对象的互操作。下面是简化后的代码：

```
using Microsoft.Office.Interop.Excel;
...
public static void Main() {
    Application excel = new Application();
    excel.Visible = true;
    excel.Workbooks.Add(Type.Missing);
    excel.Cells[1, 1].Value
            = "Text in cell A1"; // 把这个字符串放到单元格 A1 中
}
```

以下代码展示了如何利用反射在 String 目标("Jeffrey Richter")上调用方法("Contains")，向它传递一个 String 实参("ff")，并将 Boolean 结果存储到局部变量 result 中：

```
Object target = "Jeffrey Richter";
Object arg = "ff";

// 在目标上查找和希望的实参类型匹配的方法
Type[] argTypes = new Type[] { arg.GetType() };
MethodInfo method = target.GetType().GetMethod("Contains", argTypes);

// 在目标上调用方法，传递希望的实参
Object[] arguments = new Object[] { arg };
Boolean result = Convert.ToBoolean(method.Invoke(target, arguments));
```

可以利用 C#的 dynamic 类型重写上述代码，从而大幅简化语法：

```
dynamic target = "Jeffrey Richter";
dynamic arg = "ff";
Boolean result = target.Contains(arg);
```

我早先指出 C#编译器会生成 payload 代码，在运行时根据对象实际类型判断要执行什么操作。这些 payload 代码使用了称为**运行时绑定器**(runtime binder)的类。不同编程语言定义了不同的运行时绑定器来封装自己的规则。C#“运行时绑定器”的代码在 Microsoft.CSharp.dll 程序集中，生成使用 dynamic 关键字的项目必须引用该程序集。编译器的默认响应文件 CSC.rsp 中已引用了该程序集。记住，是这个程序集中的代码知道在运行时生成代码，在+操作符应用于两个 Int32 对象时执行加法，在+操作符应用于两个 String 对象时执行连接。

在运行时，Microsoft.CSharp.dll 程序集必须加载到 AppDomain 中，这会损害应用程序的性能，增大内存消耗。Microsoft.CSharp.dll 还会加载 System.dll 和 System.Core.dll。如果使用 dynamic 与 COM 组件互操作，还会加载 System.Dynamic.dll。payload 代码执行时，会在运行时生成动态代码；这些代码进入驻留于内存的程序集，即“匿名寄宿的 DynamicMethods 程序集”(Anonymously Hosted DynamicMethods Assembly)，作用是当特定 call site[①]使用具有相同运行时类型的动态实参发出大量调用时增强动态调度性能。

C#内建的动态求值功能所产生的额外开销不容忽视。虽然能用动态功能简化语法，但也要看是否值得。毕竟，加载所有这些程序集以及额外的内存消耗，会对性能造成额外影响。如果程序中只是一、两个地方需要动态行为，传统做法或许更高效。即调用反射方法(如果是托管对象)，或者进行手动类型转换(如果是 COM 对象)。

在运行时，C#的“运行时绑定器”根据对象的运行时类型分析应采取什么动态操作。绑定器首先检查类型是否实现了 IDynamicMetaObjectProvider 接口。如果是，就调用接口的 GetMetaObject 方法，它返回 DynamicMetaObject 的一个派生类型。该类型能处理对象的所有成员、方法和操作符绑定。IDynamicMetaObjectProvider 接口和 DynamicMetaObject 基类都在 System.Dynamic 命名空间中定义，都位于 System.Core.dll 程序集中。

像 Python 和 Ruby 这样的动态语言，是为它们的类型赋予了从 DynamicMetaObject 派生的类型，以便能从其他编程语言(比如 C#)中以恰当的方式访问。类似地，访问 COM 组件时，C#的“运行时绑定器”会使用知道如何与 COM 组件通信的 DynamicMetaObject 派生类型。COM DynamicMetaObject 派生类型在 System.Dynamic.dll 程序集中定义。

如果在动态表达式中使用的一个对象的类型未实现 IDynamicMetaObjectProvider 接口，C#编译器会将对象视为用 C#定义的普通类型的实例，利用反射在对象上执行操作。

dynamic 的一个限制是只能访问对象的实例成员，因为 dynamic 变量必须引用对象。但是，有时需要动态调用在运行时才能确定的一个类型的静态成员。我为此创建了 StaticMemberDynamicWrapper 类，它从 System.Dynamic.DynamicObject 派生。后者实现了 IDynamicMetaObjectProvider 接口。类内部使用了相当多的反射(这个主题将在第 23 章讨论)。以下是我的 StaticMemberDynamicWrapper 类的完整代码。

```
internal sealed class StaticMemberDynamicWrapper : DynamicObject {
   private readonly TypeInfo m_type;
   public StaticMemberDynamicWrapper(Type type) { m_type = type.GetTypeInfo(); }

   public override IEnumerable<String> GetDynamicMemberNames() {
      return m_type.DeclaredMembers.Select(mi => mi.Name);
   }

   public override Boolean TryGetMember(GetMemberBinder binder, out object result) {
```

① call site 是发出调用的地方，可理解成调用了一个目标方法的表达式或代码行。——译注

```
         result = null;
         var field = FindField(binder.Name);
         if (field != null) { result = field.GetValue(null); return true; }

         var prop = FindProperty(binder.Name, true);
         if (prop != null) { result = prop.GetValue(null, null); return true; }
         return false;
      }

      public override Boolean TrySetMember(SetMemberBinder binder, object value) {
         var field = FindField(binder.Name);
         if (field != null) { field.SetValue(null, value); return true; }

         var prop = FindProperty(binder.Name, false);
         if (prop != null) { prop.SetValue(null, value, null); return true; }
         return false;
      }

      public override Boolean TryInvokeMember(InvokeMemberBinder binder, Object[] args,
         out Object result) {
         MethodInfo method = FindMethod(binder.Name,
                              args.Select(c=>c.GetType()).ToArray());
         if (method == null) { result = null; return false; }
         result = method.Invoke(null, args);
         return true;
   }

      private MethodInfo FindMethod(String name, Type[] paramTypes) {
         return m_type.DeclaredMethods.FirstOrDefault(mi => mi.IsPublic && mi.IsStatic
                && mi.Name == name
                && ParametersMatch(mi.GetParameters(), paramTypes));
      }

      private Boolean ParametersMatch(ParameterInfo[] parameters, Type[] paramTypes) {
         if (parameters.Length != paramTypes.Length) return false;
         for (Int32 i = 0; i < parameters.Length; i++)
            if (parameters[i].ParameterType != paramTypes[i]) return false;
         return true;
      }

      private FieldInfo FindField(String name) {
         return m_type.DeclaredFields.FirstOrDefault(fi => fi.IsPublic && fi.IsStatic
            && fi.Name == name);
   }

      private PropertyInfo FindProperty(String name, Boolean get) {
         if (get)
            return m_type.DeclaredProperties.FirstOrDefault(
               pi => pi.Name == name && pi.GetMethod != null &&
               pi.GetMethod.IsPublic && pi.GetMethod.IsStatic);
```

```
        return m_type.DeclaredProperties.FirstOrDefault(
            pi => pi.Name == name && pi.SetMethod != null &&
                pi.SetMethod.IsPublic && pi.SetMethod.IsStatic);
    }
}
```

为了动态调用静态成员，传递想要操作的 Type 来构建上述类的实例，将引用放到 dynamic 变量中，再用实例成员语法调用所需的静态成员。下例展示了如何调用 String 的静态 Concat(String, String)方法。

```
dynamic stringType = new StaticMemberDynamicWrapper(typeof(String));
var r = stringType.Concat("A", "B"); // 动态调用 String 的静态 Concat 方法
Console.WriteLine(r); // 显示"AB"
```

第 6 章 类型和成员基础

本章内容：

- 类型的各种成员
- 类型的可见性
- 成员的可访问性
- 静态类
- 分部类、结构和接口
- 组件、多态和版本控制

第 4 章和第 5 章重点介绍了类型以及所有类型的所有实例都支持的一组操作，并指出可以将所有类型划分为引用类型或值类型。在本章及本部分后续的章节，将解释如何在类型中定义各种成员，从而设计出符合需要的类型。第 7 章～第 11 章将详细讨论每种成员。

6.1 类型的各种成员

类型中可定义 0 个或多个以下种类的成员。

- **常量** 常量是指出数据值恒定不变的符号。这种符号使代码更易阅读和维护。常量总与类型关联，不与类型的实例关联。常量逻辑上总是静态成员。相关内容在第 7 章“常量和字段”讨论。

- **字段** 字段表示只读或可读/可写的数据值。字段可以是静态的；这种字段被认为是*类型*状态的一部分。字段也可以是实例(非静态)；这种字段被认为是*对象*状态的一部分。强烈建议将字段声明为私有，防止类型或对象的状态被类型外部的代码破坏。相关内容在第 7 章讨论。

- **实例构造器** 实例构造器是将新*对象*的实例字段初始化为良好初始状态的特殊方法。相关内容在第 8 章“方法”讨论。

- **类型构造器** 类型构造器是将*类型*的静态字段初始化为良好初始状态的特殊方法。相关内容在第 8 章讨论。

- **方法** 方法是更改或查询类型或对象状态的函数。作用于*类型*称为静态方法，作用于

*对象*称为实例方法。方法通常要读写类型或对象的字段。相关内容在第 8 章讨论。

- **操作符重载** 操作符重载实际是方法，定义了当操作符作用于对象时，应该如何操作该对象。由于不是所有编程语言都支持操作符重载，所以操作符重载方法不是“公共语言规范”(Common Language Specification，CLS)的一部分。相关内容在第 8 章讨论。

- **转换操作符** 转换操作符是定义如何隐式或显式将对象从一种类型转型为另一种类型的方法。和操作符重载方法一样，并不是所有编程语言都支持转换操作符，所以不是 CLS 的一部分。相关内容在第 8 章讨论。

- **属性** 属性允许用简单的、字段风格的语法设置或查询类型或对象的逻辑状态，同时保证状态不被破坏。作用于*类型*称为静态属性，作用于*对象*称为实例属性。属性可以无参(非常普遍)，也可以有多个参数(相当少见，但集合类用得多)。相关内容在第 10 章“属性”讨论。

- **事件** 静态事件允许*类型*向一个或多个静态或实例方法发送通知。实例(非静态)事件允许*对象*向一个或多个静态或实例方法发送通知。引发事件通常是为了响应提供事件的那个类型或对象的状态改变。事件包含两个方法，允许静态或实例方法登记或注销对该事件的关注。除了这两个方法，事件通常还用一个委托字段来维护已登记的方法集。相关内容在第 11 章“事件”讨论。

- **类型** 类型可以定义其他嵌套类型。通常用这个办法将大的、复杂的类型分解成更小的构建单元(building block)以简化实现。

再次声明，本章宗旨并非详细描述各种成员，而是帮你打好基础，阐明这些成员的共性。

无论什么编程语言，编译器都必须能处理源代码，为上述每种成员生成元数据和 IL 代码。所有编程语言生成的元数据格式完全一致。这正是 CLR 成为“公共语言运行时”的原因。元数据是所有语言都生成和使用的公共信息。正是由于有了元数据，用一种语言写的代码才能无缝访问用另一种语言写的代码。

CLR 还利用公共元数据格式决定常量、字段、构造器、方法、属性和事件在运行时的行为。简单地说，元数据是整个 Microsoft .NET Framework 开发平台的关键，它实现了编程语言、类型和对象的无缝集成。

以下 C#代码展示了一个类型定义，其中包含所有可能的成员。代码能通过编译(有一些警告)，但你平时应该不会这样创建类型。大多数方法都没有实用价值，仅仅是为了示范编译器如何将类型及其成员转换成元数据。再次说明，后面几章会逐一对这些成员进行讨论。

```
using System;

public sealed class SomeType  {                               // 1

   // 嵌套类
   private class SomeNestedType{}                             // 2
```

```
    // 常量、只读和静态可读/可写字段
    private const Int32 c_SomeConstant = 1;                // 3
    private readonly String m_SomeReadOnlyField = "2";     // 4
    private static Int32 s_SomeReadWriteField = 3;         // 5

    // 类型构造器
    static SomeType(){}                                     // 6

    // 实例构造器
    public SomeType(Int32 x) { }                            // 7
    public SomeType() { }                                   // 8

    // 实例方法和静态方法
    private String InstanceMethod() {return null;}          // 9
    public static void Main(){}                             //10

    // 实例属性
    public Int32 SomeProp{                                  //11
        get{ return 0; }                                    //12
        set{  }                                             //13
    }

    // 实例有参属性(索引器)
    public Int32 this[String s] {                           //14
        get{return 0;}                                      //15
        set{}                                               //16
    }

    // 实例事件
    public event EventHandler SomeEvent;               //17
}
```

编译这个类型，用 ILDasm.exe 查看元数据，将得到如图 6-1 所示的输出。

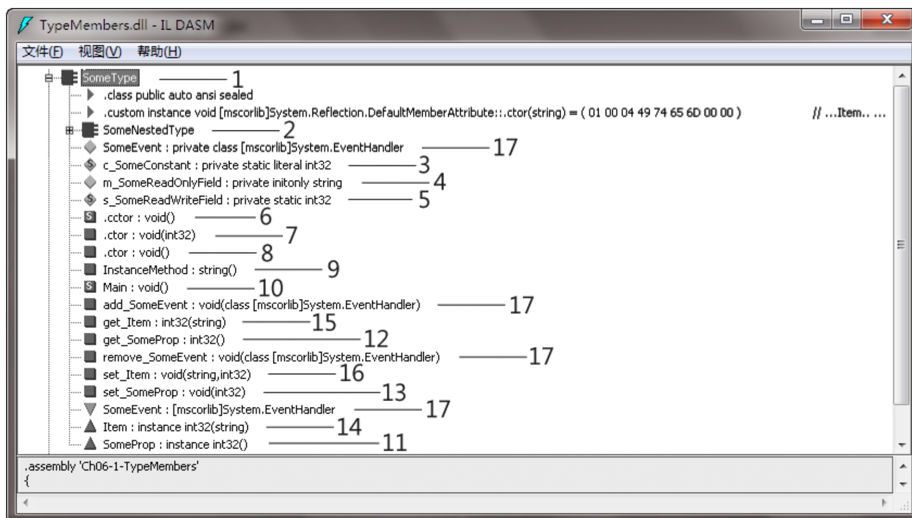


图 6-1 用 ILDasm.exe 查看 SomeType 的元数据

注意，源代码中定义的所有成员都造成编译器生成元数据。事实上，有的成员还造成编译器生成额外的成员和额外的元数据。例如，事件成员(17)造成编译器生成一个字段、两个方法和一些额外的元数据。目前不理解这些内容没有关系。但在学习后面几章时，希望你能回头看看这个例子，体会成员是如何定义的，它们对编译器生成的元数据有何影响。

6.2 类型的可见性

要定义文件范围的类型(而不是定义嵌套在另一个类型中的类型)，可将类型的可见性指定为 public 或 internal。public 类型不仅对定义程序集中的所有代码可见，还对其他程序集中的代码可见。internal 类型则仅对定义程序集中的所有代码可见，对其他程序集中的代码不可见。定义类型时不显式指定可见性，C#编译器会帮你指定 internal(限制比 public 大)。下面是几个例子。

```
using System;

// 以下类型的可见性为 public，既可由本程序集中的代码访问，
// 也可由其他程序集中的代码访问
public class ThisIsAPublicType { ... }

// 以下类型的可见性为 internal，只可由本程序集中的代码访问
internal class ThisIsAnInternalType { ... }

// 由于没有显式声明类型的可见性，以下类型的可见性默认为 internal
class ThisIsAlsoAnInternalType { ... }
```

友元程序集

假定下述情形：某公司的团队 TeamA 在某个程序集中定义了一组实用工具类型(utility type)，并希望公司的另一个团队 TeamB 的成员使用这些类型。但由于各种原因，比如时间安排、地理位置、不同的成本中心或报表结构，这两个团队不能将他们的所有类型都生成到一个程序集中；相反，每个团队都要生成自己的程序集。

为了使团队 TeamB 的程序集能使用团队 TeamA 的类型，TeamA 必须将他们的所有实用工具类型定义为 public。但这意味着工具类型会对所有程序集公开，就连另一家公司的开发人员也能写代码使用它们。这不是公司所希望的。这些实用工具类型也许做出了一些预设，而 TeamB 在写代码的时候会默认这些预设成立。我们希望 TeamA 能有一个办法将他们的实用工具类型定义为 internal，同时仍然允许团队 TeamB 访问这些类型。CLR 和 C#通过**友元程序集**(friend assembly)来提供这方面的支持。用一个程序集中的代码对另一个程序集中的内部类型进行单元测试时，友元程序集功能也能派上用场。

生成程序集时，可用 System.Runtime.CompilerServices 命名空间中的 InternalsVisibleTo 特性标明它认为是“友元”的其他程序集。该特性获取标识友元程序集名称和公钥的字符串参数(传给该特性的字符串绝不能包含版本、语言文化和处理器架构)。注意当程序集认了“友元”之后，友元程序集就能访问该程序集中的所有 internal 类型，以及这些类型的 internal 成员。下例展示一个程序集如何将两个强命名程序集“Wintellect”和“Microsoft”指定为友元程序集：

```
using System;
using System.Runtime.CompilerServices; // 为了 InternalsVisibleTo 特性
```

```
// 当前程序集中的 internal 类型可由以下两个程序集中
// 的任何代码访问(不管什么版本或语言文化)
[assembly:InternalsVisibleTo("Wintellect, PublicKey=12345678...90abcdef")]
[assembly:InternalsVisibleTo("Microsoft, PublicKey=b77a5c56...1934e089")]

internal sealed class SomeInternalType { ... }
internal sealed class AnotherInternalType { ... }
```

从友元程序集访问上述程序集的 internal 类型很容易。例如，下面展示了公钥为"12345678...90abcdef"的友元程序集"Wintellect"如何访问上述程序集的 internal 类型 SomeInternalType。

```
using System;

internal sealed class Foo {
    private static Object SomeMethod() {
     // 这个"Wintellect"程序集能访问另一个程序集的 internal 类型,
     // 就好像那是 public 类型
     SomeInternalType sit = new SomeInternalType();
     return sit;
    }
}
```

由于程序集中的类型的 internal 成员能从友元程序集访问，所以要慎重考虑类型成员的可访问性，以及要将哪些程序集声明为友元。注意，C#编译器在编译友元程序集(该程序集不含 InternalsVisibleTo 特性)时，要求使用编译器开关/out:<*file*>。使用这个编译器开关的原因在于，编译器需要知道准备编译的程序集的名称，从而判断生成的程序集是不是友元程序集。你或许以为 C#编译器能自己判断，因为平时都是它自己确定输出文件名。但事实上，在代码结束编译之前，C#编译器是不知道输出文件名的。因此，使用 /out:<*file*>编译器开关能极大增强编译性能。

另外，如果使用 C#编译器的/t:module 开关来编译模块(而不是编译成程序集)，而且该模块将成为某个友元程序集的一部分，那么需要使用 C#编译器的 /moduleassemblyname:<*string*>开关来编译该模块，它告诉编译器该模块将成为哪个程序集的一部分，使编译器设置模块中的代码，使它们能访问另一个程序集中的 internal 类型。

6.3 成员的可访问性

定义类型的成员(包括嵌套类型)时，可以指定成员的**可访问性**。在代码中引用成员时，成员的可访问性指出引用是否合法。CLR 自己定义了一组可访问性修饰符，但每种编程语言在向成员应用可访问性时，都选择了自己的一组术语以及相应的语法。例如，CLR 使用术语 Assembly 表明成员对同一程序集内的所有代码可见，而 C#语言对应的术语是 internal。

表 6-1 总结了 6 个应用于成员的可访问性修饰符。当然，任何成员要想被访问，都必须在

可见的类型中定义。例如，如果程序集 AssemblyA 定义了含有一个 `public` 方法的 `internal` 类型，那么程序集 AssemblyB 中的代码不能调用该 `public` 方法，因为 `internal` 类型对 AssemblyB 来说不可见。

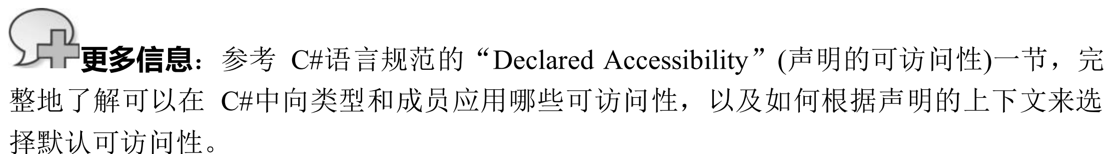
表 6-1 成员的可访问性

CLR 术语	C#术语	描述
Private	`private`	成员只能由定义类型或任何嵌套类型中的方法访问
Family	`protected`	成员只能由定义类型、任何嵌套类型或者不管在什么程序集中的派生类型中的方法访问
Family and Assembly	(不支持)	成员只能由定义类型、任何嵌套类型或者同一程序集中定义的任何派生类型中的方法访问
Assembly	`internal`	成员只能由定义程序集中的方法访问
Family or Assembly	`protected internal`	成员可由任何嵌套类型、任何派生类型(不管在什么程序集)或者定义程序集中的任何方法访问
Public	`public`	成员可由任何程序集的任何方法访问

编译代码时，编程语言的编译器检查代码是不是正确引用了类型和成员。如果代码不正确地引用了类型或成员，编译器会生成一条相应的错误消息。另外，在运行时将 IL 代码编译成本机 CPU 指令时，JIT 编译器也会确保对字段和方法的引用合法。例如，JIT 编译器如果检测到代码不正确地访问了私有字段或方法，将分别抛出 `FieldAccessException` 或 `MethodAccessException` 异常。

通过对 IL 代码进行验证，可以确保被引用成员的可访问性在运行时得到正确兑现——即使语言的编译器忽略了对可访问性的检查。另外，极有可能发生的一种情况是：语言编译器编译的代码访问的是另一个程序集中的另一个类型的 `public` 成员，但到运行时却加载了程序集的不同版本，而新版本中的 `public` 成员变成了 `protected` 或 `private` 成员。

在 C#中，如果没有显式声明成员的可访问性，编译器通常(但并不总是)默认选择 `private`(限制最大的那个)。CLR 要求接口类型的所有成员都具有 `public` 可访问性。C#编译器知道这一点，因此禁止开发人员显式指定接口成员的可访问性；编译器自动将所有成员的可访问性设为 `public`。


更多信息：参考 C#语言规范的“Declared Accessibility”(声明的可访问性)一节，完整地了解可以在 C#中向类型和成员应用哪些可访问性，以及如何根据声明的上下文来选择默认可访问性。

你也许已经注意到了，CLR 提供了称为 Family and Assembly 的可访问性。但 C#不支持。C#开发团队认为这种可访问性基本没用，所以决定放弃。

派生类型重写基类型定义的成员时，C#编译器要求原始成员和重写成员具有相同的可访问性。也就是说，如果基类成员是 protected 的，派生类中的重写成员也必须是 protected 的。但这是 C#的限制，不是 CLR 的。从基类派生时，CLR 允许放宽但不允许收紧成员的可访问性限制。例如，类可以重写基类定义的 protected 方法，将重写方法设为 public(放宽限制)。但不能重写基类定义的 protected 方法，将重写方法设为 private(收紧限制)。之所以不能在派生类中收紧对基类方法的访问，是因为 CLR 承诺派生类总能转型为基类，并获取对基类方法的访问权。如果允许派生类收紧限制，CLR 的承诺就无法兑现了。

6.4 静态类

有一些永远不需要实例化的类，例如 Console，Math，Environment 和 ThreadPool。这些类只有 static 成员。事实上，这种类唯一的作用就是对相关的成员进行分组。例如，Math 类就定义了一组执行数学运算的方法。在 C#中，要用 static 关键字定义不可实例化的类。该关键字只能应用于类，不能应用于结构(值类型)。因为 CLR 总是允许值类型实例化，这是没办法阻止的。

C#编译器对静态类进行了如下限制。

- 静态类必须直接从基类 System.Object 派生，从其他任何基类派生都没有意义。继承只适用于对象，而你不能创建静态类的实例。
- 静态类不能实现任何接口，这是因为只有使用类的实例时，才可以调用类的接口方法。
- 静态类只能定义静态成员(字段、方法、属性和事件)，任何实例成员都会导致编译器报错。
- 静态类不能作为字段、方法参数或局部变量使用，这些都是引用了实例的变量，而静态类与实例“天生相克”。编译器检测到任何这样的用法都会报错。

下面是定义了静态成员的一个静态类。代码虽能通过编译(有一个警告)，但该类没有做任何有意义的事情。

```
using System;

    public static class AStaticClass {
      public static void AStaticMethod() { }

      public static String AStaticProperty {
        get { return s_AStaticField; }
        set { s_AStaticField = value; }
    }

    private static String s_AStaticField;
```

```
        public static event EventHandler AStaticEvent;
    }
```

将上述代码编译成库(DLL)程序集，用 ILDasm.exe 查看会得到如图 6-2 所示的结果。如你所见，使用关键字 `static` 定义类，将导致 C#编译器将该类标记为 `abstract` 和 `sealed`。另外，编译器不在类型中生成实例构造器方法，你在图 6-2 中看不到实例构造器(.ctor)方法。

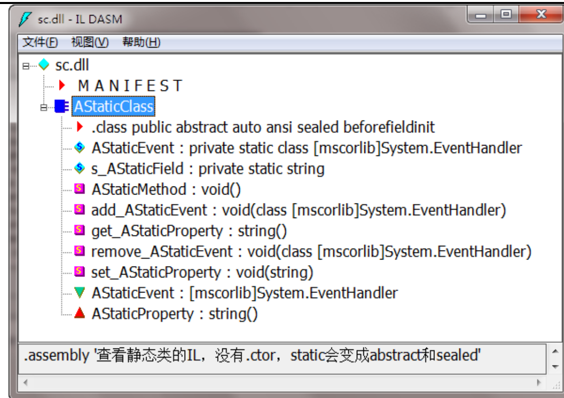


图 6-2 ILDasm.exe 表明静态类在元数据中是抽象密封类

6.5 分部类、结构和接口

本节要讨论分部类、结构和接口。`partial` 关键字告诉 C#编译器：类、结构或接口的定义源代码可能要分散到一个或多个源代码文件中。将类型源代码分散到多个文件的原因有三。

- **源代码控制**

  假定类型定义包含大量源代码，一个程序员把它从源代码控制系统中签出(check out)以进行修改。其他程序员不能同时修改这个类型，除非之后执行合并(merge)。使用 `partial` 关键字可以将类型的代码分散到多个源代码文件中，每个文件都可单独签出，多个程序员能同时编辑类型。

- **在同一个文件中将类或结构分解成不同的逻辑单元**

  我有时会创建一个类型来提供多个功能，使类型能提供完整解决方案。为简化实现，有时会在一个源代码文件中重复声明同一个分部类型。然后，分部类型的每个部分都实现一个功能，并配以它的全部字段、方法、属性、事件等。这样能方便地看到组合以提供一个功能的全体成员，从而简化编码。与此同时，可以方便地将分部类型的一部分注释掉，以便从类中删除一个完整的功能，代之以另一个实现(通过分部类型的一个新的部分)。

- **代码拆分**

  在 Microsoft Visual Studio 中新建项目时，一些源代码文件会作为项目一部分自动创建，其中包含的是为项目打基础的模板。使用 Visual Studio 在设计图面上拖放控件时，Visual Studio 会自动生成源代码，并将代码拆分到不同的源代码文件中。这提高了开发效率。很久以前，生成的代码是直接放到当前正在处理的那个源代码文件中的。这样做的问题在于，如果不小心编辑了一下生成的代码，设计器行为就可能失常。从 Visual Studio 2005 开始，新建窗体、控件等的时候，Visual Studio 自动创建两个源代

码文件：一个用于你的代码，另一个用于设计器生成的代码。由于设计器的代码在单独的文件中，所以基本上杜绝了不小心编辑到它的可能。

要为类型分散于不同文件中的每个部分都应用 `partial` 关键字。这些文件编译到一起时，编译器会合并代码，在最后的.exe 或.dll 程序集文件(或.netmodule 模块文件)中生成完整类型。“分部类型”功能完全由 C#编译器实现，CLR 对该功能一无所知，这解释了一个类型的所有源代码文件为什么必须使用相同编程语言，而且必须作为一个编译单元编译到一起。

6.6 组件、多态和版本控制

面向对象编程(Object-Oriented Programming，OOP)已问世多年。它在上个世纪 70 年代末、80 年代初首次投入应用时，应用程序的规模还很小，而且使应用程序运行起来所需的全部代码都由同一家公司编写。当然，那时确实有操作系统，应用程序也确实使用了操作系统的一些功能，但和今天的操作系统相比，那时的操作系统所提供的功能实在是太少了。

如今软件变得相当复杂，而且用户希望应用程序提供更丰富的功能，如 GUI、菜单、鼠标输入、手写板输入、打印输出、网络功能等。正是由于这个原因，操作系统和开发平台在这几年中取得了迅猛发展。另外，应用程序的开发也必须分工。不能再像以前那样，一个或几个开发人员就能写出一个应用程序需要的全部代码。这要么不可能，要么效率太低。现在的应用程序一般都包含了由许多不同的公司生成的代码。这些代码通过面向对象编程机制契合到一起。

组件软件编程(Component Software Programming，CSP)正是 OOP 发展到极致的成果。下面列举组件的一些特点。

- 组件(.NET Framework 称为程序集)让人觉得它们“已发布”。
- 组件有自己的标识(名称、版本、语言文化和公钥)。
- 组件永远维持自己的标识(程序集中的代码永远不会静态链接到另一个程序集中；.NET 总是使用动态链接)。
- 组件清楚指明它所依赖的组件(引用元数据表)。
- 组件应编档它的类和成员。C#语言通过源代码内的 XML 文档和编译器的`/doc` 命令行开关提供这个功能。
- 组件必须指定它需要的安全权限。CLR 的代码访问安全性(Code Access Security，CAS)机制提供这个功能。
- 组件发布了在任何“维护版本”中都不会改变的一个接口(对象模型)。“维护版本”(servicing version)代表组件的新版本，它旨在向后兼容组件的原始版本。通常，“维护版本”包含 bug 修复、安全补丁或者一些小的功能增强。但是，不能在“维护版本”中要求任何新的依赖关系，也不能要求任何额外的安全权限。

如最后一点所述，CSP 有很大一部分涉及版本控制。组件随着时间而改变，并根据不同的时间表来发布。版本控制使 CSP 的复杂性上升到了 OOP 无法企及的高度。(在 OOP 中，全部代码都由一家公司编写和测试，并作为一个整体发布。)本节将重点放在组件的版本控制上。

.NET Framework 中的版本号包含 4 个部分：主版本号(major version)、次版本号(minor version)、内部版本号(build number)和修订号(revision)。例如，版本号为 1.2.3.4 的程序集，其主版本号为 1，次版本号为 2，内部版本号为 3，修订号为 4。major/minor 部分通常代表程序集的一个连续的、稳定的功能集，而build/revision部分通常代表对这个功能集的一次维护。

假定某公司发布了版本号为 2.7.0.0 的程序集。之后，为了修复该组件的 bug，他们可以生成一个新的程序集，并只改动版本号的 build/revision 部分，比如 2.7.1.34。这表明该程序集是维护版本，向后兼容原始版本(2.7.0.0)。

另一方面，假定该公司想生成程序集的新版本，而且由于发生了重大变化，所以不准备向后兼容程序集的原始版本。在这种情况下，公司实际是要创建一个全新的组件，major/minor 版本号(比如 3.0.0.0)应该和原来的组件不同。

注意：此处只是说明理论上应该如何看待版本号。遗憾的是，CLR 并不以这种方式看待版本号。现在，CLR 将版本号看成是一个不透明的值，而且假如某个程序集依赖版本号为 1.2.3.4 的另一个程序集，那么 CLR 只会尝试加载版本号为 1.2.3.4 的程序集(除非设置了绑定重定向)。

前面讨论了如何使用版本号更新组件的标识，从而反映出组件的新版本。下面要讨论如何利用 CLR 和编程语言(比如 C#)提供的功能来自动适应组件可能发生的变化。

将一个组件(程序集)中定义的类型作为另一个组件(程序集)中的一个类型的基类使用时，便会发生版本控制问题。显然，如果基类的版本(被修改得)低于派生类，派生类的行为也会改变，这可能造成类的行为失常。在多态情形中，由于派生类型会重写基类型定义的虚方法，所以这个问题显得尤其突出。

C#提供了 5 个能影响组件版本控制的关键字，可将它们应用于类型以及/或者类型成员。这些关键字直接对应 CLR 用于支持组件版本控制的功能。表 6-2 总结了与组件版本控制相关的 C#关键字，并描述了每个关键字如何影响类型或者类型成员的定义。

表 6-2 C#关键字及其对组件版本控制的影响

C#关键字	类型	方法/属性/事件	常量/字段
`abstract`	表示不能构造该类型的实例	表示为了构造派生类型的实例，派生类型必须重写并实现这个成员	(不允许)
`virtual`	(不允许)	表示这个成员可由派生类型重写	(不允许)

override	(不允许)	表示派生类型正在重写基类型的成员	(不允许)
sealed	表示该类型不能用作基类型	表示这个成员不能被派生类型重写，只能将该关键字应用于正在重写虚方法的方法	(不允许)
new	应用于嵌套类型、方法、属性、事件、常量或字段时，表示该成员与基类中相似的成员无任何关系		

6.6.3 节“对类型进行版本控制时的虚方法的处理”将演示这些关键字的作用和用法。但在讨论版本控制之前，先要讨论一下 CLR 实际如何调用虚方法。

6.6.1 CLR 如何调用虚方法、属性和事件

本节重点是方法，但我们的讨论也与虚属性和虚事件密切相关。属性和事件实际作为方法实现，本书后面会用专门的章来讨论它们。

方法代表在类型或类型的实例上执行某些操作的代码。在类型上执行操作，称为**静态方法**；在类型的实例上执行操作，称为**非静态方法**。所有方法都有名称、签名和返回类型(可为 void)。CLR 允许类型定义多个同名方法，只要每个方法都有一组不同的参数或者一个不同的返回类型。所以，完全能定义两个同名、同参数的方法，只要两者返回类型不同。但除了 IL 汇编语言，我没有发现任何利用了这一“特点”的语言。大多数语言(包括 C#)在判断方法的唯一性时，除了方法名之外，都只以参数为准，方法返回类型会被忽略。(C# 在定义转换操作符方法时实际上放宽了此限制，详见第 8 章。)

以下 Employee 类定义了三种不同的方法：

```
internal class Employee {
    // 非虚实例方法
    public            Int32    GetYearsEmployed() { ... }

    // 虚方法(“虚”暗示着“实例”)
    public virtual    String   GetProgressReport() { ... }

    // 静态方法
    public static     Employee Lookup(String name) { ... }
}
```

编译上述代码，编译器会在程序集的方法定义表中写入三个记录项，每个记录项都用一组标志(flag)指明方法是实例方法、虚方法还是静态方法。

写代码调用这些方法，生成调用代码的编译器会检查方法定义的标志(flag)，判断应如何生成 IL 代码来正确调用方法。CLR 提供两个方法调用指令。

- **call**
  该 IL 指令可调用静态方法、实例方法和虚方法。用 call 指令调用静态方法，必须指定方法的定义类型。用 call 指令调用实例方法或虚方法，必须指定引用了对象的变量。call 指令假定该变量不为 null。换言之，变量本身的类型指明了方法的定义类

型。如果变量的类型没有定义该方法，就检查基类型来查找匹配方法。call 指令经常用于以非虚方式调用虚方法。

- **callvirt**

  该 IL 指令可调用实例方法和虚方法，不能调用静态方法。用 callvirt 指令调用实例方法或虚方法，必须指定引用了对象的变量。用 callvirt 指令调用非虚实例方法，变量的类型指明了方法的定义类型。用 callvirt 指令调用虚实例方法，CLR 调查发出调用的对象的实际类型，然后以多态方式调用方法。为了确定类型，发出调用的变量绝不能是 null。换言之，编译这个调用时，JIT 编译器会生成代码来验证变量的值是不是 null。如果是，callvirt 指令造成 CLR 抛出 NullReferenceException 异常。正是由于要进行这种额外的检查，所以 callvirt 指令的执行速度比 call 指令稍慢。注意，即使 callvirt 指令调用的是非虚实例方法，也要执行这种 null 检查。

现在综合运用上述知识，看看 C#如何使用这些不同的 IL 指令：

```
using System;

public sealed class Program{
    public static void Main(){
        Console.WriteLine();     //调用静态方法

        Object o = new Object();
        o.GetHashCode();         //调用虚实例方法
        o.GetType();             //调用非虚实例方法
    }
}
```

编译上述代码，查看最后得到的 IL，结果如下：

```
.method public hidebysig static void Main() cil managed {
  .entrypoint
  // Code size 26 (0x1a)
  .maxstack 1
  .locals init (object o)
  IL_0000: call void System.Console::WriteLine()
  IL_0005: newobj instance void System.Object::.ctor()
  IL_000a: stloc.0
  IL_000b: ldloc.0
  IL_000c: callvirt instance int32 System.Object::GetHashCode()
  IL_0011: pop
  IL_0012: ldloc.0
  IL_0013: callvirt instance class System.Type System.Object::GetType()
  IL_0018: pop
  IL_0019: ret
} // end of method Program::Main
```

注意，C#编译器用 call 指令调用 Console 的 WriteLine 方法。这在意料之中，因为 WriteLine 是静态方法。接着用 callvirt 指令调用 GetHashCode，这也在意料之中，因为

GetHashCode 是虚方法。最后，C#编译器用 callvirt 指令调用 GetType 方法。这就有点出乎意料了，因为 GetType 不是虚方法。但这是可行的，因为对代码进行 JIT 编译时，CLR 知道 GetType 不是虚方法，所以在 JIT 编译好的代码中，会直接以非虚方式调用 GetType。

那么，为什么 C#编译器不干脆生成 call 指令呢？答案是 C#团队认为，JIT 编译器应生成代码来验证发出调用的对象不为 null。这意味着对非虚实例方法的调用要稍慢一点。这也意味着以下 C#代码将抛出 NullReferenceException 异常。注意，在另一些编程语言中，以下代码是能正常工作的。

```
using System;

public sealed class Program{
    public Int32 GetFive(){ return 5; }
    public static void Main(){
        Program p = null;
        Int32 x = p.GetFive();  // 在 C#中抛出 NullReferenceException 异常
    }
}
```

上述代码理论上并无问题。虽然变量 p 确实为 null，但在调用非虚方法(GetFive)时，CLR 唯一需要知道的就是 p 的数据类型(Program)。如果真的允许调用 GetFive，那么 this 实参值将是 null。由于 GetFive 方法内部并未使用该实参，所以不会抛出 NullReferenceException 异常。但是，由于 C#编译器生成 callvirt 而不是 call 指令，所以上述代码抛出了 NullReferenceException 异常。

重要提示：将方法定义为非虚后，将来永远都不要把它更改为虚方法。这是因为某些编译器会用 call 而不是 callvirt 调用非虚方法。如果方法从非虚变成虚，而引用(该方法的)代码[1]没有重新编译，那么会以非虚方式调用虚方法，造成应用程序的行为变得无法预料。用 C#写的引用代码不会出问题，因为 C#坚持用 callvirt 指令调用所有实例方法。但是，如果引用代码是用其他语言写的，就可能出问题。

编译器有时用 call 而不是 callvirt 调用虚方法。虽然刚开始有点难以理解，但以下代码证明了有时真的需要这样做：

```
internal class SomeClass {
    // ToString 是基类 Object 定义的虚方法
    public override String ToString() {

    // 编译器使用 IL 指令'call'，
    // 以非虚方式调用 Object 的 ToString 方法
```

[1] 平时说在代码中“引用”一个方法或字段时，就是指在代码中调用或使用它来执行特定功能。——译注

```
        // 如果编译器用'callvirt'而不是'call',
        // 那么该方法将递归调用自身，直至栈溢出
        return base.ToString();
        }
    }
```

调用虚方法 base.ToString 时，C#编译器生成 call 指令来确保以非虚方式调用基类的 ToString 方法。这是必要的，因为如果以虚方式调用 ToString，调用会递归执行，直至线程栈溢出，这显然不是你所期望的。

调用值类型定义的方法时，编译器倾向于使用 call 指令，因为值类型是密封的。这意味着即使值类型含有虚方法也不用考虑多态性，这使调用更快。此外，值类型实例的本质保证它永不为 null，所以永远不抛出 NullReferenceException 异常。最后，如果以虚方式调用值类型中的虚方法，CLR 要获取对值类型的类型对象的引用，以便引用(类型对象中的)方法表，这要求对值类型装箱。装箱对堆造成更大压力，迫使进行更频繁的垃圾回收，使性能受到影响。

无论用 call 还是 callvirt 调用实例方法或虚方法，这些方法通常接收隐藏的 this 实参作为方法第一个参数。this 实参引用了当前要操作的对象。

设计类型时应尽量减少虚方法数量。首先，调用虚方法的速度比调用非虚方法慢。其次，JIT 编译器不能内联(inline)虚方法[①]，这进一步影响性能。第三，虚方法使组件版本控制变得更脆弱，详情参见下一节。第四，定义基类型时，经常要提供一组重载的简便方法(convenience method)。如果希望这些方法是多态的，最好的办法就是使最复杂的方法成为虚方法，使所有重载的简便方法成为非虚方法。顺便说一句，遵循这个原则，还可在改善组件版本控制的同时，不至于对派生类型产生负面影响。下面是一个例子：

```
public class Set {
  private Int32 m_length = 0;

  // 这个重载的简便方法是非虚的
  public Int32 Find(Object value) {
    return Find(value, 0, m_length);
  }

  // 这个重载的简便方法是非虚的
  public Int32 Find(Object value, Int32 startIndex) {
    return Find(value, startIndex, m_length - startIndex);
  }

  // 功能最丰富的方法是虚方法，可以被重写
  public virtual Int32 Find(Object value, Int32 startIndex, Int32 endIndex)
  {
```

[①] 所谓方法的“内联”，就是在调用方法的位置直接嵌入方法的实际代码。——译注

```
        // 可被重写的实现放在这里…
    }

        // 其他方法放在这里
}
```

6.6.2 合理使用类型的可见性和成员的可访问性

使用.NET Framework 时，应用程序是由多个公司生产的多个程序集所定义的类型构成的。这意味着开发人员对所用的组件以及其中定义的类型几乎没有什么控制权。开发人员通常无法访问源代码(甚至可能不知道组件用什么编程语言创建)，而且不同组件的版本发布一般都基于不同的时间表。除此之外，由于多态和受保护成员，基类开发人员必须信任派生类开发人员所写的代码。当然，派生类的开发人员也必须信任从基类继承的代码。设计组件和类型时，应慎重考虑这些问题。

本节描述了设计类型时应如何思考这些问题，具体就是如何正确设置类型的可见性和成员的可访问性来取得最优结果。

首先，在定义一个新类型时，编译器本应默认生成密封类，使其不能作为基类使用。但是，包括 C#编译器在内的许多编译器都默认生成非密封类，只是允许开发人员使用关键字 sealed 将类显式标记为密封。我认为，现在的编译器使用了错误的默认设定。不过，亡羊补牢，为时不晚，希望将来的编译器能改正这一错误。密封类之所以比非密封类更好，是由于以下三方面的原因。

- **版本控制**

  如果类最初密封，将来就可以在不破坏兼容性的前提下更改为非密封。但如果最初非密封，将来就不可能更改为密封，因为这将中断派生类。除此之外，如果非密封类定义了非密封虚方法，那么必须在新版本的类中保持虚方法调用顺序，否则可能中断派生类。例如，如果一个方法先调用虚方法 A，再调用虚方法 B，那么以后不应将代码更改为先调用方法 B，再调用方法 A，因为重写的方法可能依赖于方法的调用顺序。

- **性能**

  如上一节所述，调用虚方法在性能上不及调用非虚方法，因为 CLR 必须在运行时查找对象的类型，判断要调用的方法由哪个类型定义。但是，如果 JIT 编译器看到使用密封类型的虚方法调用，就可直接采用非虚方式调用虚方法，从而生成更高效的代码。之所以能这么做，是因为密封类自然不会有派生类。例如，在下面的代码中，JIT 编译器可以采用非虚方式调用虚方法 ToString：

```
using System;
public sealed class Point {
    private Int32 m_x, m_y;
    public Point(Int32 x, Int32 y) { m_x = x; m_y = y; }

    public override String ToString() {
```

```
            return String.Format("({0}, {1})", m_x, m_y);
        }

        public static void Main() {
            Point p = new Point(3, 4);

            // C#编译器在此生成 callvirt 指令，
            // 但 JIT 编译器将优化这个调用，并生
            // 成代码来非虚地调用 ToString。这
            // 是因为 p 的类型是 Point，而 Point
            // 是密封类
            Console.WriteLine(p.ToString());
        }
    }
```

- **安全性和可预测性**

  类必须保护自己的状态，不允许被破坏。当类处于非密封状态时，只要它的任何数据字段或者在内部对这些字段进行处理的方法是可以访问的，而且不是私有的，派生类就能访问和更改基类的状态。另外，派生类既可重写基类的虚方法，也可直接调用这个虚方法在基类中的实现。一旦将某个方法、属性或事件设为 `virtual`，基类就会丧失对它的行为和状态的部分控制权。所以，除非经过了认真考虑，否则这种做法可能导致对象的行为变得不可预测，还可能留下安全隐患。

密封类的问题在于它可能对类型的用户造成很大的不便。有的时候，开发人员希望从现有类型派生出一个类，在其中添加额外字段或状态信息来满足自己应用程序的需要。他们甚至希望在派生类中定义辅助方法(helper method)或简便方法(convenience method)来操纵这些额外的字段。虽然 CLR 没有提供机制允许你用辅助方法或字段来扩展一个已经生成的类型，但可利用 C#的扩展方法(第 8 章)模拟辅助方法，还可利用 `ConditionalWeakTable` 类(第 21 章)模拟为对象附加状态。

以下是我自己在定义类时遵循的原则。

- 定义类时，除非确定要将其作为基类，并允许派生类对它进行特化[1]，否则总是显式地指定为 `sealed` 类。如前所述，这与 C#以及其他许多编译器的默认方式相反。另外，我默认将类指定为 `internal` 类，除非我希望在程序集外部公开这个类。幸好，如果不显式指定类型的可见性，C#编译器默认使用的就是 `internal`。如果我真的要定义一个可由其他人继承的类，同时不想允许特化，那么我会重写并密封继承的所有虚方法。

- 在类的内部，我总是毫不犹豫地将数据字段定义为 `private`。幸好，C#默认就将字段

[1] 特化(specialization)是指继承了基类的东西不算，还对这些东西进行特殊处理，加入自己的东西。——译注

标记为 private。事实上，我情愿 C#强制所有字段都标记为 private，根本不允许 protected，internal 和 public 等等。状态一旦公开，就极易产生问题，造成对象的行为无法预测，并留下安全隐患。即使只将一些字段声明为 internal 也会如此。即使在单个程序集中，也很难跟踪引用了一个字段的所有代码，尤其是假如代码由几个开发人员编写，并编译到同一个程序集中。

- 在类的内部，我总是将自己的方法、属性和事件定义为 private 和非虚。幸好，C#默认也是这样的。当然，我会将某个方法、属性和事件定义为 public，以便公开类型的某些功能。我会尽量避免将上述任何成员定义为 protected 或 internal，因为这会使类型面临更大的安全风险。即使迫不得已，我也会尽量选择 protected 或 internal。virtual 永远最后才考虑，因为虚成员会放弃许多控制，丧失独立性，变得彻底依赖于派生类的正确行为。

- OOP 有一条古老的格言，大意是当事情变得过于复杂时，就搞更多的类型出来。当算法的实现开始变得复杂时，我会定义一些辅助类型来封装独立的功能。如果定义的辅助类型只由一个“超类型”使用，我会在“超类型”中嵌套这些辅助类型。这样除了可以限制范围，还允许嵌套的辅助类型的代码引用“超类型”所定义的私有成员。但是，Visual Studio 的代码分析工具(FxCopCmd.exe)强制执行了一条设计规则，即对外公开的嵌套类型必须在文件或程序集范围中定义，不能在另一个类型中定义。之所以会有这个规则，是因为一些开发人员觉得引用嵌套类型时，所用的语法过于繁琐。我赞同该规则，自己绝不会定义公共嵌套类型。

6.6.3 对类型进行版本控制时的虚方法的处理

如前所述，在组件软件编程(Component Software Programming，CSP)环境中，版本控制是非常重要的问题。第 3 章已讨论了部分版本控制问题。那一章解释了强命名程序集，并讨论了管理员如何确保应用程序绑定到和生成/测试时一样的程序集。但是，还有其他版本控制问题会造成源代码兼容性问题。例如，如果类型要作为基类型使用，那么增加或修改它的成员时务必非常小心。下面来看一些例子。

假定 CompanyA 定义了 Phone 类型：

```
namespace CompanyA {
    public class Phone {
        public void Dial() {
            Console.WriteLine("Phone.Dial");
            //在这里执行拨号操作
        }
    }
}
```

再假定 CompanyB 定义了 BetterPhone 类型，使用 CompanyA 的 Phone 类型作为基类型：

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {
```

```
        public void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 在这里执行建立连接的操作
        }
    }
}
```

CompanyB 编译上述代码时，C#编译器生成以下警告消息：

```
warning CS0108:"CompanyB.BetterPhone.Dial()"隐藏了继承的成员"CompanyA.Phone.Dial()"。如果
是有意隐藏，请使用关键字 new
```

该警告告诉开发人员 `BetterPhone` 类正在定义一个 `Dial` 方法，它会隐藏 `Phone` 类定义的 `Dial`。新方法可能改变 `Dial` 的语义(这个语义是 `CompanyA` 最初创建 `Dial` 方法时定义的)。

编译器就潜在的语言不匹配问题发出警告，这是一个令人欣赏的设计。编译器甚至贴心地告诉你如何消除这条警告，办法是在 `BetterPhone` 类中定义 `Dial` 时，在前面加一个 `new` 关键字。以下是改正后的 `BetterPhone` 类：

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {

        // 新的 Dial 方法变得与 Phone 的 Dial 方法无关了
        public new void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 在这里执行建立连接的操作
        }
    }
}
```

现在，CompanyB 能在其应用程序中使用 `BetterPhone.Dial`。以下是 CompanyB 可能写的一些示例代码：

```
public sealed class Program{
    public static void Main(){
        CompanyB.BetterPhone phone = new CompanyB.BetterPhone();
        phone.Dial();
    }
}
```

运行上述代码，输出结果如下所示：

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
```

输出符合 CompanyB 的预期。调用 Dial 方法时，会调用由 BetterPhone 类定义的新 Dial。在新 Dial 中，先调用虚方法 EstablishConnection，再调用基类型 Phone 中的 Dial 方法。

现在，假定几家公司计划使用 CompanyA 的 Phone 类型。再假定这几家公司都认为在 Dial 方法中建立连接的主意非常好。CompanyA 收到这个反馈，决定对 Phone 类进行修订：

```
namespace CompanyA{
    public class Phone{
        public void Dial(){
            Console.WriteLine("Phone.Dial");
            EstablishConnection();
            // 在这里执行拨号操作
        }

        protected virtual void EstablishConnection(){
            Console.WriteLine("Phone.EstablishConnection");
            // 在这里执行建立连接的操作
        }
    }
}
```

现在，一旦 CompanyB 编译它的 BetterPhone 类型(从 CompanyA 的新版本 Phone 派生)，编译器将生成以下警告消息：

```
warning CS0114: "CompanyB.BetterPhone.EstablishConnection()" 将隐藏继承的成员 "CompanyA.Phone.EstablishConnection()"。若要使当前成员重写该实现，请添加关键字 override。否则，添加关键字 new。
```

编译器警告 Phone 和 BetterPhone 都提供了 EstablishConnection 方法，而且两者的语义可能不一致。只是简单地重新编译 BetterPhone，可能无法获得和使用第一个版本的 Phone 类型时相同的行为。

如果 CompanyB 认定 EstablishConnection 方法在两个类型中的语义不一致，CompanyB 可以告诉编译器使用 BetterPhone 类中定义的 Dial 和 EstablishConnection 方法，它们与基类型 Phone 中定义的 EstablishConnection 方法没有关系。CompanyB 可以为 EstablishConnection 方法添加 new 关键字来告诉编译器这一点。

```
namespace CompanyB {
    public class BetterPhone : CompanyA.Phone {

        // 保留关键字 new，指明该方法与基类型的
        // Dial 方法没有关系
        public new void Dial() {
            Console.WriteLine("BetterPhone.Dial");
```

```
            EstablishConnection();
            base.Dial();
        }

        // 为这个方法添加关键字 new，指明该方法与基类型的
        // EstablishConnection 方法没有关系
        protected new virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 在这里执行建立连接的操作
        }
    }
}
```

在这段代码中，关键字 new 告诉编译器生成元数据，向 CLR 澄清 BetterPhone 类型的 EstablishConnection 方法应被视为由 BetterPhone 类型引入的一个新函数。这样一来，CLR 就知道 Phone 和 BetterPhone 这两个类中的该同名方法无任何关系。

执行相同的应用程序代码(前面列出的 Main 方法中的代码)，输出结果如下所示：

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
Phone.EstablishConnection
```

这个输出表明，在 Main 方法中调用 Dial，调用的是 BetterPhone 类定义的新 Dial 方法。后者调用了同样由 BetterPhone 类定义的虚方法 EstablishConnection。BetterPhone 的 EstablishConnection 方法返回后，将调用 Phone 的 Dial 方法。Phone 的 Dial 方法调用了 EstablishConnection，但由于 BetterPhone 的 EstablishConnection 使用 new 进行了标识，所以不认为 BetterPhone 的 EstablishConnection 是对 Phone 的虚方法 EstablishConnection 的重写。最终结果是，Phone 的 Dial 方法调用了 Phone 的 EstablishConnection 方法——这正是我们所期望的。

注意：如果编译器像原生 C++编译器那样默认将方法视为重写，那么 BetterPhone 的开发者就不能使用方法名 Dial 和 EstablishConnection 了。这极有可能造成整个源代码 base 的连锁反应，破坏源代码和二进制兼容性。这种波及面太大的改变是我们不希望的，尤其是中大型的项目。但是，如果更改方法名只会造成源代码发生适度更新，就应该更改方法名，避免 Dial 和 EstablishConnection 方法的两种不同的含义使其他开发人员产生混淆。

还有一个办法是，CompanyB 可以获得 CompanyA 的新版本 Phone 类型，并确定 Dial 和 EstablishConnection 在 Phone 中的语义正好是他们所希望的。这种情况下，CompanyB 可通过完全移除 Dial 方法来修改他们的 BetterPhone 类型。另外，由于 CompanyB 现在希望告诉编译器，BetterPhone 的 EstablishConnection 方法和 Phone 的 EstablishConnection 方法是相关的，所以必须移除 new 关键字。但是，仅仅移除 new 关键字还不够，因为编译

器目前还无法准确判断 BetterPhone 的 EstablishConnection 方法的意图。为了准确表示意图，CompanyB 的开发人员还必须将 BetterPhone 的 EstablishConnection 方法由 virtual 改变为 override。以下代码展示了新版本的 BetterPhone 类。

```
namespace CompanyB{
    public class BetterPhone : CompanyA.Phone {

        // 删除 Dial 方法(从基类继承 Dial)

        // 移除关键字'new'，将关键字'virtual'修改为'override'，
        // 指明该方法与基类的 EstablishConnection 方法的关系
        protected override void EstablishConnection(){
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 在这里执行建立连接的操作
        }
    }
}
```

执行相同的应用程序代码(前面列出的 Main 方法中的代码)，输出结果如下所示：

```
Phone.Dial
BetterPhone.EstablishConnection
```

该输出结果表明，在 Main 中调用 Dial 方法，调用的是由 Phone 定义、并由 BetterPhone 继承的 Dial 方法。然后，当 Phone 的 Dial 方法调用虚方法 EstablishConnection 时，实际调用的是 BetterPhone 类的 EstablishConnection 方法，因为它重写了由 Phone 定义的虚方法 EstablishConnection。

第 7 章 常量和字段

本章内容：

- 常量
- 字段

本章介绍如何向类型添加数据成员，具体就是常量和字段。

7.1 常量

常量是值从不变化的符号。定义常量符号时，它的值必须能在编译时确定。确定后，编译器将常量值保存到程序集元数据中。这意味着只能定义编译器识别的基元类型的常量。在C#中，以下类型是基元类型，可用于定义常量：Boolean，Char，Byte，SByte，Int16，UInt16，Int32，UInt32，Int64，UInt64，Single，Double，Decimal 和 String。然而，C#还允许定义非基元类型的一个常量变量(constant variable)，前提是要把值设为 null：

```
using System;

public sealed class SomeType {
    // SomeType 不是基元类型，但 C#允许
    // 值为 null 的这种类型的“常量变量”
    public const SomeType Empty = null;
}
```

由于常量值从不变化，所以常量总是被视为类型定义的一部分。换言之，常量总是被视为静态成员，而不是实例成员。定义常量将导致创建元数据。

代码引用常量符号时，编译器在定义常量的程序集的元数据中查找该符号，提取常量的值，将值嵌入生成的 IL 代码中。由于常量的值直接嵌入代码，所以在运行时不需要为常量分配任何内存[①]。除此之外，不能获取常量的地址，也不能以传引用的方式传递常量。这些限制意味着常量不能很好地支持跨程序集的版本控制。因此，只有确定一个符号的值从不

[①] “不为常量分配内存”不是说它不占内存。在编程中说到“分配(allocate)内存”时，一般都是指在运行时从“堆”中动态分配一块内存。这是一个耗时、耗资源的操作。详情参见 4.1 节。——译注

变化才应定义常量。(将 MaxInt16 定义为 32767 就是一个很好的例子)。下面来演示我刚才所说的内容。首先，请输入以下代码，并将其编译成一个 DLL 程序集。

```
using System;

public sealed class SomeLibraryType {
    // 注意：C#不允许为常量指定 static 关键字，
    // 因为常量总是隐式为 static
    public const Int32 MaxEntriesInList = 50;
}
```

接着用以下代码生成一个应用程序程序集[①]：

```
using System;

public sealed class Program {
    public static void Main() {
        Console.WriteLine("Max entries supported in list: "
            + SomeLibraryType.MaxEntriesInList);
    }
}
```

注意，代码引用了在 SomeLibraryType 类中定义的 MaxEntriesInList 常量。编译器生成应用程序代码时，会注意到 MaxEntriesInList 是值为 50 的常量符号，所以会将 Int32 值 50 嵌入应用程序的 IL 代码，如下所示。事实上，在生成了应用程序程序集之后，运行时根本不会加载 DLL 程序集，可以把它从磁盘上删除。

```
.method public hidebysig static void Main() cil managed
{
   .entrypoint
   // Code size        25 (0x19)

   .maxstack 8
   IL_0000: nop
   IL_0001: ldstr      "Max entries supported in list: "
   IL_0006: ldc.i4.s   50
   IL_0008: box        [mscorlib]System.Int32
   IL_000d: call       string [mscorlib]System.String::Concat(object, object)
   IL_0012: call       void [mscorlib]System.Console::WriteLine(string)
   IL_0017: nop
   IL_0018: ret
} // end of method Program::Main
```

这个例子清楚地展示了版本控制问题。如果开发人员将常量 MaxEntriesInList 的值更改为 1000，并且只是重新生成了 DLL 程序集，那么应用程序程序集不会受到任何影响。应用程序要想获得新值，那么也必须重新编译。如果希望在运行时从一个程序集中动态提取

[①] 用 csc.exe 的/r 开关来引用刚才的.dll 文件。——译注

另一个程序集中的值，那么不应该使用常量，而应该使用 readonly 字段，详情参见下一节。

7.2 字段

字段是一种数据成员，其中容纳了一个值类型的实例或者对一个引用类型的引用。表 7-1 总结了可应用于字段的修饰符。

表 7-1 字段修饰符

CLR 术语	C#术语	说明
Static	Static	这种字段是类型状态的一部分，而非对象状态的一部分
Instance	(默认)	这种字段与类型的一个实例关联，而非与类型本身关联
InitOnly	readonly	这种字段只能由一个构造器方法中的代码写入
Volatile	volatile	编译器、CLR 和硬件不会对访问这种字段的代码执行“线程不安全”的优化措施。只有以下类型才能标记为 volatile：所有引用类型，Single，Boolean，Byte，SByte，Int16，UInt16，Int32，UInt32，Char，以及基础类型为 Byte，SByte，Int16，UInt16，Int32 或 UInt32 的所有枚举类型。volatile 字段将在第 29 章“基元线程同步构造”讨论[①]

如表 7-1 所示，CLR 支持类型(静态)字段和实例(非静态)字段。如果是类型字段，容纳字段数据所需的动态内存是在类型对象中分配的，而类型对象是在类型加载到一个 AppDomain 时创建的(参见第 22 章“CLR 寄宿和 AppDomain”)。那么，什么时候将类型加载到一个 AppDomain 中呢？这通常是在引用了该类型的任何方法首次进行 JIT 编译的时候。如果是实例字段，那么容纳字段数据所需的动态内存是在构造类型的实例时分配的。

由于字段存储在动态(分配的)内存中，所以它们的值在运行时才能获取。字段还解决了常量存在的版本控制问题。此外，字段可以是任何数据类型，不像常量那样仅限于编译器内置的基元类型。

CLR 支持 readonly 字段和 read/write 字段。大多数字段都是 read/write 字段，意味着在代码执行过程中，字段值可多次改变。但是，readonly 字段只能在构造器方法中写入。(构造器方法只能调用一次，即对象首次创建时。)编译器和验证机制确保 readonly 字段不会被构造器以外的其他任何方法写入。注意，可利用反射来修改 readonly 字段。

现在，让我们以 7.1 节“常量”的代码为例，使用一个静态 readonly 字段来修正版本控制

① 文档将 volatile 翻译为“可变”。其实它是“短暂存在”、“易变”的意思，因为可能有多个线程想都对这种字段进行修改，所以“易变”或“易失”更佳。——译注

问题。下面是新版本 DLL 程序集的代码：

```
using System;

public sealed class SomeLibraryType {
    // 字段要和类型关联，就必须使用 static 关键字
    public static readonly Int32 MaxEntriesInList = 50;
}
```

就修改这里就可以了，应用程序的代码不必修改。但是，为了观察新的行为，必须重新生成它。当应用程序的 Main 方法运行时，CLR 将加载 DLL 程序集(现在运行时需要该程序集了)，并从分配给它的动态内存中提取 MaxEntriesInList 字段的值。当然，该值是 50。

假设 DLL 程序集的开发人员将 50 改为 1000，并重新生成程序集。当应用程序代码重新执行时，它会自动提取字段的新值 1000。应用程序不需要重新生成，可以直接运行(尽管性能会受一点影响)。要注意的是，当前假定的是 DLL 程序集的新版本没有进行强命名，而且应用程序的版本策略是让 CLR 加载这个新版本。

下例演示了如何定义一个与类型本身关联的 readonly 静态字段。还定义了 read/write 静态字段，以及 readonly 和 read/write 实例字段。

```
public sealed class SomeType {
    // 这是一个静态 readonly 字段；在运行时对这个类进行初始化时，
    // 它的值会被计算并存储到内存中
    public static readonly Random s_random = new Random();

    // 这是一个静态 read/write 字段
    private static Int32 s_numberOfWrites = 0;

    // 这是一个实例 readonly 字段
    public readonly String Pathname = "Untitled";

    // 这是一个实例 read/write 字段
    private System.IO.FileStream m_fs;

    public SomeType(String pathname) {
        // 该行修改只读字段 pathname，
        //在构造器中可以这样做
        this.Pathname = pathname;
    }

    public String DoSomething() {
        // 该行读写静态 read/write 字段
        s_numberOfWrites = s_numberOfWrites + 1;

        // 该行读取 readonly 实例字段
        return Pathname;
    }
}
```

在上述代码中，许多字段都是内联初始化的[①]。C#允许使用这种简便的内联初始化语法来初始化类的常量、read/write 字段和 readonly 字段。第 8 章“方法”会讲到，C#实际是在构造器中对字段进行初始化的，字段的内联初始化只是一种语法上的简化。另外，在C#中初始化字段时，如果使用内联语法，而不是在构造器中赋值，那么有一些性能问题需要考虑，具体也将在第 8 章讨论。

重要提示：如果某个字段是引用类型，而且该字段被标记为 readonly，那么不可变的是引用，而非字段引用的对象。以下代码对此进行了演示：

```
public sealed class AType {
    // InvalidChars 总是引用同一个数组对象
    public static readonly Char[] InvalidChars = new Char[] {'A','B','C'};
}

public sealed class AnotherType {
    public static void M() {
        // 下面三行代码是合法的，可通过编译，并可成功
        // 修改 InvalidChars 数组中的字符
        AType.InvalidChars[0] = 'X';
        AType.InvalidChars[1] = 'Y';
        AType.InvalidChars[2] = 'Z';

        // 下面这行代码是非法的，无法通过编译，
        // 因为不能让 InvalidChars 引用别的什么东西
        AType.InvalidChars = new Char[] { 'X', 'Y', 'Z' };
    }
}
```

[①] 内联(inline)初始化是指在代码中直接赋值来初始化，而不是将对构造器的调用写出来。——译注

第 8 章 方法

本章内容：

- 实例构造器和类(引用类型)
- 实例构造器和结构(值类型)
- 类型构造器
- 操作符重载方法
- 转换操作符方法
- 扩展方法
- 分部方法

本章重点讨论你将来可能遇到的各种方法，包括实例构造器和类型构造器。还会讲述如何定义方法来重载操作符和类型转换(以进行隐式和显式转型)。还会讨论扩展方法，以便将自己的实例方法从逻辑上“添加”到现有类型中。还会讨论分部方法，允许将类型的实现分散到多个组成部分中。

8.1 实例构造器和类(引用类型)

构造器[①]是将类型的实例初始化为良好状态的特殊方法。构造器方法在“方法定义元数据表”中始终叫做`.ctor`(constructor 的简称)。创建引用类型的实例时，首先为实例的数据字段分配内存，然后初始化对象的附加字段(类型对象指针和同步块索引)[②]，最后调用类型的实例构造器来设置对象的初始状态。

构造引用类型的对象时，在调用类型的实例构造器之前，为对象分配的内存总是先被“置零”。没有被构造器显式重写的所有字段都保证获得`0`或`null`值。

和其他方法不同，实例构造器永远不能被继承。也就是说，类只有类自己定义的实例构造

---

① “构造器”(constructor)也称为“构造函数”。相应地，“析构器”也称为“析构函数”。——译注

② 这些附加的字段称为 overhead fields，“overhead”是开销的意思，意味着是创建对象时必须的“开销”。——译注

器。由于永远不能继承实例构造器，所以实例构造器不能使用以下修饰符：virtual，new，override，sealed 和 abstract。如果类没有显式定义任何构造器，那么 C#编译器将定义一个默认(无参)构造器。在它的实现中，只是简单地调用了基类的无参构造器。

例如下面这个类：

```
public class SomeType {
}
```

它等价于：

```
public class SomeType {
    public SomeType() : base() { }
}
```

如果类的修饰符为abstract，那么编译器生成的默认构造器的可访问性就为protected[①]；否则，构造器会被赋予 public 可访问性。如果基类没有提供无参构造器，那么派生类必须显式调用一个基类构造器，否则编译器会报错。如果类的修饰符为 static(sealed 和 abstract)[②]，那么编译器根本不会在类的定义中生成默认构造器。

一个类型可以定义多个实例构造器。每个构造器都必须有不同的签名，而且每个都可以有不同的可访问性。为了使代码“可验证”(verifiable)，类的实例构造器在访问从基类继承的任何字段之前，必须先调用基类的构造器。如果派生类的构造器没有显式调用一个基类构造器，C#编译器会自动生成对默认的基类构造器的调用。最终，System.Object 的公共无参构造器会得到调用。该构造器什么都不做，会直接返回。由于 System.Object 没有定义实例数据字段，所以它的构造器无事可做。

极少数时候，可以在不调用实例构造器的前提下创建类型的实例。一个典型的例子是 Object 的 MemberwiseClone 方法。该方法的作用是分配内存，初始化对象的附加字段(类型对象指针和同步块索引)，然后将源对象的字节数据复制到新对象中。另外，用运行时序列化器(runtime serializer)反序列化对象时，通常也不需要调用构造器。反序列化代码使用System.Runtime.Serialization.FormatterServices类型的GetUninitializedObject或者GetSafeUninitializedObject方法为对象分配内存，期间不会调用一个构造器。详情参见第 24 章“运行时序列化”。

重要提示：在构造器中，不要调用任何可能影响所构造对象的虚方法。原因是假如被实例化的类型重写了虚方法，那么在调用基类的构造器时，实际执行的是派生类型的虚方法实现。但在这个时候，尚未完成对继承层次结构中所有字段的初始化(此时，派生类

---

① 这种类型的成员可由派生类型访问，不管这些派生类型是不是在同一个程序集中。——译注

② 静态类在元数据中是抽象密封类。——译注

型的构造器还没有运行)。所以，调用虚方法会导致无法预测的行为。[①]

C#语言用简单的语法在构造引用类型的实例时初始化类型中定义的字段：

```
internal sealed class SomeType {
    private Int32 m_x = 5;
}
```

构造 SomeType 的对象时，它的 m_x 字段被初始化为 5。这是如何发生的呢？检查一下 SomeType 的构造器方法(也称作.ctor)的 IL 代码就明白了，如下所示：

```
.method public hidebysig specialname rtspecialname
          instance void .ctor() cil managed
{
  // Code size 14 (0xe)
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: ldc.i4.5
  IL_0002: stfld   int32     SomeType::m_x
  IL_0007: ldarg.0
  IL_0008: call              instance void [mscorlib]System.Object::.ctor()
  IL_000d: ret
} // end of method SomeType::.ctor
```

可以看出，SomeType 的构造器是先将值 5 存储到字段 m_x，再调用基类的构造器。换句话说，C#编译器提供了一个简化的语法，允许以“内联”(其实就是嵌入)方式初始化实例字段。但在幕后，它会将这种语法转换成构造器方法中的代码来执行初始化。这同时提醒我们注意代码的膨胀效应。如以下类定义所示：

```
internal sealed class SomeType {
    private Int32      m_x = 5;
    private String     m_s = "Hi there";
    private Double     m_d = 3.14159;
    private Byte       m_b;

    // 下面是一些构造器
    public SomeType()           {...}
    public SomeType(Int32 x)    {...}
    public SomeType(String s)   {...; m_d=10;}
}
```

编译器为这三个构造器方法生成代码时，在每个方法的开始位置，都会包含用于初始化 m_x，m_s 和 m_d 的代码。在这些初始化代码之后，编译器会插入对基类构造器的调用。再然后，会插入构造器自己的代码。例如，对于获取一个 String 参数的构造器，编译器生

[①] 记住，创建派生类的实例时，首先会调用基类的构造函数，然后才会调用派生类的构造函数。这是一个自上而下的过程，从基类到派生类。——译注

成的代码首先初始化 m_x，m_s 和 m_d，再调用基类(Object)的构造器，再执行自己的代码(最后是用值 10 覆盖 m_d 原先的值)。注意，即使没有代码显式初始化 m_b，m_b 也保证会被初始化为 0。

注意：编译器在调用基类构造器前，会使用简化语法对所有字段进行初始化，以维持源代码给人留下的“这些字段总是有一个值”的印象。但是，假如基类构造器调用了虚方法并回调由派生类定义的方法，就可能出问题。在这种情况下，使用简化语法初始化的字段在调用虚方法之前就初始化好了。

由于有三个构造器，所以编译器将生成三次初始化 m_x，m_s 和 m_d 的代码——每个构造器一次。因此，如果有几个已初始化的实例字段和多个重载的构造器方法，那么可以考虑不要在定义字段的同时初始化。相反，专门创建一个构造器来执行这些公共的初始化。然后，让其他构造器都显式调用这个公共初始化构造器。这样能减少生成的代码。下例演示了如何在 C#中利用 this 关键字显式调用另一个构造器：

```
using System;
internal sealed class SomeType {
    // 不要显式初始化下面的字段
    private Int32      m_x;
    private String     m_s;
    private Double     m_d;
    private Byte       m_b;

    // 这个专门的构造器将所有字段都设为默认值，
    // 其他所有构造器都显式调用该构造器
    public SomeType() {
        m_x = 5;
        m_s = "Hi there";
        m_d = 3.14159;
        m_b = 0xff;
    }

    // 该构造器将所有字段都设为默认值，然后修改 m_x
    public SomeType(Int32 x) : this() {
        m_x = x;
    }

    // 该构造器将所有字段都设为默认值，然后修改 m_s
    public SomeType(String s) : this() {
        m_s = s;
    }

    // 该构造器将所有字段都设为默认值，然后修改 m_x 和 m_s
    public SomeType(Int32 x, String s) : this() {
        m_x = x;
        m_s = s;
    }
```

```
}
```

8.2 实例构造器和结构(值类型)

值类型(struct)构造器的工作方式与引用类型(class)的构造器截然不同。CLR 总是允许创建值类型的实例，并且没有办法阻止值类型的实例化。所以，值类型其实并不需要定义构造器，C#编译器根本不会为值类型内联(嵌入)默认的无参构造器。来看下面的代码：

```
internal struct Point {
    public Int32 m_x, m_y;
}
internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;
}
```

为了构造一个 Rectangle，必须使用 new 操作符，而且必须指定构造器。在这个例子中，调用的是C#编译器自动生成的默认构造器。为引用类型 Rectangle 的对象分配内存时，内存中将包含 Point 值类型的两个实例。考虑到性能，CLR 不会为包含在引用类型中的每个值类型字段都主动调用构造器。但是，如前所述，值类型的字段会被初始化为 0 或 null。

CLR 确实允许为值类型定义构造器。但必须显式调用才会执行。下面是一个例子。

```
internal struct Point {
    public Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }
}

internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;

    public Rectangle() {
        // 在 C#中，向一个值类型应用关键字 new，
        // 可以调用构造器来初始化值类型的字段
        m_topLeft  = new Point(1, 2);
        m_bottomRight= new Point(100, 200);
    }
}
```

值类型的实例构造器只有显式调用才会执行。因此，如果 Rectangle 的构造器没有使用 new 操作符来调用 Point 的构造器，从而初始化 Rectangle 的 m_topLeft 字段和 m_bottomRight 字段，那么两个 Point 字段中的 m_x 和 m_y 字段都将默认初始化为 0。

前面展示的 Point 值类型没有定义默认无参构造器。现在进行如下改写：

```
internal struct Point {
    public Int32 m_x, m_y;

    public Point() {
        m_x = m_y = 5;
    }
}

internal sealed class Rectangle {
    public Point m_topLeft, m_bottomRight;

    public Rectangle() {
    }
}
```

现在，构造一个新的 Rectangle 对象时，两个 Point 字段中的 m_x 和 m_y 字段会被初始化成多少？是 0 还是 5？(提示：小心上当！)

许多开发人员(尤其是那些有 C++背景的)都觉得 C#编译器会在 Rectangle 的构造器中生成代码，为 Rectangle 的两个字段自动调用 Point 的默认无参构造器。但是，为了增强应用程序的运行时性能，C#编译器不会自动生成这样的代码。实际上，即便值类型提供了无参构造器，许多编译器也永远不会生成代码来自动调用它。为了执行值类型的无参构造器，开发人员必须增加显式调用值类型构造器的代码。

基于上面这一段的描述，你可能会觉得，对于上述示例代码，Rectangle 类的两个 Point 字段的 m_x 和 m_y 字段都会初始化为 0，因为代码没有在任何地方显式调用了 Point 的构造器。

但我说过，这是一个容易让人上当的问题。这里的关键在于，C#编译器根本不允许值类型定义无参构造器。所以，上述代码实际是编译不了的。试图编译上述代码时，C#编译器会显示以下消息：error CS0568: 结构不能包含显式的无参数构造器。

C#编译器故意不允许值类型定义无参构造器，目的是防止开发人员对这种构造器在什么时候调用产生迷惑。由于不能定义无参构造器，所以编译器永远不会生成自动调用它的代码。没有无参构造器，值类型的字段总是被初始化为 0 或 null。

注意：严格地说，只有当值类型的字段嵌套到引用类型中时，才保证被初始化为 0 或 null。基于栈的值类型字段则无此保证。但是，为了确保代码的“可验证性”(verifiability)，任何基于栈的值类型字段都必须在读取之前写入(赋值)。如果允许代码先读取值类型的一个字段，再向其写入，那么会造成安全漏洞。对于基于栈的值类型中的所有字段，C#和其他能生成“可验证”代码的编译器可以保证对它们进行“置零”，或至少保证在读取之前赋值，确保不会在运行时因验证失败而抛出异常。所以，你完全可以忽略本“注意”的内容，假定自己的值类型的字段都会被初始化为 0 或 null。

注意，虽然 C#不允许值类型带有无参构造器，但 CLR 允许。所以，如果不在乎前面描述

的问题，可以使用另一编程语言(比如 IL 汇编语言)定义带有无参构造器的值类型。

由于 C#不允许为值类型定义无参构造器，所以编译以下类型时，C#编译器将显示消息：`error CS0573: "SomeValType.m_x" :` 结构中不能有实例字段初始值设定项。

```
internal struct SomeValType {
    // 不能在值类型中内联实例字段的初始化
    private Int32 m_x = 5;
}
```

另外，为了生成“可验证”代码，在访问值类型的任何字段之前，都需要完全对全部字段的赋值。所以，值类型的任何构造器都必须初始化值类型的全部字段。以下类型为值类型定义了一个构造器，但没有初始化值类型的全部字段：

```
internal struct SomeValType {
    private Int32 m_x, m_y;

    // C#允许为值类型定义有参构造器
    public SomeValType(Int32 x) {
        m_x = x;
        // 注意 m_y 没有在这里初始化
    }
}
```

编译上述类型，C#编译器会显示消息：

`error CS0171:` 在控制返回到调用方之前，字段`"SomeValType.m_y"`必须完全赋值。

为了修正这个问题，需要在构造器中为 `m_y` 赋一个值(通常是 `0`)。下面是对值类型的全部字段进行赋值的一个替代方案：

```
// C#允许为值类型定义有参构造器
public SomeValType(Int32 x) {
   // 看起来很奇怪，但编译没问题，会将所有字段初始化为 0/null
   this = new SomeValType();

   m_x = x; // 用 x 覆盖 m_x 的 0
   // 注意此时 m_y 已自动初始化为 0
}
```

在值类型的构造器中，`this` 代表值类型本身的一个实例，用 `new` 创建的值类型的一个实例可以赋给 `this`。在 `new` 的过程中，会将所有字段置为零。而在引用类型的构造器中，`this` 被认为是只读的，所以不能对它进行赋值。

8.3 类型构造器

除了实例构造器，CLR 还支持类型构造器(type constructor)，也称为静态构造器(static constructor)、类构造器(class constructor)或者类型初始化器(type initializer)。类型构造器可

应用于接口(虽然 C#编译器不允许)、引用类型和值类型。实例构造器的作用是设置类型的实例的初始状态。对应地，类型构造器的作用是设置类型的初始状态。类型默认没有定义类型构造器。如果定义，也只能定义一个。此外，类型构造器永远没有参数。以下代码演示了如何在 C#中为引用类型和值类型定义一个类型构造器：

```
internal sealed class SomeRefType {
    static SomeRefType() {
        // SomeRefType 被首次访问时，执行这里的代码
    }
}

internal struct SomeValType{
    // C#允许值类型定义无参的类型构造器
    static SomeValType() {
        // SomeValType 被首次访问时，执行这里的代码
    }
}
```

可以看出，定义类型构造器类似于定义无参实例构造器，区别在于必须标记为static。此外，类型构造器总是私有；C#自动把它们标记为 private。事实上，如果在源代码中显式将类型构造器标记为 private(或其他访问修饰符)，C#编译器会显示以下消息：error CS0515: 静态构造函数中不允许出现访问修饰符。之所以必须私有，是为了防止任何由开发人员写的代码调用它，对它的调用总是由 CLR 负责。

重要提示：虽然能在值类型中定义类型构造器，但永远都不要真的那么做，因为 CLR 有时不会调用值类型的静态类型构造器。下面是一个例子：

```
internal struct SomeValType {
    static SomeValType() {
        Console.WriteLine("这句话永远不会显示");
    }
    public Int32 m_x;
}

public sealed class Program {
    public static void Main() {
        SomeValType[] a = new SomeValType[10];
        a[0].m_x = 123;
        Console.WriteLine(a[0].m_x); // 显示 123
    }
}
```

类型构造器的调用比较麻烦。JIT 编译器在编译一个方法时，会查看代码中都引用了哪些类型。任何一个类型定义了类型构造器，JIT 编译器都会检查针对当前 AppDomain，是否已经执行了这个类型构造器。如果构造器从未执行，JIT 编译器会在它生成的本机(native)代码中添加对类型构造器的调用。如果类型构造器已经执行，JIT 编译器就不添加对它的调用，因为它知道类型已经初始化好了。

现在，当方法被 JIT 编译完毕之后，线程开始执行它，最终会执行到调用类型构造器的代码。事实上，多个线程可能同时执行相同的方法。CLR 希望确保在每个 AppDomain 中，一个类型构造器只执行一次。为了保证这一点，在调用类型构造器时，调用线程要获取一个互斥线程同步锁。这样一来，如果多个线程试图同时调用某个类型的静态构造器，只有一个线程才可以获得锁，其他线程会被阻塞(blocked)。第一个线程会执行静态构造器中的代码。当第一个线程离开构造器后，正在等待的线程将被唤醒，然后发现构造器的代码已被执行过。因此，这些线程不会再次执行那些代码，将直接从构造器方法返回。除此之外，如果再次调用这样的一个方法，CLR 知道类型构造器已被执行过，从而确保构造器不被再次调用。

注意：由于 CLR 保证一个类型构造器在每个 AppDomain 中只执行一次，而且(这种执行)是线程安全的，所以非常适合在类型构造器中初始化类型需要的任何单实例(Singleton)对象。[①]

单个线程中的两个类型构造器包含相互引用的代码可能出问题。例如，假定 `ClassA` 的类型构造器包含了引用 `ClassB` 的代码，`ClassB` 的类型构造器包含了引用 `ClassA` 的代码。在这种情况下，CLR 仍然保证每个类型构造器的代码只被执行一次；但是，完全有可能在 `ClassA` 的类型构造器还没有执行完毕的前提下，就开始执行 `ClassB` 的类型构造器。因此，应尽量避免写会造成这种情况的代码。事实上，由于是 CLR 负责类型构造器的调用，所以任何代码都不应要求以一个特定的顺序调用类型构造器。

最后，如果类型构造器抛出未处理的异常，CLR 会认为类型不可用。试图访问该类型的任何字段或方法都会抛出 `System.TypeInitializationException` 异常。

类型构造器中的代码只能访问类型的静态字段，并且它的常规用途就是初始化这些字段。和实例字段一样，C#提供了一个简单的语法来初始化类型的静态字段：

```
internal sealed class SomeType {
   private static Int32 s_x = 5;
}
```

注意：虽然 C#不允许值类型为它的*实例*字段使用内联字段初始化语法，但可以为*静态*字段使用。换句话说，如果将前面定义的 `SomeType` 类型从 `class` 改为 `struct`，那么代码能通过编译，而且会像你预期的那样工作。

---

① 在每个 AppDomain 中只能有一个实例，这样的类型就是单实例类型。——译注

生成上述代码时，编译器自动为 SomeType 生成一个类型构造器，好像源代码原本是这样的：

```
internal sealed class SomeType {
    private static Int32 s_x;
    static SomeType() { s_x = 5; }
}
```

使用 ILDasm.exe 查看类型构造器的 IL，很容易验证编译器实际生成的东西。类型构造器方法总是叫.cctor(代表 class constructor)。

在以下代码中，可以看到.cctor 方法是 private 和 static 的。另外，注意方法中的代码确实将值 5 加载到静态字段 s_x 中。

```
.method private hidebysig specialname rtspecialname static
          void .cctor() cil managed
{
  // Code size 7 (0x7)
  .maxstack 8
  IL_0000: ldc.i4.5
  IL_0001: stsfld  int32 SomeType::s_x
  IL_0006: ret
} // end of method SomeType::.cctor
```

类型构造器不应调用基类型的类型构造器。这种调用之所以没必要，是因为类型不可能有静态字段是从基类型分享或继承的。

注意：有的语言(比如 Java)希望在访问类型时自动调用该类型的类型构造器，并调用所有基类型的类型构造器。此外，类型所实现的接口也必须调用接口的类型构造器。[①] CLR 不支持这种行为。但是，使用由 System.Runtime.CompilerServices.RuntimeHelpers 提供的 RunClassConstructor 方法，编译器和开发人员可以实现这种行为。任何语言如果想实现这种行为，可以告诉它的编译器在一个类型的类型构造器中生成代码来调用所有基类型的这个方法。用 RunClassConstructor 方法调用一个类型构造器时，CLR 知道类型构造器之前是否执行过。如果是，CLR 不会再次调用它。

最后，假定有以下代码：

```
internal sealed class SomeType {
    private static Int32 s_x = 5;

    static SomeType() {
       s_x = 10;
```

[①] 在 CLR 的眼中，接口即类型。详情请参见第 13 章“接口”。——译注

```
        }
    }
```

在这个例子中，C#编译器会生成单一的类型构造器方法。它首先将 `s_x` 初始化为 `5`，再把它修改成 `10`。换言之，当 C#编译器为类型构造器生成 IL 代码时，它首先生成的是初始化静态字段所需的代码，然后才会添加你的类型构造器方法中显式包含的代码。

重要提示：偶尔有开发人员问我，是否可以在卸载类型时执行一些代码。首先要搞清楚的是，类型只有在 AppDomain 卸载时才会卸载。AppDomain 卸载时，用于标识类型的对象(类型对象)将成为“不可达”的对象(不再有对它的引用)，垃圾回收器会回收类型对象的内存。这个行为导致许多开发人员以为可以为类型添加一个静态 `Finalize` 方法。当类型卸载时，就自动地调用这个方法。遗憾的是，CLR 并不支持静态 `Finalize` 方法。但是，也并非完全没有办法。要在 AppDomain 卸载时执行一些代码，可以向 `System.AppDomain` 类型的 `DomainUnload` 事件登记一个回调方法。

8.4 操作符重载方法

有的语言允许类型定义操作符应该如何操作类型的实例。例如，许多类型(比如 `System.String`)都重载了相等(`==`)和不等(`!=`)操作符。CLR 对操作符重载一无所知，它甚至不知道什么是操作符。是编程语言定义了每个操作符的含义，以及当这些特殊符号出现时，应该生成什么样的代码。

例如在 C#中，向基元(类型的)数字应用+符号，编译器会生成将两个数加到一起的代码。将+符号应用于 `String` 对象，C#编译器则生成将两个字符串连接(拼接)到一起的代码。测试不等性时，C#使用的是`!=`符号，Microsoft Visual Basic 用的则是`<>`。最后，`^`在 C#中的含义为异或(XOR)，在 Visual Basic 中则为求幂。

虽然 CLR 对操作符一无所知，但它确实规定了语言应如何公开操作符重载，以便由另一种语言的代码使用。每种编程语言都要自行决定是否支持操作符重载。如果决定支持，还要决定用什么语法来表示和使用它们。至于 CLR，操作符重载只是方法而已。

对编程语言的选择决定了你是否获得对操作符重载的支持，以及具体的语法是什么。编译源代码时，编译器会生成一个标识操作符行为的方法。CLR 规范要求操作符重载方法必须是 `public` 和 `static` 方法。另外，C# (以及其他许多语言)要求操作符重载方法至少有一个参数的类型与当前定义这个方法的类型相同。之所以要进行这样的限制，是为了使 C#编译器能在合理的时间内找到要绑定的操作符方法。

以下 C#代码展示了在一个类中定义的操作符重载方法：[①]

```
public sealed class Complex {
    public static Complex operator+(Complex c1, Complex c2) { ... }
}
```

编译器为名为 op_Addition 的方法生成元数据方法定义项；这个方法定义项还设置了 specialname 标志，表明这是一个“特殊”方法。编程语言的编译器(包括 C#编译器)看到源代码中出现一个+操作符时，会检查是否有一个操作数的类型定义了名为 op_Addition 的 specialname 方法，而且该方法的参数兼容于操作数的类型。如果存在这样的方法，编译器就生成调用它的代码。不存在这样的方法就报告编译错误。

表 8-1 和表 8-2 总结了 C#允许重载的一元和二元操作符，以及由编译器生成的对应的 CLS(Common Language Specification，公共语言规范)方法名。下一节会解释这些表的第三列。

表 8-1 C#的一元操作符及其相容于 CLS 的方法名

C#操作符	特殊方法名	推荐的相容于 CLS 的方法名
+	op_UnaryPlus	Plus
-	op_UnaryNegation	Negate
!	op_LogicalNot	Not
~	op_OnesComplement	OnesComplement
++	op_Increment	Increment
--	op_Decrement	Decrement
(无)	op_True	IsTrue { get; }
(无)	op_False	IsFalse { get; }

表 8-2 C#的二元操作符及其相容于 CLS 的方法名

C#操作符	特殊方法名	推荐的相容于 CLS 的方法名
+	op_Addition	Add
-	op_Subtraction	Subtract
*	op_Multiply	Multiply
/	op_Division	Divide

[①] Complex 建模的是“复数”类。形如 $x + yi$(x 和 y 均为实数)的数称为复数。其中，x 称为实部，y 称为虚部，i 为虚数单位(-1 的平方根)。——译注

%	op_Modulus	Mod
&	op_BitwiseAnd	BitwiseAnd
|	op_BitwiseOr	BitwiseOr
^	op_ExclusiveOr	Xor
<<	op_LeftShift	LeftShift
>>	op_RightShift	RightShift
==	op_Equality	Equals
!=	op_Inequality	Equals
<	op_LessThan	Compare
>	op_GreaterThan	Compare
<=	op_LessThanOrEqual	Compare
>=	op_GreaterThanOrEqual	Compare

CLR 规范定义了许多额外的可重载的操作符，但 C#不支持这些额外的操作符。由于是非主流，所以此处不列出它们。如果对完整列表感兴趣，请访问 CLI 的 ECMA 规范(*https://tinyurl.com/4nykrnmt*)，下载 PDF，并阅读 Partition I：Concepts and Architecture 的 I.10.3.1 节(一元操作符)和 I.10.3.2 节(二元操作符)。

注意：检查 Framework 类库(FCL)的核心数值类型(Int32，Int64 和 UInt32 等)，会发现它们没有定义任何操作符重载方法。之所以不定义，是因为编译器会(在代码中)专门查找针对这些基元类型执行的操作(运算)，并生成 IL 指令来直接操作这些类型的实例。如果类型要提供方法，而且编译器要生成代码来调用这些方法，方法调用就会产生额外的运行时开销。而且，方法最后无论如何都要执行一些 IL 指令来完成你希望的操作。这正是核心 FCL 类型没有定义任何操作符重载方法的原因。对于开发人员，这意味着假如选择的编程语言不支持其中的某个 FCL 类型，便不能对该类型的实例执行任何操作。

操作符和编程语言互操作性

操作符重载是很有用的工具，允许开发人员用简洁的代码表达自己的想法。但并不是所有编程语言都支持操作符重载。使用不支持操作符重载的语言时，语言不知道如何解释+操作符(除非类型是该语言的基元类型)，编译器会报错。使用不支持操作符重载的编程语言时，语言应该允许你直接调用希望的 op_*方法(例如 op_Addition)。

即使语言不支持在一个类型中定义+操作符重载，该类型仍有可能提供了一个 op_Addition 方法。在这种情况下，可不可以在 C#中使用+操作符来调用这个 op_Addition 方法呢？答案是否定的。C#编译器检测到+操作符时，会查找关联了 specialname 元数据标志的 op_Addition 方法，以确定 op_Addition 方法是要作为操作符重载方法使用。但是，由于现在这个 op_Addition 方法是由不支持操作符重载的编程语言生成的，所以方法没有关联

specialname 标记。因此，C#编译器会报告编译错误。当然，用任何编程语言写的代码都可以显式调用碰巧命名为 op_Addition 的方法，但编译器不会将一个+号的使用翻译成对这个方法的调用。

注意：FCL 的 System.Decimal 类型(实际是结构)很好地演示了如何重载操作符并根据 Microsoft 设计规范使用友好的方法名(*https://tinyurl.com/3xeszzx4*)。

Microsoft 操作符方法的命名规则之我见

操作符重载方法什么时候能调用，什么时候不能调用，这些规则会令人感觉非常困惑。如果支持操作符重载的编译器不生成 specialname 元数据标记，规则会简单得多，而且开发人员使用提供了操作符重载方法的类型也会更轻松。在这种情况下，支持操作符重载的语言都将支持操作符符号语法，而且所有语言都将支持显式调用各种 op_方法。我不理解为什么微软非要把它搞得这么复杂。希望微软在编译器未来的版本中放宽这些限制。

如果一个类型定义了操作符重载方法，微软还建议类型定义更友好的公共静态(public static)方法，并在这种方法的内部调用操作符重载方法。例如，根据微软的设计规范，重载了 op_Addition 方法的类型应定义一个公共的、名字更友好的 Add 方法。表 8-1 和表 8-2 的第三列展示了每个操作符推荐使用的友好名称。因此，前面的示例类型 Complex 应该像下面这样定义：

```
public sealed class Complex {
    public static Complex operator+(Complex c1, Complex c2) { ... }
    public static Complex Add(Complex c1, Complex c2) { return(c1 + c2); }
}
```

用任何语言写的代码都肯定能调用 Add 这样的友好操作符方法。但是，微软建议为类型提供友好方法名的设计规范使局面进一步复杂化。在我看来，这种额外的复杂性完全没必要。而且，除非 JIT 编译器能内联(直接嵌入)友好方法的代码，否则调用它们将导致额外的性能损失。内联代码使 JIT 编译器能够优化代码，移除额外的方法调用，并提升运行时性能。

8.5 转换操作符方法

有的时候，我们需要将对象从一种类型转换为另一种类型(例如将 Byte 转换为 Int32)。当源类型和目标类型都是编译器识别的基元类型时，编译器自己就知道如何生成转换对象所需的代码。

如果源类型或目标类型不是基元类型，编译器会生成代码，要求 CLR 执行转换(强制转型)。这种情况下，CLR 只是检查源对象的类型和目标类型(或者从目标类型派生的其他类型)是

不是相同。但是，有时需要将对象从一种类型转换成全然不同的其他类型。例如，System.Xml.Linq.XElement 类允许将 XML 元素转换成 Boolean，(U)Int32，(U)Int64，Single，Double，Decimal，String，DateTime，DateTimeOffset，TimeSpan，Guid 或者所有这些类型(String 除外)的可空版本。另外，假设 FCL 包含了一个 Rational(有理数)类型，那么如果能将 Int32 或 Single 转换成 Rational，那么会显得很方便；反之亦然。

为了进行这样的转换，Rational 类型应该定义只有一个参数的公共构造器，该参数要求是源类型的实例。还应该定义无参的公共实例方法 ToXxx(类似于你熟悉的 ToString 方法)，每个方法都将定义类型[1]的实例转换成 Xxx 类型。以下代码展示了如何为 Rational 类型正确定义转换构造器和方法。

```
public sealed class Rational {
     // 由一个 Int32 构造一个 Rational
     public Rational(Int32 num){ ... }

     // 由一个 Single 构造一个 Rational
     public Rational(Single num){ ... }

     // 将一个 Rational 转换成一个 Int32
     public Int32 ToInt32(){ ... }

     // 将一个 Rational 转换成一个 Single
     public Single ToSingle(){ ... }
}
```

调用这些构造器和方法，使用任何编程语言的开发人员都能将 Int32 或 Single 对象转换成 Rational 对象，反之亦然。这些转换能给编程带来很多方便。设计类型时，应认真考虑类型需要支持的转换构造器和方法。

上一节讨论了某些编程语言如何提供操作符重载。事实上，有些编程语言(比如 C#)还提供了转换操作符重载。**转换操作符**是将对象从一种类型转换成另一种类型的方法。可以使用特殊的语法来定义转换操作符方法。CLR 规范要求转换操作符重载方法必须是 public 和 static 方法。此外，C#(以及其他许多语言)要求参数类型和返回类型二者必有其一与定义转换方法的类型相同。之所以要进行这个限制，是为了使 C#编译器能在一个合理的时间内找到要绑定的操作符方法。以下代码为 Rational 类型添加了 4 个转换操作符方法：

```
public sealed class Rational {
    // 由一个 Int32 构造一个 Rational
    public Rational(Int32 num){ ... }

    // 由一个 Single 构造一个 Rational
     public Rational(Single num){ ... }
```

[1] 定义该方法的类型。——译注

```
    // 将一个 Rational 转换成一个 Int32
     public Int32 ToInt32(){ ... }

    // 将一个 Rational 转换成一个 Single
     public Single ToSingle(){ ... }

    // 由一个 Int32 隐式构造并返回一个 Rational
    public static implicit operator Rational(Int32 num) {
        return new Rational(num);
    }

    // 由一个 Single 隐式构造并返回一个 Rational
    public static implicit operator Rational(Single num) {
        return new Rational(num);
    }

    // 由一个 Rational 显式返回一个 Int32
    public static explicit operator Int32(Rational r) {
        return r.ToInt32();
    }

    // 由一个 Rational 显式返回一个 Single
    public static explicit operator Single(Rational r) {
        return r.ToSingle();
    }
}
```

对于转换操作符方法，编译器既可生成代码来隐式调用转换操作符方法，也可只有在源代码进行了显式转型时才生成代码来调用转换操作符方法。在 C#中，implicit 关键字告诉编译器为了生成代码来调用方法，不需要在源代码中进行显式转型。相反，explicit 关键字告诉编译器只有在发现了显式转型时，才调用方法。

在 implicit 或 explicit 关键字之后，要指定 operator 关键字告诉编译器该方法是一个转换操作符。在 operator 之后，指定对象要转换成什么类型。在圆括号内，则指定要从什么类型转换。

像前面那样为 Rational 类型定义了转换操作符之后，就可以写像下面这样的 C#代码：

```
public sealed class Program {
    public static void Main() {
        Rational r1 = 5;         // Int32 隐式转型为 Rational
        Rational r2 = 2.5F;      // Single 隐式转型为 Rational

        Int32 x = (Int32) r1;    // Rational 显式转型为 Int32
        Single s = (Single) r2;  // Rational 显式转型为 Single
    }
}
```

在幕后，C#编译器检测到代码中的转型，并内部生成 IL 代码来调用 Rational 类型定义的

转换操作符方法。现在的问题是，这些方法的名称是什么？编译 Rational 类型并查看元数据，会发现编译器为定义的每个转换操作符都生成了一个方法。Rational 类型的 4 个转换操作符方法的元数据如下：

```
public static Rational      op_Implicit(Int32 num)
public static Rational      op_Implicit(Single num)
public static Int32         op_Explicit(Rational r)
public static Single        op_Explicit(Rational r)
```

可以看出，将对象从一种类型转换成另一种类型的方法总是叫做 op_Implicit 或者 op_Explicit。只有在转换不损失精度或数量级的前提下(比如将一个 Int32 转换成 Rational)，才能定义隐式转换操作符。如果转换会造成精度或数量级的损失(比如将 Rational 转换成 Int32)，就应该定义一个显式转换操作符。显式转换失败，应该让显式转换操作符方法抛出 OverflowException 或者 InvalidOperationException 异常。

注意：两个 op_Explicit 方法获取相同的参数，也就是一个 Rational。但两个方法的返回类型不同，一个是 Int32，另一个是 Single。这是仅凭返回类型来区分两个方法的例子。CLR 允许在一个类型中定义仅返回类型不同的多个方法。但是，只有极少数语言支持这个能力。你可能已经注意到了，C++，C#，Visual Basic 和 Java 语言都不允许在一个类型中定义仅返回类型不同的多个方法。个别语言(比如 IL 汇编语言)允许开发人员显式选择调用其中哪一个方法。当然，IL 汇编语言的程序员不应利用这个能力，否则定义的方法无法从其他语言中调用。虽然 C#语言没有向 C#程序员公开这个能力，但当一个类型定义了转换操作符方法时，C#编译器会在内部利用这个能力。

C#编译器提供了对转换操作符的完全支持。如果检测到代码中正在使用某个类型的对象，但实际期望的是另一种类型的对象，编译器就会查找能执行这种转换的隐式转换操作符方法，并生成代码来调用该方法。如果存在隐式转换操作符方法，编译器会在结果 IL 代码中生成对它的调用。如果编译器看到源代码是将对象从一种类型显式转换为另一种类型，就会查找能执行这种转换的隐式或显式转换操作符方法。如果找到一个，编译器就生成 IL 代码来调用它。如果没有找到合适的转换操作符方法，就报错并停止编译。

注意：使用强制类型转换表达式时，C#生成代码来调用显式转换操作符方法。使用 C#的 `as` 或 `is` 操作符时，则永远不会调用这些方法(参见 4.2 节)。

为了真正理解操作符重载方法和转换操作符方法，强烈建议将 `System.Decimal` 类型作为典型来研究。`Decimal` 定义了几个构造器，允许将对象从各种类型转换为 `Decimal`。还定义了几个 `ToXxx` 方法，允许将 `Decimal` 转换成其他类型。最后，`Decimal` 类型还定义了几个转换操作符方法和操作符重载方法。

8.6 扩展方法

理解 C#**扩展方法**最好的办法就是从例子中学习。14.3.2 节“`StringBuilder` 的成员”会提到，`StringBuilder` 类提供的字符串处理方法比 `String` 类少。这其实是很奇怪的一件事情：由于 `StringBuilder` 类是可变的(mutable)，所以它才应该是进行字符串处理的首选方式。现在，假定你想自己定义一些缺失的方法以更方便地操作 `StringBuilder`。例如，你也许想定义以下 `IndexOf` 方法：

```
public static class StringBuilderExtensions {
    public static Int32 IndexOf(StringBuilder sb, Char value) {
        for (Int32 index = 0; index < sb.Length; index++)
            if (sb[index] == value) return index;
        return -1;
    }
}
```

定义好这个方法后，可以在代码中使用它，如下所示：

```
StringBuilder sb=new StringBuilder("Hello. My name is Jeff."); //初始字符串

// 先将句点更改为感叹号，再获取!字符的索引(5)
Int32 index = StringBuilderExtensions.IndexOf(sb.Replace('.', '!'), '!');
```

上述代码工作起来没问题，但从程序员的角度看不理想。第一个问题是，要获取一个 `StringBuilder` 中的某个字符的索引，必须先知道 `StringBuilderExtensions` 类的存在。第二个问题是，代码没有反映出在 `StringBuilder` 对象上执行的操作的顺序，使代码很难写、读和维护。程序员希望先调用 `Replace`，再调用 `IndexOf`。但最后一行代码从左向右

读，先看到的是 IndexOf，然后才看到 Replace。当然，可以像下面这样重写，使代码的行为看起来更容易理解：

```
// 首先，将句点更改成感叹号
sb.Replace('.', '!');

// 然后，获取!字符的索引(5)
Int32 index = StringBuilderExtensions.IndexOf(sb, '!');
```

但这两个版本都存在另一个不容忽视的问题，它影响了我们对代码行为的理解。使用 StringBuilderExtensions 显得“小题大做”，造成程序员无法专注于当前要执行的操作：IndexOf。如果 StringBuilder 类定义了自己的 IndexOf 方法，上述代码就可以重写为：

```
// 把句点更改为感叹号，获取!字符的索引(5)
Int32 index = sb.Replace('.', '!').IndexOf('!');
```

哇，是不是立刻就显得高端大气上档次了？一眼就能看出在 StringBuilder 对象中，是先将句点更改为感叹号，再获取感叹号的索引。

有了这个例子作为铺垫，就很容易理解 C#扩展方法的意义了。它允许定义一个静态方法，并以实例方法的语法来调用。换言之，现在既能定义自己的 IndexOf 方法，又能避免上述三个问题。要将 IndexOf 方法转变成扩展方法，只需在第一个参数前添加 this 关键字：

```
public static class StringBuilderExtensions {
    public static Int32 IndexOf(this StringBuilder sb, Char value) {
        for (Int32 index = 0; index < sb.Length; index++)
            if (sb[index] == value) return index;
        return -1;
    }
}
```

现在，当编译器看到以下代码：

```
Int32 index = sb.IndexOf('X');
```

就会首先检查 StringBuilder 类或者它的任何基类是否提供了获取单个 Char 参数、名为 IndexOf 的一个实例方法。如果是，就生成 IL 代码来调用它。如果没有找到匹配的实例方法，就继续检查是否有任何静态类定义了名为 IndexOf 的静态方法，方法的第一个参数的类型和当前用于调用方法的那个表达式的类型匹配，而且该类型必须用 this 关键字标识。在本例中，表达式是 sb，类型是 StringBuilder。所以编译器会查找一个 IndexOf 方法，它有两个参数：一个 StringBuilder(用 this 关键字标记)，以及一个 Char。编译器找到了这个 IndexOf 方法，所以生成相应的 IL 代码来调用这个静态方法。

OK——这解释了编译器如何解决前面提到的、会影响代码理解的最后两个问题。但是，还没有说第一个问题是如何解决的：程序员怎么知道有这样的一个 IndexOf 方法，可以用它操作 StringBuilder 对象呢？这个问题是通过 Microsoft Visual Studio 的“智能感知”功能来解决的。在编辑器中输入句点符号，会弹出 Visual Studio 的“智能感知”窗口，列出

当前可用的实例方法。现在，这个窗口还会列出可作用于句点左侧表达式类型的扩展方法。图 8-1 展示了 Visual Studio 的“智能感知”窗口；扩展方法的图标中有一个下箭头，方法旁边的“工具提示”表明该方法实际是一个扩展方法。这是相当实用的一个功能，因为现在可以轻松定义自己的方法来操作各种类型，其他程序员在使用这些类型的对象时，也能轻松地发现你的方法。

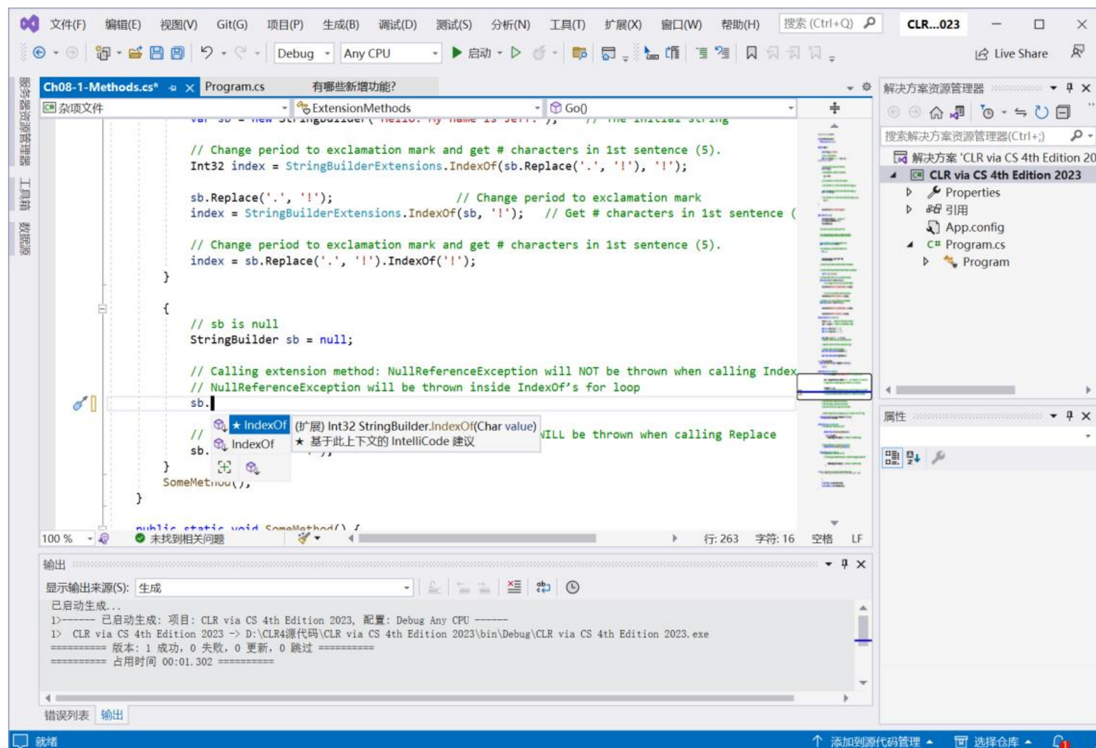


图 8-1 Visual Studio 的“智能感知”窗口能列出扩展方法

8.6.1 规则和指导原则

关于扩展方法，有一些额外的规则和指导原则需要注意。

- C#只支持扩展方法，不支持扩展属性、扩展事件、扩展操作符等。
- 扩展方法(第一个参数前面有 `this` 的方法)必须在非泛型的静态类中声明。然而，用于容纳扩展方法的类名没有限制，可以随便叫什么名字。当然，扩展方法至少要有一个参数，而且只有第一个参数能用 `this` 关键字标记。
- C#编译器在静态类中查找扩展方法时，要求静态类本身必须具有文件作用域[①]。如果静态类嵌套在另一个类中，C#编译器将显示以下消息：`error CS1109:` 扩展方法必须

---

① 类要具有整个文件的作用域，而不能嵌套在某个类中而只具有该类的作用域。——译注

在顶级静态类中定义；StringBuilderExtensions 是嵌套类。

- 由于静态类可以取任何名字，所以 C#编译器要花一定时间来寻找扩展方法，它必须检查文件作用域中的所有静态类，并扫描它们的所有静态方法来查找一个匹配。为增强性能，并避免找到非你所愿的扩展方法，C#编译器要求“导入”扩展方法。例如，如果有人在 Wintellect 命名空间中定义了一个 StringBuilderExtensions 类，那么程序员为了访问这个类的扩展方法，必须在其源代码文件顶部写一条 using Wintellect; 指令。

- 多个静态类可以定义相同的扩展方法。如果编译器检测到存在两个或多个扩展方法，就会显示以下消息：error CS0121: 在以下方法或属性之间的调用不明确："StringBuilderExtensions.IndexOf(string, char)" 和 "AnotherStringBuilderExtensions.IndexOf(string, char)"。修正这个错误必须修改源代码。具体地说，不能再用实例方法语法来调用这个静态方法。相反，必须使用静态方法语法。换言之，必须显式指定静态类的名称，明确告诉编译器要调用哪个方法。

- 使用这个功能须谨慎，一个原因是并非所有程序员都熟悉它。例如，用一个扩展方法扩展一个类型时，同时也扩展了派生类型。所以，不要将 System.Object 作为扩展方法的第一个参数，否则这个方法在所有表达式类型上都能调用，造成 Visual Studio 的“智能感知”窗口被填充太多垃圾信息。

- 扩展方法可能存在版本控制问题。如果 Microsoft 未来为他们的 StringBuilder 类添加了 IndexOf 实例方法，而且和我的代码调用的原型一样，那么在重新编译我的代码时，编译器会绑定到 Microsoft 的 IndexOf 实例方法，而不是我的静态 IndexOf 方法。这样我的程序就会有不同的行为。版本控制问题是使用扩展方法须谨慎的另一个原因。

8.6.2 用扩展方法扩展各种类型

前面演示了如何为 StringBuilder 类定义扩展方法。我要指出的一个问题是，由于扩展方法实际是对一个静态方法的调用，所以 CLR 不会生成代码对调用方法的表达式的值进行 null 值检查(不保证它非空)：

```
// sb 为 null
StringBuilder sb = null;

// 调用扩展方法： NullReferenceException 异常不会在调用 IndexOf 时抛出，
// 相反，NullReferenceException 是在 IndexOf 内部的 for 循环中抛出的
sb.IndexOf('X');

// 调用实例方法：NullReferenceException 异常会在调用 Replace 时抛出
sb.Replace('.', '!');
```

还要注意，可以为接口类型定义扩展方法，如下所示：

```
public static void ShowItems<T>(this IEnumerable<T> collection) {
    foreach (var item in collection)
        Console.WriteLine(item);
}
```

任何表达式，只要它最终的类型实现了 IEnumerable<T>接口，就能调用上述扩展方法：

```
public static void Main() {
   // 每个 Char 在控制台上单独显示一行
   "Grant".ShowItems();

   // 每个 String 在控制台上单独显示一行
   new[] { "Jeff", "Kristin" }.ShowItems();

   // 每个 Int32 在控制台上单独显示一行
   new List<Int32>() { 1, 2, 3 }.ShowItems();
}
```

重要提示：扩展方法是微软的 LINQ(Language Integrated Query，语言集成查询)技术的基础。要想仔细研究提供了大量扩展方法的一个典型的类，请自行在文档中查看静态类 System.Linq.Enumerable 及其所有静态扩展方法。这个类中的每个扩展方法都扩展了 IEnumerable 或 IEnumerable<T>接口。

还可以为委托类型定义扩展方法，如下所示：

```
public static void InvokeAndCatch<TException>(this Action<Object> d, Object o)
   where TException : Exception {
   try { d(o); }
   catch (TException) { }
}
```

下面演示了如何调用它[①]：

```
Action<Object> action = o =>
            Console.WriteLine(o.GetType()); // 抛出 NullReferenceException
action.InvokeAndCatch<NullReferenceException>(null); // 吞噬 NullReferenceException
```

还可为枚举类型添加扩展方法(15.3 节“向枚举类型添加方法”展示了一个例子)。

最后，C#编译器允许创建委托(参见第 17 章“委托”)来引用一个对象上的扩展方法：

```
public static void Main () {
    // 创建一个 Action 委托(实例)来引用静态 ShowItems 扩展方法，
    // 并初始化第一个实参来引用字符串"Jeff"
    Action a = "Jeff".ShowItems;
    ...
```

[①] 对“吞噬”异常的解释请参见 20.8.2 节。——译注

```
        // 调用(Invoke)委托，后者调用(call)① ShowItems，
        // 并向它传递对字符串"Jeff"的引用
        a();
    }
```

在上述代码中，C#编译器生成 IL 代码来构造一个 Action 委托。创建委托时，会向构造器传递应调用的方法，同时传递一个对象引用，这个引用应传给方法的隐藏 this 参数。正常情况下，创建引用静态方法的委托时，对象引用是 null，因为静态方法没有 this 参数。但在这个例子中，C#编译器生成特殊代码创建一个委托来引用静态方法(ShowItems)，而静态方法的目标对象是对"Jeff"字符串的引用。稍后，当这个委托被调用(invoke)时，CLR 会调用(call)静态方法，并向其传递对"Jeff"字符串的引用。这是编译器耍的小“花招”，但效果不错，而且只要你不去细想内部发生的事情，整个过程还是感觉非常自然的。

8.6.3 ExtensionAttribute 类

扩展方法的概念要不是 C#特有的就好了！具体地说，我们希望程序员能用一种编程语言定义一组扩展方法，并让其他语言的程序员利用它们。要实现这一点，选择的编译器必须能够搜索静态类型和方法来寻找匹配的扩展方法。另外，速度慢了还不行。编译器必须快速完成上述搜索，将编译时间控制在合理范围内。

在 C#中，一旦用 this 关键字标记了某个静态方法的第一个参数，编译器就会在内部向该方法应用一个定制特性。该特性会在最终生成的文件的元数据中持久性地存储下来。该特性在 System.Core.dll 程序集中定义，它看起来像下面这样：

```
// 在 System.Runtime.CompilerServices 命名空间中定义
[AttributeUsage(   AttributeTargets.Method | AttributeTargets.Class
                   AttributeTargets.Assembly)]
public sealed class ExtensionAttribute : Attribute {
}
```

除此之外，任何静态类只要包含至少一个扩展方法，它的元数据中也会应用这个特性。类似地，任何程序集只要包含了至少一个符合上述特点的静态类，它的元数据中也会应用这个特性。这样一来，如果代码调用了一个不存在的实例方法，编译器就能快速扫描引用的所有程序集，判断它们哪些包含了扩展方法。然后，在这些程序集中，可以只扫描包含了扩展方法的静态类。在每个这样的静态类中，可以只扫描扩展方法来查找匹配。利用这个

---

① “调用一个委托实例”中的“调用”对应的是 invoke，理解为“唤出”更恰当。它和后面的“在一个对象上调用方法”中的“调用”稍有不同，后者对应的是 call。在英语的语境中，invoke 和 call 的区别在于，在执行一个所有信息都已知的方法时，用 call 比较恰当。这些信息包括要引用的类型，方法的签名以及方法名。但是，在需要先“唤出”某个东西来帮你调用一个信息不明的方法时，用 invoke 就比较恰当。但是，由于两者均翻译为“调用”不会对读者的理解造成太大的困扰，所以本书仍然采用约定俗成的方式来进行翻译，只是在必要的时候附加英文原文提醒你区分。——译注

技术，代码能以最快速度编译完毕。

注意：ExtensionAttribute 类在 System.Core.dll 程序集中定义。这意味着，即使在编译代码时没有使用 System.Core.dll 中的任何类型，甚至没有引用 System.Core.dll，编译器生成的结果程序集仍然会嵌入对 System.Core.dll 的引用。但是，这并不是太大的问题，因为 ExtensionAttribute 仅在编译时使用；在运行时，除非应用程序使用了 System.Core.dll 程序集中的其他内容，否则不需要加载该程序集。

8.7 分部方法

假定用某个工具生成了包含类型定义的 C#源代码文件，该工具知道在生成的代码中可能存在你想要自定义类型行为的地方。正常情况下，是让工具生成的代码调用虚方法来进行定制。工具生成的代码还必须包含虚方法的定义。另外，这些方法的实现是什么事情都不做，直接返回了事。现在，如果想定制类的行为，你可以从基类派生并定义自己的类，并重写虚方法来实现自己想要的行为。下面是一个例子：

```
// 工具生成的代码，存储在某个源代码文件中：
internal class Base {
    private String m_name;

    // 在更改 m_name 字段前调用
    protected virtual void OnNameChanging(String value) {
    }

    public String Name {
        get { return m_name; }
        set {
           OnNameChanging(value.ToUpper());    // 告诉类要进行更改了
           m_name = value;                     // 更改字段
        }
    }
}

// 开发人员生成的代码，重写了虚方法，存储在另一个源代码文件中：
internal class Derived : Base {
    protected override void OnNameChanging(string value) {
        if (String.IsNullOrEmpty(value))
            throw new ArgumentNullException("value");
    }
}
```

遗憾的是，上述代码存在两个问题。

- 类型必须是非密封的类。这个技术不能用于密封类，也不能用于值类型(值类型隐式密封)。此外，这个技术不能用于静态方法，因为静态方法不能重写。

- 效率问题。定义一个类型只是为了重写一个方法，这会浪费少量系统资源。另外，即使不想重写 OnNameChanging 的行为，基类代码仍需调用一个什么都不做、直接就返回的虚方法。另外，无论 OnNameChanging 是否访问传给它的实参，编译器都会生成对 ToUpper 进行调用的 IL 代码。

利用 C#的**分部方法**功能，可以在解决上述问题的同时重写(override)类的行为。以下代码使用分部方法实现和上述代码完全一样的语义：

```
// 工具生成的代码，存储在某个源代码文件中：
internal sealed partial class Base {
    private String m_name;

    // 这是分部方法的声明
    partial void OnNameChanging(String value);

    public String Name {
        get { return m_name; }
        set {
            OnNameChanging(value.ToUpper());   // 通知类要进行更改了
            m_name = value;                    // 更改字段
        }
    }
}

// 开发人员生成的代码，存储在另一个源代码文件中：
internal sealed partial class Base {

    // 这是分部方法的实现，会在 m_name 更改前调用
    partial void OnNameChanging(String value) {
        if (String.IsNullOrEmpty(value))
            throw new ArgumentNullException("value");
    }
}
```

这个新版本要注意以下几个问题。

- 类现在密封(虽然并非一定如此)。事实上，类可以是静态类，甚至可以是值类型。
- 工具和开发者所生成的代码真的是一个类型定义的两个部分。要更多地了解分部类型，请参见 6.5 节“分部类、结构和接口”。
- 工具生成的代码包含分部方法的声明。要用 partial 关键字标记，无主体(方法体)。
- 开发者生成的代码实现这个声明。该方法也要用 partial 关键字标记，有主体。

编译上述代码后，可以获得和原始代码一样的效果。现在的好处在于，可以重新运行工具，在新的源代码文件中生成新的代码，但你自己的代码是存储在一个单独的文件中的，不会受到影响。另外，这个技术可用于密封类、静态类以及值类型。

注意：在 Visual Studio 编辑器中，如果输入 partial 并按空格键，“智能感知”窗口会列出当前类型定义的、还没有匹配实现的所有分部方法声明。可以方便地从窗口中选择一个分部方法。然后，Visual Studio 会自动生成方法原型。这个功能提高了编程效率。

除此之外，分部方法还提供了另一个巨大的提升。如果不想修改工具生成的类型的行为，那么根本不需要提供自己的源代码文件。如果只是对工具生成的代码进行编译，编译器会改变生成的 IL 代码和元数据，使工具生成的代码看起来变成下面这样：

```
// 如果没有分部方法的“实现”部分，
// 那么工具生成的代码在逻辑上就等价于下面的代码
internal sealed class Base {
    private String m_name;

    public String Name {
        get { return m_name; }
        set {
            m_name = value; // 更改字段
        }
    }
}
```

也就是说，如果没有实现分部方法，编译器不会生成任何代表分部方法的元数据。此外，编译器不会生成任何调用分部方法的 IL 指令。而且，编译器不会生成对本该传给分部方法的实参进行求值的 IL 指令。在这个例子中，编译器不会生成调用 ToUpper 方法的代码。结果就是更少的元数据/IL，运行时性能也得到了提升。

注意：分部方法的工作方式类似于 System.Diagnostics.ConditionalAttribute 特性。然而，分部方法只能在单个类型中使用，而 ConditionalAttribute 能用于有选择地调用(invoke)另一个类型中定义的方法。

规则和指导原则

关于分部方法，有一些额外的规则和指导原则需要注意。

- 它们只能在分部类或结构中声明。
- 分部方法的返回类型始终是 void，任何参数都不能用 out 修饰符来标记。之所以有这两个限制，是因为方法在运行时可能不存在，所以不能将变量初始化为方法也许会返回的东西。类似地，不允许 out 参数是因为方法必须初始化它，而方法可能不存在。分部方法可以有 ref 参数，可以是泛型方法，可以是实例或静态方法，而且可以标记为 unsafe。
- 当然，分部方法的声明和实现部分必须具有完全一致的签名。如果两者都应用了定制特性，编译器会合并两个方法的特性。应用于参数的任何特性也会合并。
- 如果没有对应的实现部分，便不能在代码中创建一个委托来引用这个分部方法。这同样是由于方法在运行时不存在。编译器会报告以下消息：error CS0762: 无法通过方法"Base.OnNameChanging(string)"创建委托，因为该方法是没有实现声明的分部方法。
- 分部方法总是被视为 private 方法，但 C#编译器禁止在分部方法声明之前添加 private 关键字。

第 9 章 参数

本章内容：

- 可选参数和命名参数
- 隐式类型的局部变量
- 以传引用的方式向方法传递参数
- 向方法传递可变数量的参数
- 参数和返回类型的设计规范
- 常量性

本章重点在于向方法传递参数的各种方式，包括如何可选地指定参数，按名称指定参数，按引用传递参数，以及如何定义方法来接受可变数量的参数。

9.1 可选参数和命名参数

设计方法的参数时，可以为部分或全部参数分配默认值。然后，调用这些方法的代码可以选择不提供部分实参，直接使用其默认值。此外，调用方法时，可以通过指定参数名称来传递实参。以下代码演示了可选参数和命名(具名)参数的用法：

```
using System;
public static class Program {
    private static Int32 s_n = 0;

    private static void M(Int32 x = 9, String s = "A",
        DateTime dt = default(DateTime), Guid guid = new Guid()) {

        Console.WriteLine("x={0}, s={1}, dt={2}, guid={3}", x, s, dt, guid);
    }

    public static void Main() {
        // 1. 等同于 M(9, "A", default(DateTime), new Guid());
        M();

        // 2. 等同于 M(8, "X", default(DateTime), new Guid());
        M(8, "X");
```

```
            // 3. 等同于 M(5, "A", DateTime.Now, Guid.NewGuid());
            M(5, guid: Guid.NewGuid(), dt: DateTime.Now);

            // 4. 等同于 M(0, "1", default(DateTime), new Guid());
            M(s_n++, s_n++.ToString());

            // 5. 等同于以下两行代码：
            // String t1 = "2"; Int32 t2 = 3;
            // M(t2, t1, default(DateTime), new Guid());
            M(s: (s_n++).ToString(), x: s_n++);
        }
    }
```

运行程序得到以下输出：

```
x=9, s=A, dt=0001/1/1 0:00:00, guid=00000000-0000-0000-0000-000000000000
x=8, s=X, dt=0001/1/1 0:00:00, guid=00000000-0000-0000-0000-000000000000
x=5, s=A, dt=2023/9/22 23:11:26, guid=650de8db-dce3-4864-b055-c022e4456eef
x=0, s=1, dt=0001/1/1 0:00:00, guid=00000000-0000-0000-0000-000000000000
x=3, s=2, dt=0001/1/1 0:00:00, guid=00000000-0000-0000-0000-000000000000
```

如你所见，如果调用时省略了一个实参，C#编译器会自动嵌入参数的默认值。对 M 的第 3 个和第 5 个调用使用了 C#的命名参数功能。在这两个调用中，我为 x 显式传递了值，并指出要为名为 guid 和 dt 的参数传递实参。

向方法传递实参时，编译器按从左到右的顺序对实参进行求值。在对 M 的第 4 个调用中，s_n 的当前值(0)传给 x，然后 s_n 递增。随后，s_n 的当前值(1)作为字符串传给 s，然后继续递增到 2。使用命名参数传递实参时，编译器仍然按从左到右的顺序对实参进行求值。在对 M 的第 5 个调用中，s_n 中的当前值(2)被转换成字符串，并保存到编译器创建的临时变量(t1)中。接着，s_n 递增到 3，这个值保存到编译器创建的另一个临时变量(t2)中。然后，s_n 继续递增到 4。最后在调用 M 时，向它传递的实参依次是 t2，t1，一个默认 DateTime 和一个新建的 Guid。

9.1.1 规则和指导原则

如果在方法中为部分参数指定了默认值，请注意以下额外的规则和指导原则。

- 可为方法、构造器方法和有参属性(C#索引器)的参数指定默认值。还可为属于委托定义一部分的参数指定默认值。以后调用该委托类型的变量时可省略实参来接受默认值。
- 有默认值的参数必须放在没有默认值的所有参数之后。换言之，一旦定义了有默认值的参数，它右边的所有参数也必须有默认值。例如在前面的 M 方法定义中，如果删除 s 的默认值("A")，就会出现编译错误。但这个规则有一个例外：“参数数组”[1]这种参

[1] 在本章后面 9.4 节“向方法传递可变数量的参数”详细讨论。

数必须放在所有参数(包括有默认值的这些)之后，而且数组本身不能有一个默认值。

- 默认值必须是编译时能确定的常量值。那么，哪些参数能设置默认值？这些参数的类型可以是 C#认定的基元类型(参见第 5 章的表 5-1)。还包括枚举类型，以及能设为 `null` 的任何引用类型。值类型的参数可将默认值设为值类型的实例，并让它的所有字段都包含零值。可以使用 `default` 关键字或者 `new` 关键字来表达这个意思；两种语法将生成完全一致的 IL 代码。在 `M` 方法中设置 `dt` 参数和 `guid` 参数的默认值时，分别使用的就是这两种语法。

- 不要重命名参数变量，否则任何调用者以传参数名的方式传递实参，它们的代码也必须修改。例如，在前面的 `M` 方法声明中，如果将 `dt` 变量重命名为 `dateTime`，对 `M` 的第三个调用就会造成编译器显示以下消息：`error CS1739: "M"`的最佳重载没有名为`"dt"`的参数。

- 如果方法从模块外部调用，更改参数的默认值具有潜在的危险性。call site[①]在它的调用中嵌入默认值。如果以后更改了参数的默认值，但没有重新编译包含 call site 的代码，它在调用你的方法时就会传递旧的默认值。可以考虑将默认值 `0`/`null` 作为哨兵值使用，从而指出默认行为。这样一来，即使更改了默认值，也不必重新编译包含了 call site 的全部代码。下面是一个例子：

```
// 不要这样做：
private static String MakePath(String filename = "Untitled") {
    return String.Format(@"C:\{0}.txt", filename);
}

// 而要这样做：
private static String MakePath(String filename = null) {
    // 这里使用了空接合操作符(??)；详情参见第 19 章
    return String.Format(@"C:\{0}.txt", filename ?? "Untitled");
}
```

- 如果参数用 `ref` 或 `out` 关键字进行了标识，就不能设置默认值。因为没有办法为这些参数传递有意义的默认值。

使用可选或命名参数调用方法时，还要注意以下额外的规则和指导原则。

- 实参可按任意顺序传递，但命名实参只能出现在实参列表的尾部。

- 可按名称将实参传给没有默认值的参数，但所有必须的实参都必须传递(无论按位置还是按名称)，编译器才能编译代码。

- C#不允许省略逗号之间的实参，比如 `M(1, ,DateTime.Now)`。因为这会造成对可读性的影响，程序员将被迫去数逗号。对于有默认值的参数，如果想省略它们的实参，以

---

① call site 是发出调用的地方，可理解成调用了一个目标方法的表达式或代码行。——译注

传参数名的方式传递实参即可。

- 如果参数要求 ref/out，为了以传参数名的方式传递实参，请使用下面这样的语法：

```
// 方法声明：
private static void M(ref Int32 x) { ... }

// 方法调用：
Int32 a = 5;
M(x: ref a);
```

注意：写 C#代码和 Microsoft Office 的 COM 对象模型进行互操作时，C#的可选参数和命名参数功能非常好用。另外，调用 COM 组件时，如果是以传引用的方式传递实参，C#还允许省略 ref/out，进一步简化编码。但如果调用的不是 COM 组件，C#就要求必须向实参应用 ref/out 关键字。

9.1.2 DefaultParameterValue 和 Optional 特性

默认和可选参数的概念要不是 C#特有的就好了！具体地说，我们希望程序员能用一种编程语言定义一个方法，指出哪些参数是可选的，以及它们的默认值是什么。然后，另一种语言的程序员可以调用该方法。为了实现这一点，编译器必须允许调用者忽略一些实参，还必须能确定这些实参的默认值。

在 C#中，一旦为参数分配了默认值，编译器就会在内部向该参数应用定制特性 System.Runtime.InteropServices.OptionalAttribute。该特性会在最终生成的文件的元数据中持久性地存储下来。另外，编译器还会向参数应用 System.Runtime.InteropServices.DefaultParameterValueAttribut 特性，并将该特性持久性存储到生成的文件的元数据中。然后，会向 DefaultParameterValueAttribute 的构造器传递你在源代码中指定的常量值。

之后，一旦编译器发现某个方法调用缺失了部分实参，就可以确定省略的是可选的实参，并从元数据中提取默认值，将值自动嵌入调用中。

9.2 隐式类型的局部变量

C#能根据初始化表达式的类型推断方法中的局部变量的类型，如下所示：

```
private static void ImplicitlyTypedLocalVariables() {
  var name = "Jeff";
  ShowVariableType(name);        // 显示: System.String

  // var n = null;               // 错误，不能将 null 赋给隐式类型的局部变量
  var x = (String)null;          // 可以这样写，但意义不大
  ShowVariableType(x);           // 显示: System.String
```

```
    var numbers = new Int32[] { 1, 2, 3, 4 };
    ShowVariableType(numbers);     // 显示: System.Int32[]

    // 复杂类型能少打一些字
    var collection = new Dictionary<String, Single>() { { "Grant", 4.0f } };

    //显示: System.Collections.Generic.Dictionary`2[System.String,System.Single]
    ShowVariableType(collection);

    foreach (var item in collection) {
        //显示: System.Collections.Generic.KeyValuePair`2[System.String,System.Single]
        ShowVariableType(item);
    }
  }

  private static void ShowVariableType<T>(T t) {
    Console.WriteLine(typeof(T));
  }
```

ImplicitlyTypedLocalVariables 方法中的第一行代码使用 C#的 var 关键字引入了一个新的局部变量。为了确定 name 变量的类型，编译器要检查赋值操作符(=)右侧的表达式的类型。由于"Jeff"是字符串，所以编译器推断 name 的类型是 String。为了证明编译器正确推断出类型，我写了 ShowVariableType 方法。这个泛型方法推断它的实参的类型并在控制台上显示。为方便阅读，我在 ImplicitlyTypedLocalVariables 方法内部以注释形式列出了每次调用 ShowVariableType 方法的显示结果。

ImplicitlyTypedLocalVariables 方法内部的第二个赋值(被注释掉了)会造成编译错误：error CS0815: 无法将<null>赋予隐式类型化的局部变量。这是由于 null 能隐式转型为任何引用类型或可空值类型。因此，编译器无法推断它的确切类型。但在第三个赋值语句中，我证明只要显式指定了类型(本例是 String)，就可以将隐式类型(化)的局部变量初始化为 null。这样做虽然可行，但意义不大，因为可以写 String x = null;来获得同样效果。

第 4 个赋值语句反映了 C#“隐式类型局部变量”功能的真正价值。没有这个功能，就不得不在赋值操作符的左右两侧都指定 Dictionary<String, Single>。这不仅需要打更多的字，而且以后修改了集合类型或者任何泛型参数类型，赋值操作符两侧的代码也要修改。

在 foreach 循环中，我用 var 让编译器自动推断集合中的元素的类型。这证明了 var 能很好地用于 foreach，using 和 for 语句。它在对代码做实验时也很好用。例如，可以用方法的返回值初始化隐式类型的局部变量。开发方法时，则可以灵活更改返回类型。编译器能察觉到返回类型的变化，并自动更改变量的类型！当然，如果使用变量的代码没有相应地进行修改，还是像使用旧类型那样使用它，就可能无法编译。

在 Microsoft Visual Studio 中，鼠标放到 var 上将显示一条“工具提示”，指出编译器根据表达式推断出来的类型。在方法中使用匿名类型时必须用到 C#的隐式类型局部变量，详情参见第 10 章“属性”。

不能用 var 声明方法的参数类型。原因显而易见，因为编译器必须根据在 call site 传递的实参来推断参数类型，但 call site 可能一个都没有，也可能有好多个[①]。除此之外，不能用 var 声明类型中的字段。C#的这个限制是出于多方面的考虑。一个原因是字段可以被多个方法访问，而 C#团队认为这个协定(变量的类型)应该显式陈述。另一个原因是一旦允许，匿名类型(第 10 章)就会泄露到方法的外部。

重要提示：不要混淆 dynamic 和 var。用 var 声明局部变量只是一种简化语法，它要求编译器根据表达式推断具体数据类型。var 关键字只能声明方法内部的局部变量，而 dynamic 关键字适用于局部变量、字段和参数。表达式不能转型为 var，但能转型为 dynamic。必须显式初始化用 var 声明的变量，但无需初始化用 dynamic 声明的变量。欲知 C# dynamic 类型的详情，请参见 5.5 节“dynamic 基元类型”。

9.3 以传引用的方式向方法传递参数

CLR 默认所有方法参数都传值。传递引用类型的对象时，对象引用(或者说指向对象的指针)被传给方法。注意，引用(或指针)本身是传值的，这意味着方法能修改对象，而调用者能看到这些修改。对于值类型的实例，传给方法的是实例的一个副本，意味着方法将获得它专用的一个值类型实例副本，调用者中的实例不受影响。

重要提示：在方法中，你必须知道传递的每个参数是引用类型还是值类型，因为你编写的用于操作参数的代码可能会有显著的不同。

CLR 允许以传引用而非传值的方式传递参数。C#用关键字 out 或 ref 来为此提供支持。两个关键字都告诉 C#编译器生成元数据来指明该参数是传引用的。编译器将生成代码来传递参数的地址，而非传递参数本身。

CLR 本身不区分 out 和 ref，这意味着不管使用哪个关键字，都会生成相同的 IL 代码。另外，元数据也几乎完全一致，其中只有一个 bit 除外，它用于记录声明方法时指定的是 out 还是 ref。但是，C#编译器对这两个关键字是区别对待的，而且这个区别决定了由哪个方法负责初始化所引用的对象。如果方法的参数用 out 来标记，表明不指望调用者在调用方法之前初始化对象。被调用的方法不能一开始就直接读取参数的值，而且在返回之前必须向这个参数赋值。相反，如果方法的参数用 ref 来标记，调用者就必须在调用该方法前初始化参数的值，被调用的方法可以读取参数值以及/或者向参数写入。

对于 out 和 ref，引用类型和值类型的行为迥然有异。先看看为值类型使用 out 和 ref 的情况。

```
public sealed class Program {
```

[①] 要么一个类型都推断不出来，要么多个推断发生冲突。——译注

```
        public static void Main() {
            Int32 x;                      // x 没有初始化
            GetVal(out x);                // x 不必初始化
            Console.WriteLine(x);         // 显示 "10"
        }

        private static void GetVal(out Int32 v) {
            v = 10; // 该方法必须初始化 v
        }
    }
```

在上述代码中，x 在 Main 的栈帧[①]中声明。然后，x 的地址传递给 GetVal。GetVal 的 v 是一个指针，指向 Main 栈帧中的 Int32 值。在 GetVal 内部，v 指向的那个 Int32 被更改为 10。当 GetVal 返回时，Main 的 x 就有了一个为 10 的值，控制台上会显示 10。为大的值类型使用 out 可以提升代码的执行效率，因为它避免了在进行方法调用时复制值类型字段的实例。

下例将 out 替换成了 ref：

```
    public sealed class Program {
        public static void Main(){
            Int32 x = 5;              // x 已经初始化
            AddVal(ref x);            // x 必须初始化
            Console.WriteLine(x);     // 显示"15"
        }

        private static void AddVal(ref Int32 v) {
            v += 10;   // 该方法可以使用 v 的已初始化的值
        }
    }
```

在上述代码中，x 也在 Main 的栈帧中声明，并初始化为 5。然后，x 的地址传给 AddVal。AddVal 的 v 是一个指针，指向 Main 栈帧中的 Int32 值。在 AddVal 内部，v 指向的那个 Int32 要求必须是已经初始化的。因此，AddVal 可以在任何表达式中直接使用该初始值。AddVal 还可以自由更改这个值，新值会“返回给”调用者。在本例中，AddVal 将 10 加到初始值上。AddVal 返回时，Main 的 x 将包含 15，这个值会在控制台上显示出来。

综上所述，从 IL 和 CLR 的角度看，out 和 ref 是同一码事：都导致传递指向实例的一个指针。但从编译器的角度看，两者是有区别的。根据是 out 还是 ref，编译器会按照不同的标准来验证你写的代码是否正确。以下代码试图向要求 ref 参数的方法传递未初始化的值，结果是编译器报告以下错误：error CS0165: 使用了未赋值的局部变量"x"。

```
    public sealed class Program {
```

① 栈帧(stack frame)代表当前线程的调用栈中的一个方法调用。在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。详情参见 4.4 节。——译注

```
    public static void Main() {
        Int32 x;                // x 没有初始化

        // 下一行代码无法通过编译，编译器将报告：
        // error CS0165: 使用了未赋值的局部变量"x"
        AddVal(ref x);

        Console.WriteLine(x);
    }

    private static void AddVal(ref Int32 v) {
        v += 10; // 该方法可使用 v 的已初始化的值
    }
}
```

重要提示：经常有人问我，为什么 C#要求在调用方法时也必须指定 out 或 ref？毕竟，编译器知道被调用的方法要求的是 out 还是 ref，所以应该能正确编译代码。事实上，C#编译器确实能自动采取正确的操作。但是，C#语言的设计者认为调用者应显式表明意图。这样，在 call site(调用位置)那里，就可以很明显地看出被调用的方法会更改所传递的变量的值。

另外，CLR 允许根据使用的是 out 还是 ref 参数对方法进行重载。例如，在 C#中，以下代码是合法的，可以通过编译：

```
public sealed class Point {
    static void Add(Point p) { ... }
    static void Add(ref Point p) { ... }
}
```

两个重载方法仅 out 和 ref 有别则不合法，因为两个签名的元数据完全相同。所以，不能在上述 Point 类型中再定义以下方法：

```
static void Add(out Point p) { ... }
```

试图在 Point 类型中添加这个 Add 方法，C#编译器会显示以下消息：Error CS0663: "Point"不能定义仅在参数修饰符"out"和"ref"上存在区别的重载方法。

为值类型使用 out 和 ref，效果等同于以传值的方式传递引用类型。对于值类型，out 和 ref 允许方法操纵单一的值类型实例。调用者必须为实例分配内存，被调用者则操纵该内存(中的内容)。对于引用类型，调用者为一个指针分配内存(该指针指向一个引用类型的对象)，被调用者则操纵这个指针。正因为如此，仅当方法“返回”对其已知对象的引用时，为引用类型使用 out 和 ref 才有意义。以下代码对此进行了演示。

```
using System;
```

```
using System.IO;

public sealed class Program {
    public static void Main() {
        FileStream fs;   // fs 没有初始化

        // 打开第一个待处理的文件
        StartProcessingFiles(out fs);

        // 如果有更多需要处理的文件，就继续
        for (; fs != null; ContinueProcessingFiles(ref fs)) {

            // 处理一个文件
            fs.Read(...);
        }
    }

    private static void StartProcessingFiles(out FileStream fs) {
        fs = new FileStream(...);   // fs 必须在这个方法中初始化
    }

    private static void ContinueProcessingFiles(ref FileStream fs) {
        fs.Close();    // 关闭上一个操作的文件

        // 打开下一个文件；如果没有更多文件，就“返回”null
        if (noMoreFilesToProcess) fs = null;
        else fs = new FileStream (...);
    }
}
```

可以看出，上述代码最大的不同就是定义了一些使用了 out 或 ref 引用类型参数的方法，并用这些方法构造对象。指向新对象的指针将“返回给”调用者。还要注意，ContinueProcessingFiles 方法可以操作传给它的对象，并“返回”一个新对象。之所以能这样做，是因为参数是用 ref 关键字来标记的。上述代码可以稍微简化一下。

```
using System;
using System.IO;

public sealed class Program {
    public static void Main() {
        FileStream fs = null; // 初始化为 null (必要的操作)

        // 打开第一个待处理的文件
        ProcessFiles(ref fs);

        // 如果有更多需要处理的文件，就继续
        for (; fs != null; ProcessFiles(ref fs)) {

            // 处理文件
            fs.Read(...);
```

```
            }
        }

        private static void ProcessFiles(ref FileStream fs) {
            // 如果先前的文件是打开的，就将其关闭
            if (fs != null) fs.Close(); // 关闭上一个操作的文件

            // 打开下一个文件；如果没有更多的文件，就“返回”null
            if (noMoreFilesToProcess) fs = null;
            else fs = new FileStream (...);
        }
    }
```

下例演示了如何用 ref 关键字实现一个用于交换两个引用类型的方法：

```
    public static void Swap(ref Object a, ref Object b) {
        Object t = b; // t 代表“临时”
        b = a;
        a = t;
    }
```

为了交换对两个 String 对象的引用，你或许以为代码能像下面这样写：

```
    public static void SomeMethod(){
        String s1="Jeffrey";
        String s2="Richter";

        Swap(ref s1,ref s2);
        Console.WriteLine(s1); // 显示"Richter"
        Console.WriteLine(s2); // 显示"Jeffrey"
    }
```

但上述代码无法通过编译。问题在于，对于以传引用的方式传给方法的变量，它的类型必须与方法签名中声明的类型相同。换言之，Swap 预期的是两个 Object 引用，而不是两个 String 引用。为了交换两个 String 引用，代码要像下面这样写：

```
    public static void SomeMethod() {
        String s1 = "Jeffrey";
        String s2 = "Richter";

        // 以传引用的方式传递的变量，
        // 必须和方法预期的匹配
        Object o1 = s1, o2 = s2;
        Swap(ref o1, ref o2);

        // 完事后再将 Object 转型为 String
        s1 = (String) o1;
        s2 = (String) o2;

        Console.WriteLine(s1); // 显示"Richter"
        Console.WriteLine(s2); // 显示"Jeffrey"
```

```
}
```

这个版本的 SomeMethod 可以通过编译，并会如预期的一样执行。传递的参数之所以必须与方法预期的参数匹配，原因是保障类型安全性。以下代码(幸好不会编译)演示了类型安全性是如何被破坏的：

```
internal sealed class SomeType {
    public Int32 m_val;
}

public sealed class Program {
    public static void Main() {
        SomeType st;

        // 以下代码将产生编译错误：
        // error CS1503: 参数 1:无法从"out SomeType"转换为"out object"
        GetAnObject(out st);

        Console.WriteLine(st.m_val);
    }

    private static void GetAnObject(out Object o) {
        o = new String('X', 100);
    }
}
```

在上述代码中，Main 显然预期 GetAnObject 方法返回一个 SomeType 对象。但是，由于 GetAnObject 的签名表示的是一个 Object 引用，所以 GetAnObject 可以将 o 初始化为任意类型的对象。在这个例子中，当 GetAnObject 返回 Main 时，st 引用一个 String，显然不是 SomeType 对象，所以对 Console.WriteLine 的调用肯定会失败。幸好，C#编译器不会编译上述代码，因为 st 是一个 SomeType 引用，而 GetAnObject 要求的是一个 Object 引用。

可以通过泛型来修正这些方法，使它们按你的预期来运行。下面修正了前面的 Swap 方法：

```
public static void Swap<T>(ref T a, ref T b) {
    T t = b;
    b = a;
    a = t;
}
```

重写了 Swap 后，以下代码(和以前展示过的完全相同)就能通过编译了，而且能完美运行：

```
public static void SomeMethod(){
    String s1="Jeffrey";
    String s2="Richter";

    Swap(ref s1,ref s2);
    Console.WriteLine(s1); // 显示"Richter"
    Console.WriteLine(s2); // 显示"Jeffrey"
```

```
}
```

要查看用泛型来解决这个问题的其他例子，请参见 System.Threading 命名空间中的 Interlocked 类的 CompareExchange 和 Exchange 方法。

9.4 向方法传递可变数量的参数

方法有时需要获取可变数量的参数。例如，System.String 类型的一些方法允许连接(拼接)任意数量的字符串，还有一些方法允许指定一组要统一格式化的字符串。

为了接收可变数量的参数，方法要像下面这样声明：

```
static Int32 Add(params Int32[] values) {
    // 注意：如果愿意，可将 values 数组传给其他方法

    Int32 sum = 0;
    if (values != null) {
        for (Int32 x = 0; x < values.Length; x++)
            sum += values[x];
    }
    return sum;
}
```

除了 params 关键字，这个方法的一切对你来说都应该是非常熟悉的。params 只能应用于方法签名中的最后一个参数。暂时忽略 params 关键字，可以明显地看出 Add 方法接收一个 Int32 类型的数组，遍历数组，将其中的 Int32 值加到一起，结果 sum 返回给调用者。

显然，可以像下面这样调用该方法：

```
public static void Main() {
    // 显示 "15"
    Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 } ));
}
```

数组能用任意数量的一组元素来初始化，再传给 Add 方法进行处理。尽管上述代码可以通过编译并能正确运行，但并不好看。我们当然希望能像下面这样调用 Add 方法：

```
public static void Main() {
    // 显示 "15"
    Console.WriteLine(Add(1, 2, 3, 4, 5));
}
```

由于 params 关键字的存在，所以的确能这样做。params 关键字告诉编译器向参数应用定制特性 System.ParamArrayAttribute 的一个实例。

C#编译器检测到方法调用时，会先检查所有具有指定名称、同时参数没有应用 ParamArray 特性的方法。找到匹配的方法，就生成调用它所需的代码。没有找到，就接着检查应用了 ParamArray 特性的方法。找到匹配的方法，编译器先生成代码来构造一个数

组，填充它的元素，再生成代码来调用所选的方法。

上个例子并没有定义可获取 5 个 Int32 兼容实参的 Add 方法。但是，编译器发现在一个 Add 方法调用中传递了一组 Int32 值，而且有一个 Add 方法的 Int32 数组参数应用了 ParamArray 特性。因此，编译器认为这是一个匹配，会生成代码将实参保存到一个 Int32 类型的数组中，再调用 Add 方法并传递该数组。最终结果就是，你可以写代码直接向 Add 方法传递一组实参，而编译器会生成代码，像前面的第一个版本的方法调用那样，构造和初始化一个数组来容纳这些实参。

只有方法的最后一个参数才可以用 params 关键字(ParamArrayAttribute)标记。另外， 这个参数只能标识一维数组(任意类型)。可为这个参数传递 null 值，或传递对包含零个元素的一个数组的引用。以下 Add 调用能正常编译和运行，生成的结果是 0(和预期的一样)：

```
public static void Main() {
  // 以下两行都显示"0"
  Console.WriteLine(Add());           // 向 Add 传递 new Int32[0]
  Console.WriteLine(Add(null));       // 向 Add 传递 null：更高效(因为不会分配数组)
}
```

前面所有例子都只是演示了如何写方法来获取任意数量的 Int32 参数。那么，如何写方法来获取任意数量、任意类型的参数呢？答案很简单：只需修改方法原型，让它获取一个 Object[]而不是 Int32[]。以下方法显示传给它的每个对象的类型：

```
public sealed class Program {
    public static void Main() {
        DisplayTypes(new Object(), new Random(), "Jeff", 5);
    }

    private static void DisplayTypes(params Object[] objects) {
        if (objects != null) {
            foreach (Object o in objects)
                Console.WriteLine(o.GetType());
         }
    }
}
```

上述代码的输出如下：

```
System.Object
System.Random
System.String
System.Int32
```

重要提示：注意，调用参数数量可变的方法对性能有所影响(除非显式传递 null)。毕竟，数组对象必须在堆上分配，数组元素必须初始化，而且数组的内存最终需要垃圾回收。要减小对性能的影响，可考虑定义几个没有使用 params 关键字的重载版本。关于这方面的范例，请参考 System.String 类的 Concat 方法，该方法定义了以下重载版本：

```
public sealed class String : Object, ... {
    public static string Concat(object arg0);
    public static string Concat(object arg0, object arg1);
    public static string Concat(object arg0, object arg1, object arg2);
    public static string Concat(params object[] args);

    public static string Concat(string str0, string str1);
    public static string Concat(string str0, string str1, string str2);
    public static string Concat(string str0, string str1, string str2, string str3);
    public static string Concat(params string[] values);
}
```

如你所见，Concat 方法定义了几个没有使用 params 关键字的重载版本。这是为了改善常规情形下的性能。使用了 params 关键字的重载则用于不太常见的情形；在这些情形下，性能有一定的损失。但幸运的是，这些情形本来就不常见。

9.5 参数和返回类型的设计规范

声明方法的参数类型时，应尽量指定最弱的类型，宁愿要接口也不要基类。例如，如果要写方法来处理一组数据项，最好是用接口(比如 IEnumerable<T>)来声明参数，而不要用强数据类型(比如 List<T>)或者更强的接口类型(比如 ICollection<T>或 IList<T>)：

```
// 好：方法使用了弱参数类型
public void ManipulateItems<T>(IEnumerable<T> collection) { ... }

// 不好：方法使用了强参数类型
public void ManipulateItems<T>(List<T> collection) { ... }
```

原因是调用第一个方法时可传递数组对象、List<T>对象、String 对象或者其他对象——只要对象的类型实现了 IEnumerable<T>接口。相反，第二个方法只允许传递 List<T>对象，不接受数组或 String 对象。显然，第一个方法更好，它更灵活，适合更广泛的情形。

当然，如果方法需要的是列表(而非任何可枚举的对象)，就应该将参数类型声明为 IList<T>。但仍然要避免将参数类型声明为 List<T>。声明为 IList<T>，调用者可以向方法传递数组和实现了 IList<T>的其他类型的对象。

注意，这里的例子讨论的是集合，是用接口体系结构来设计的。讨论使用基类体系结构设计的类时，概念同样适用。例如，要实现对流中的字节进行处理的方法，可定义以下方法：

```
// 好：方法使用了弱参数类型
```

```
public void ProcessBytes(Stream someStream) { ... }

// 不好：方法使用了强参数类型
public void ProcessBytes(FileStream fileStream) { ... }
```

第一个方法能处理任何流，包括 FileStream，NetworkStream 和 MemoryStream 等。第二个则只能处理 FileStream 流，这限制了它的应用。

与此相反的是，一般最好将方法的返回类型声明为最强的类型(防止受限于一个特定类型)。例如，方法最好返回 FileStream 而不是 Stream 对象：

```
// 好：方法使用了强返回类型
public FileStream OpenFile() { ... }

// 不好：方法使用了弱返回类型
public Stream OpenFile() { ... }
```

第一个方法是首选的，它允许方法的调用者将返回对象视为 FileStream 对象或者 Stream 对象。但第二个方法要求调用者只能将返回对象视为 Stream 对象。总之，要确保调用者在调用方法时有尽量大的灵活性，使方法的适用范围更大。

有的时候，需要在不影响调用者的前提下修改方法的内部实现。在刚才的例子中，OpenFile 方法不太可能更改内部实现来返回除 FileStream(或 FileStream 的派生类型)之外的其他对象。但是，如果方法返回的是 List<String>对象，就可能想在未来的某个时候修改它的内部实现来返回一个 String[]。如果想保持一定的灵活性，于未来更改方法返回的东西，那么请事先选择一个较弱的返回类型。例如：

```
// 灵活：方法使用了较弱的返回类型
public IList<String> GetStringCollection() { ... }

// 不灵活：方法使用了较强的返回类型
public List<String> GetStringCollection() { ... }
```

在这个例子中，即使 GetStringCollection 方法在内部使用一个 List<String>对象并返回它，但最好还是修改方法的原型，使它返回一个 IList<String>。将来，GetStringCollection 方法可以更改它的内部集合来使用一个 String[]；与此同时，不需要修改调用者的源代码。事实上，调用者甚至不需要重新编译。注意，这个例子在较弱的类型中选择的是最强的那一个。例如，它没有使用最弱的 IEnumerable<String>，也没有使用较强的 ICollection<String>。[①]

---

① IList 继承自 ICollection，ICollection 继承自 IEnumerable。作者的意思是说，在这三个比 List<String>都弱的类型中“矮子里面挑高个”，选择最强的 IList。——译注

9.6 常量性

有的语言(比如非托管 C++)允许将方法或参数声明为常量，从而禁止实例方法中的代码更改对象的任何字段，或者更改传给方法的任何对象。CLR 没有提供这个功能，许多程序员因此觉得遗憾。既然 CLR 都不提供，面向它的任何编程语言(包括 C#)自然也无法提供。

首先要注意，非托管 C++将实例方法或参数声明为 `const` 只能防止程序员用一般的代码来更改对象或参数。方法内部总是可以更改对象或实参的。这要么是通过强制类型转换来去掉“常量性”，要么通过获取对象/实参的地址，再向那个地址写入。从某种意义上说，非托管 C++向程序员撒了一个谎，使他们以为常量对象或实参不能写入(其实可以)。

实现类型时，开发人员可以避免写操纵对象或实参的代码。例如，`String` 类就没有提供任何能更改 `String` 对象的方法，所以字符串是不可变(immutable)的。

此外，微软很难为 CLR 赋予验证常量对象/实参未被更改的能力。CLR 将不得不对每个写入操作进行验证，确定该写入针对的不是常量对象。这对性能的影响非常大。当然，如果检测到有违反常量性的地方，会造成 CLR 抛出异常。此外，如果支持常量性，还会给开发人员带来大量复杂性。例如，如果类型是不可变的，它的所有派生类型都不得不遵守这个约定。除此之外，在不可变的类型中，字段也必须不可变。

考虑到这些原因以及其他许多原因，CLR 没有提供对常量对象/实参的支持。

第 10 章 属性

本章内容：

- 无参属性
- 有参属性
- 调用属性访问器方法时的性能
- 属性访问器的可访问性
- 泛型属性访问器方法

本章讨论**属性**(properties)，它允许源代码用简化语法来调用方法。CLR 支持两种属性：**无参属性**，平时说的属性就是指它；**有参属性**，它在不同的编程语言中有不同的称呼。例如，C#将有参属性称为**索引器**，Microsoft Visual Basic 将有参属性称为**默认属性**。还要讨论如何使用“对象和集合初始化器”来初始化属性，以及如何用 C#的匿名类型和 `System.Tuple` 类型将多个属性打包到一起。

10.1 无参属性

许多类型都定义了能被获取或更改的状态信息。这种状态信息一般作为类型的字段成员实现。例如，以下类型定义包含两个字段：

```
public sealed class Employee {
    public String Name;          // 员工姓名
    public Int32 Age;            // 员工年龄
}
```

创建该类型的实例后，可以使用以下形式的代码轻松获取(get)或设置(set)它的状态信息：

```
Employee e = new Employee();
e.Name = "Jeffrey Richter";      // 设置员工姓名
e.Age = 45;                      // 设置员工年龄

Console.WriteLine(e.Name);  // 显示 "Jeffrey Richter"
```

这种查询和设置对象状态信息的做法十分常见。但我必须争辩的是，永远都不应该像这样实现。面向对象设计和编程的重要原则之一就是**数据封装**，这意味着类型的字段永远不应

该公开，否则很容易因为不恰当使用字段而破坏对象的状态。例如，以下代码可以很容易地破坏一个 Employee 对象：

```
e.Age = -5;   // 怎么可能有人是-5 岁呢？
```

还有其他原因促使我们封装对类型中的数据字段的访问。其一，你可能希望访问字段来执行一些副作用①、缓存某些值或者推迟创建一些内部对象②。其二，你可能希望以线程安全的方式访问字段。其三，字段可能是一个逻辑字段，它的值不由内存中的字节表示，而是通过某个算法来计算获得。

基于上述原因，强烈建议将所有字段都设为 private。要允许用户或类型获取或设置状态信息，就公开一个针对该用途的方法。封装了字段访问的方法通常称为**访问器**(accessor)方法。访问器方法可选择对数据的合理性进行检查，确保对象的状态永远不被破坏。例如，我将上一个类重写为以下形式：

```
public sealed class Employee {
    private String m_Name;  // 字段现在是私有的
    private Int32 m_Age;    // 字段现在是私有的

    public String GetName() {
        return(m_Name);
    }

    public void SetName(String value) {
        m_Name = value;
    }

    public Int32 GetAge() {
        return(m_Age);
    }

    public void SetAge(Int32 value) {
        if (value < 0)
            throw new ArgumentOutOfRangeException("value",
                value.ToString(),
                "The value must be greater than or equal to 0");
        m_Age = value;
    }
}
```

虽然这只是一个简单的例子，但还是可以看出数据字段封装带来的巨大好处。另外可以看出，实现只读属性或只写属性是多么简单！只需选择不实现一个访问器方法即可。另外，

① 即 side effect；在计算机编程中，如果一个函数或表达式除了生成一个值，还会造成状态的改变，就说它会造成副作用；或者说会执行一个副作用。——译注

② 推迟创建对象是指在对象第一次需要时才真正创建它。——译注

将 SetXxx 方法标记为 protected，就可以只允许派生类型修改值。

但是，像这样进行数据封装有两个缺点。首先，因为不得不实现额外的方法，所以必须写更多的代码；其次，类型的用户必须调用方法，而不能直接引用字段名。

```
e.SetName("Jeffrey Richter");           // 更新员工姓名
String EmployeeName = e.GetName();      // 获取员工姓名
e.SetAge(41);                           // 更新员工年龄
e.SetAge(-5);                           // 抛出 ArgumentOutOfRangeException 异常
Int32 EmployeeAge = e.GetAge();         // 获取员工年龄
```

我个人认为这些缺点微不足道。不过，编程语言和 CLR 还是提供了一个称为**属性**(property)的机制。它缓解了第一个缺点所造成的影响，同时完全消除了第二个缺点。

下面的类使用了属性，它与前面定义的类功能相同：

```
public sealed class Employee {
    private String m_Name;
    private Int32 m_Age;

    public String Name {
        get { return (m_Name); }
        set { m_Name = value; } // 关键字 value 总是代表新值
    }

    public Int32 Age {
        get { return (m_Age); }
        set {
            if (value < 0)  // 关键字 value 总是代表新值
                throw new ArgumentOutOfRangeException("value",
                    value.ToString(),
                    "The value must be greater than or equal to 0");
            m_Age = value;
        }
    }
}
```

如你所见，属性使类型的定义稍微复杂了一些，但由于属性允许采用以下形式来写代码，所以额外的付出还是值得的：

```
e.Name = "Jeffrey Richter";         // set 员工姓名
String EmployeeName = e.Name;       // get 员工姓名
e.Age = 41;                         // set 员工年龄
e.Age = -5;                         // 抛出 ArgumentOutOfRangeException 异常
Int32 EmployeeAge = e.Age;          // get 员工年龄
```

可将属性想象成**智能字段**，即背后有额外逻辑的字段。CLR 支持静态、实例、抽象和虚属性。另外，属性可用任意“可访问性”修饰符来标记(详情参见第 6.3 节“成员的可访问性”)，而且可以在接口中定义(详情参见第 13 章“接口”)。

每个属性都有名称和类型(类型不能是 void)。属性不能重载，即不能定义名称相同、类型不同的两个属性。定义属性时通常同时指定 get 和 set 两个方法。但可省略 set 方法来定义只读属性，或省略 get 方法来定义只写属性。

经常利用属性的 get 和 set 方法操纵类型中定义的私有字段。私有字段通常称为**支持字段**(backing field)。但 get 和 set 方法并非一定要访问支持字段。例如，System.Threading.Thread 类型提供了 Priority 属性来直接和操作系统通信。Thread 对象内部没有一个关于线程优先级的字段。没有支持字段的另一种典型的属性是在运行时计算的只读属性。例如，以 0 结束的一个数组的长度或者已知高度和宽度的一个矩形的面积。

定义属性时，取决于属性的定义，编译器在最后的托管程序集中生成以下两项或三项。

- 代表属性 get 访问器的方法。仅在属性定义了 get 访问器方法时生成。
- 代表属性 set 访问器的方法。仅在属性定义了 set 访问器方法时生成。
- 托管程序集元数据中的属性定义。这一项必然生成。

以前面的 Employee 类型为例。编译器编译该类型时发现其中的 Name 和 Age 属性。由于两者都有 get 和 set 访问器方法，所以编译器在 Employee 类型中生成 4 个方法定义，这造成原始代码似乎是像下面这样写的：

```
public sealed class Employee {
    private String     m_Name;
    private Int32      m_Age;

    public String get_Name() {
        return m_Name;
    }
    public void   set_Name(String value) {
        m_Name = value;  // 实参 value 总是代表新设的值
    }

    public Int32 get_Age() {
        return m_Age;
    }

    public void set_Age(Int32 value) {
        if (value < 0)  // value 总是代表新值
            throw new ArgumentOutOfRangeException("value",
                value.ToString(),
                "The value must be greater than or equal to 0");
        m_Age = value;
    }
}
```

编译器在你指定的属性名之前自动附加 get_或 set_前缀来生成方法名。C#内建了对属性的支持。C#编译器发现代码试图获取或设置属性时，实际会生成对上述某个方法的调用。

即使编程语言不直接支持属性，也可调用需要的访问器方法来访问属性。效果一样，只是代码看起来没那么优雅。

除了生成访问器方法，针对源代码中定义的每一个属性，编译器还会在托管程序集的元数据中生成一个属性定义项。在这个记录项中，包含了一些标志(flags)以及属性的类型。另外，它还引用了 get 和 set 访问器方法。这些信息唯一的作用就是在“属性”这种抽象概念与它的访问器方法之间建立起一个联系。编译器和其他工具可以利用这种元数据信息(使用 System.Reflection.PropertyInfo 类来获得)。CLR 不使用这种元数据信息，在运行时只需要访问器方法。

10.1.1 自动实现的属性

如果只是为了封装一个支持字段而创建属性，C#还提供了一种更简洁的语法，称为**自动实现的属性**(Automatically Implemented Property，后文简称为 AIP)，例如下面的 Name 属性：

```
public sealed class Employee {
    // 自动实现的属性
    public String Name { get; set; }

    private Int32 m_Age;

    public Int32 Age {
        get { return(m_Age); }
        set {
            if (value < 0)     // value 关键字总是代表新值
                throw new ArgumentOutOfRangeException("value",
                   value.ToString(),
                   "The value must be greater than or equal to 0");
             m_Age = value;
        }
    }
}
```

声明属性而不提供 get/set 方法的实现，C#会自动为你声明一个私有字段。在本例中，字段的类型是 String，也就是属性的类型。另外，编译器会自动实现 get_Name 和 set_Name 方法，分别返回和设置字段中的值。

和直接声明名为 Name 的 public String 字段相比，AIP 的优势在哪里？事实上，两者存在一处重要的区别。使用 AIP，意味着你已经创建了一个属性。访问该属性的任何代码实际都会调用 get 和 set 方法。如果以后决定自己实现 get 和/或 set 方法，而不是接受编译器的默认实现，访问属性的任何代码都不必重新编译。然而，如果将 Name 声明为字段，以后又想把它更改为属性，那么访问字段的所有代码都必须重新编译才能访问属性方法。

- 我个人不喜欢编译器的 AIP 功能，通常会避免使用它。理由是：字段声明语法可以包含初始化部分，这样就可以在一行代码中声明并初始化字段。但是，没有简单的语法初始化 AIP。因此，必须在每个构造器方法中显式初始化每个 AIP。

- 运行时序列化引擎将字段名持久存储到序列化的流中。AIP 的支持字段名称由编译器决定，每次重新编译代码都可能更改这个名称。因此，任何类型只要含有一个 AIP，就没办法对该类型的实例进行反序列化。在任何想要序列化或反序列化的类型中，都不要使用 AIP 功能。

- 调试时不能在 AIP 的 get 或 set 方法上添加断点，所以不好检测应用程序在什么时候获取或设置这个属性。相反，手动实现的属性可以设置断点，追踪错误时更方便。

还要注意，如果使用 AIP，属性必然是可读和可写的。换言之，编译器肯定同时生成 get 和 set 方法。这很合理，对于一个只写的字段，它的值不能读取有什么用呢？类似地，只读字段肯定具有默认值。另外，由于不知道编译器生成的支持字段到底是什么名字，所以代码只能用属性名访问属性。而且，如果你决定显式实现任何访问器方法(get 或 set)，那么两个访问器方法都必须显式实现，这时也就用不上 AIP 了。换言之，AIP 是作用于整个属性的；要么都用，要么都不用。不能显式实现一个访问器方法，而让另一个自动实现。

10.1.2 合理定义属性

我个人不喜欢属性，我还希望 Microsoft .NET Framework 及其编程语言不要提供对属性的支持。理由是属性看起来和字段相似，但本质是方法。这造成了大量误解。程序员在看到貌似访问字段的代码时，会做出一些对属性来说不成立的假定，具体如下所示。

- 属性可以只读或只写，而字段访问总是可读和可写的(一个例外是 readonly 字段仅在构造器中可写)。如果定义属性，最好同时为它提供 get 和 set 访问器方法。

- 属性方法可能抛出异常；字段访问永远不会。

- 属性不能作为 out 或 ref 参数传给方法，而字段可以。例如以下代码是编译不了的：

```
using System;

public sealed class SomeType {
    private static String Name {
        get { return null; }
        set {}
    }

    static void MethodWithOutParam(out String n) { n = null; }

    public static void Main() {
        // 对于下一行代码，C#编译器将报告以下错误消息：
        // error CS0206: 属性或索引器不能作为 out 或 ref 参数传递
        MethodWithOutParam(out Name);
    }
}
```

- 属性方法可能花较长时间执行，字段访问则总是立即完成。许多人使用属性是为了线程同步，这就可能造成线程永远阻塞。所以，要线程同步就不要使用属性，而要使用

方法。此外，如果你的类可以被远程访问(例如，在类从 System.MashalByRefObject 派生的情况下)，那么调用属性方法会非常慢。在这种情况下应该优先使用方法而不是属性。我个人认为，从 MashalByRefObject 派生的类永远都不应该使用属性。

- 连续多次调用，属性方法每次都可能返回不同的值，字段则每次都返回相同的值。例如，System.DateTime 类的只读属性 Now 返回当前日期和时间。每次查询这个属性都返回不同的值。这是一个错误，微软现在很想修正这个错误，将 Now 改成方法而不是属性。Environment 类的 TickCount 属性也是微软犯的一个错。

- 属性方法可能造成可观察到的副作用[①]，字段访问则永远不会。类型的使用者应该能按照他/她选择的任何顺序设置类型定义的各个属性，而不会造成类型中(因为设置顺序的不同)出现不同的行为。

- 属性方法可能需要额外的内存，或者返回的引用并非指向对象状态一部分，造成对返回对象的修改作用不到原始对象身上。而查询字段所返回的引用总是指向原始对象状态的一部分。使用会返回一个拷贝的属性很容易引起混淆，文档也经常没有专门说明。[②]

属性和 Visual Studio 调试器

Microsoft Visual Studio 允许在调试器的监视窗口中输入一个对象的属性。这样一来，每次遇到一个断点，调试器都会调用属性的 get 访问器方法，并显示返回值。这个技术在追踪错误时很有用，但也有可能造成 bug，并损害调试性能。例如，假定为“网络共享”中的文件创建了一个 FileStream，然后将 FileStream 的 Length 属性添加到调试器的监视窗口中。现在，每次遇到一个断点，调试器都会调用 Length 的 get 访问器方法，该方法内部向服务器发出一个网络请求来获取文件的当前长度。这显著影响了性能！

类似地，假定属性的 get 访问器方法有一个“副作用”，那么每次抵达断点，都会执行该“副作用”。例如，假定属性的 get 访问器方法在每次调用时都会递增一个计数器(一个“副作用”)，那么每次抵达断点，都会递增这个计数器。考虑到可能有这样的问题，Visual Studio 允许为监视窗口中显示的属性关闭属性求值。要在 Visual Studio 中关闭属性求值，请选择“工具”|“选项”|“调试”|“常规”。然后，在如图 10-1 所示的列表框中，清除勾选“启用属性求值和其他隐式函数调用”。注意，即使清除了这个选项，仍可将属性添加到监视窗口，然后手动强制 Visual Studio 对它进行求值。为此，单击监视窗口“值”列中的强制求值圆圈即可。

---

① 这里的副作用(side effect)是指，访问属性时，除了单纯设置或获取属性，还会造成对象状态的改变。如果存在多个副作用，程序的行为就要依赖于历史；或者说要依赖于求值顺序。如果以不同顺序设置属性，而类型每一次的行为都不同，那么显然是不合理的。——译注

② 指开发人员在文档中没有清楚地指明这是属性，而且返回的是一个拷贝，造成别人在使用这个类时产生混淆。——译注

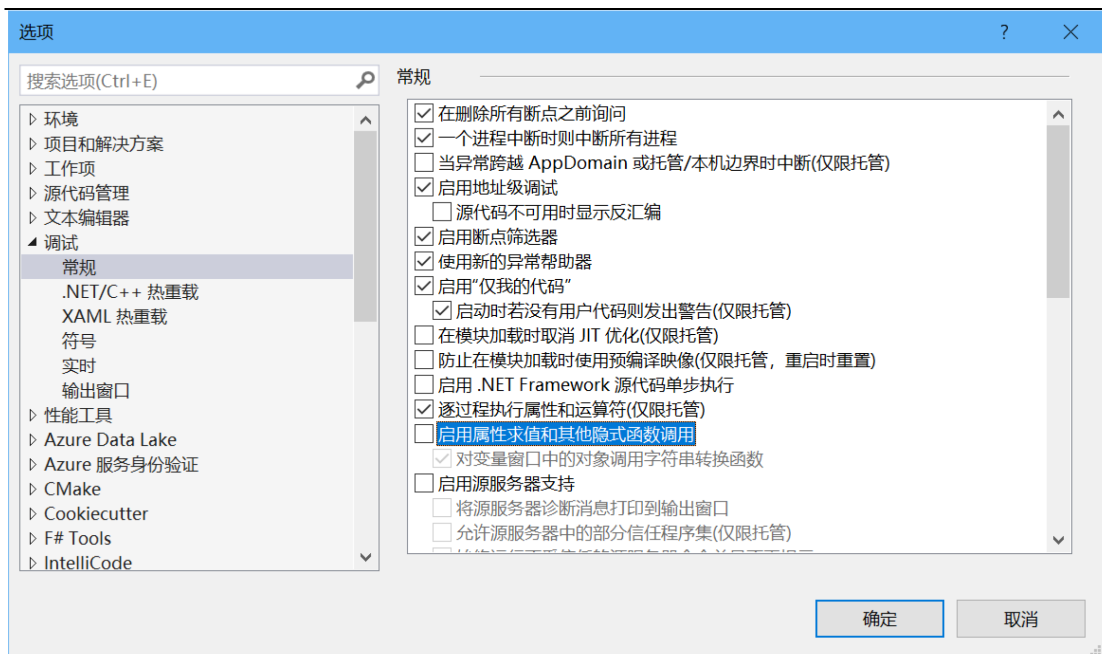


图 10-1 Visual Studio 的常规调试设置

我发现，现在的人对属性的依赖有过之而无不及，经常是不管有没有必要都使用属性。仔细研究一下上面的属性与字段区别列表，你会发现只有在极个别的情况下，定义属性才真正有用，同时不会造成开发人员的混淆。属性唯一的好处就是提供了简化的语法。和调用普通方法(非属性中的方法)相比，属性不仅不会提升代码的性能，还会妨碍对代码的理解。要是我参与.NET Framework 以及编译器的设计，根本就不会提供属性；相反，我会让程序员老老实实地实现 `GetXxx` 和 `SetXxx` 方法。然后，编译器可以提供一些特殊的、简化的语法来调用这些方法。但是，我希望编译器使用有别于字段访问的语法，使程序员能真正理解他们正在做什么——是在调用一个方法！

10.1.3 对象和集合初始化器

经常要构造一个对象并设置对象的一些公共属性(或字段)。为了简化这种常见的编程模式，C#语言支持一种特殊的对象初始化语法。下面是一个例子：

```
Employee e = new Employee() { Name = "Jeff", Age = 45 };
```

这个语句做了好几件事情，包括构造一个 `Employee` 对象，调用它的无参构造器，将它的公共 `Name` 属性设为`"Jeff"`，并将公共 `Age` 属性设为 `45`。事实上，这一行代码等价于以下几行代码。可以自行检查两者的 IL 代码来加以验证。

```
Employee e = new Employee();
e.Name = "Jeff";
e.Age = 45;
```

对象初始化器语法真正的好处在于，它允许在表达式的上下文(相对于语句的上下文)中编码，允许组合多个函数，进而增强了代码的可读性。例如，现在可以写这样的代码：

```
String s = new Employee() { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

这个语句做的事情更多，首先还是构造一个 Employee 对象，调用它的构造器，再初始化两个公共属性。然后，在结果表达式上，先调用 ToString，再调用 ToUpper。要深入了解函数的组合使用，请参见 8.6 节“扩展方法”。

作为一个小的补充，如果想调用的本来就是一个无参构造器，C#还允许省略起始大括号之前的圆括号。下面这行代码生成与上一行相同的 IL：

```
String s = new Employee { Name = "Jeff", Age = 45 }.ToString().ToUpper();
```

如果属性的类型实现了 IEnumerable 或 IEnumerable<T>接口，属性就被认为是集合，而集合的初始化是一种相加(additive)操作，而非替换(replacement)操作。例如，假定有下面这个类定义：

```
public sealed class Classroom {
    private List<String> m_students = new List<String>();
    public List<String> Students { get { return m_students; } }

    public Classroom() {}
}
```

现在可以写代码来构造一个 Classroom 对象，并像下面这样初始化 Students 集合：

```
public static void M() {
    Classroom classroom = new Classroom {
        Students = { "Jeff", "Kristin", "Aidan", "Grant" }
    };

    // 显示教室中的 4 个学生
    foreach (var student in classroom.Students)
        Console.WriteLine(student);
}
```

编译上述代码时，编译器发现 Students 属性的类型是 List<String>，而且这个类型实现了 IEnumerable<String>接口。现在，编译器假定 List<String>类型提供了一个名为 Add 的方法(因为大多数集合类都提供了 Add 方法将数据项添加到集合)。然后，编译器生成代码来调用集合的 Add 方法。所以，上述代码会被编译器转换成这样：

```
public static void M() {
    Classroom classroom = new Classroom();
    classroom.Students.Add("Jeff");
    classroom.Students.Add("Kristin");
    classroom.Students.Add("Aidan");
    classroom.Students.Add("Grant");

    // 显示教室中的 4 个学生
```

```
        foreach (var student in classroom.Students)
            Console.WriteLine(student);
    }
```

如果属性的类型实现了 IEnumerable 或 IEnumerable<T>，但未提供 Add 方法，编译器就不允许使用集合初始化语法向集合中添加数据项；相反，编译器报告以下消息：error CS0117: "System.Collections.Generic.IEnumerable<string>"不包含"Add"的定义。

有的集合的 Add 方法要获取多个实参，比如 Dictionary 的 Add：

```
public void Add(TKey key, TValue value);
```

通过在集合初始化器中嵌套大括号的方式，可以向 Add 方法传递多个实参，如下所示：

```
var table = new Dictionary<String, Int32> {
    { "Jeffrey", 1 }, { "Kristin", 2 }, { "Aidan", 3 }, { "Grant", 4 }
};
```

它等价于以下代码：

```
var table = new Dictionary<String, Int32>();
table.Add("Jeffrey", 1);
table.Add("Kristin", 2);
table.Add("Aidan", 3);
table.Add("Grant", 4);
```

10.1.4 匿名类型

利用 C#的匿名类型功能，可以用很简洁的语法来自动声明不可变(immutable)的元组类型。元组[①]类型是含有一组属性的类型，这些属性通常以某种方式相互关联。在以下代码的第一行中，我定义了含有两个属性(String 类型的 Name 和 Int32 类型的 Year)的类，构造了该类型的实例，将 Name 属性设为"Jeff"，将 Year 属性设为 1964。

```
// 定义类型，构造实例，并初始化属性
var o1 = new { Name = "Jeff", Year = 1964 };

// 在控制台上显示属性：Name=Jeff, Year=1964
Console.WriteLine("Name={0}, Year={1}", o1.Name, o1.Year);
```

第一行代码创建了匿名类型，我没有在 new 关键字后指定类型名称，所以编译器会自动创建类型名称，而且不会告诉我这个名称具体是什么(这正是匿名的含义)。这行代码使用上一节讨论的“对象初始化器”语法来声明属性，同时初始化这些属性。另外，由于我(开发人员)不知道编译时的类型名称，也就不知道变量 o1 声明为什么类型。但这不是问题，因为可以像第 9 章讨论过的那样使用 C#的“隐式类型局部变量”功能(var)。它的作用是告诉编译器根据赋值操作符(=)右侧的表达式推断类型。

① 元组(tuple)一词来源于对顺序的抽象：single，double，triple，quadruple，quintuple，*n*-tuple。

现在，让我们将注意力放在编译器实际做的事情上面。遇到下面这样的一行代码时：

```
var o = new { 属性1 = 表达式1, ..., 属性N = 表达式N };
```

编译器会推断每个表达式的类型，创建推断类型的私有字段，为每个字段创建公共只读属性，并创建一个构造器来接收所有这些表达式。在构造器的代码中，会用传给它的表达式的求值结果来初始化私有只读字段。除此之外，编译器还会重写 Object 的 Equals，GetHashCode 和 ToString 方法，并生成所有这些方法中的代码。最终，编译器生成的类看起来像下面这样：

```
[CompilerGenerated]
internal sealed class <>f__AnonymousType0<...>: Object {
    private readonly t1 f1;
    public t1 p1 { get { return f1; } }

    ...

    private readonly tn fn;
    public tn pn { get { return fn; } }

    public <>f__AnonymousType0<...>(t1 a1, ..., tn an) {
        f1 = a1; ...; fn = an; // 设置所有字段
    }

    public override Boolean Equals(Object value) {
        // 任何字段不匹配就返回 false，否则返回 true
    }

    public override Int32 GetHashCode() {
        // 返回根据每个字段的哈希码生成的一个哈希码
    }

    public override String ToString() {
        // 返回“属性名=值”对的以逗号分隔的列表
    }
}
```

编译器会生成 Equals 和 GetHashCode 方法，因此匿名类型的实例能放到哈希表集合中。属性是只读的，而非可读可写，目的是防止对象的哈希码发生改变。如果对象在哈希表中作为键使用，那么更改它的哈希码会造成再也找不到它。编译器会生成 ToString 方法来帮助进行调试。在 Visual Studio 调试器中，可将鼠标指针放在引用了匿名类型实例的一个变量上方。随后，Visual Studio 会调用 ToString 方法，并在一个提示窗口中显示结果字符串。顺便说一句，在编辑器中写代码时，Visual Studio 的“智能感知”功能会提示属性名，这是非常好用的一个功能。

编译器支持用另外两种语法在匿名类型中声明属性。匿名类型能根据变量推断属性名和类型。

```
String Name = "Grant";
DateTime dt = DateTime.Now;

// 有两个属性的一个匿名类型
// 1. String Name 属性设为"Grant"
// 2. Int32 Year 属性设为 dt 中的年份
var o2 = new { Name, dt.Year };
```

在这个例子中，编译器判断第一个属性应该叫 Name。由于 Name 是局部变量的名称，所以编译器将属性类型设为与局部变量相同的类型：String。对于第二个属性，编译器使用字段/属性的名称：Year。Year 是 DateTime 类的一个 Int32 属性，所以匿名类型中的 Year 属性也是一个 Int32。现在，当编译器构造这个匿名类型的一个实例时，会将实例的 Name 属性设为 Name 局部变量中的值，使 Name 属性引用同一个"Grant"字符串。编译器还要将实例的 Year 属性设为从 dt 的 Year 属性返回的同一个值。

编译器在定义匿名类型时是非常“善解人意”的。如果它看到你在源代码中定义了多个匿名类型，而且这些类型具有相同的结构，那么它只会创建一个匿名类型定义，但创建该类型的多个实例。所谓“相同的结构”，是指在这些匿名类型中，每个属性都有相同的类型和名称，而且这些属性的指定顺序相同。在前面的几个例子中，变量 o1 和 o2 就是同类型的，因为在定义匿名类型的两行代码中，都是先一个 String 类型的 Name 属性，再一个 Int32 类型的 Year 属性。

由于两个变量(o1 和 o2)的类型相同，所以可以做一些非常“酷”的事情，比如检查两个对象是否包含相等的值，并将对一个对象的引用赋给正在指向另一个对象的变量，如下所示：

```
// 类型支持相等性测试和赋值操作
Console.WriteLine("Objects are equal: " + o1.Equals(o2));
o1 = o2; // 赋值
```

另外，由于类型的这种同一性，所以可以创建一个隐式类型的数组(详情参见 16.1 节“初始化数组元素”)，在其中包含一组匿名类型的对象。

```
// 之所以能这样写，是因为所有对象都是同一个匿名类型
var people = new[] {
    o1, // o1 的定义参见本节开头
    new { Name = "Kristin", Year = 1970 },
    new { Name = "Aidan", Year = 2003 },
    new { Name = "Grant", Year = 2008 }
};

// 下面展示如何遍历匿名类型的对象构成的一个数组(var 是必须的)
foreach (var person in people)
    Console.WriteLine("Person={0}, Year={1}", person.Name, person.Year);
```

匿名类型经常与 LINQ(Language Integrated Query，语言集成查询)配合使用。可用 LINQ 执行查询，从而生成由一组对象构成的集合，这些对象都是相同的匿名类型。然后，可以对结果集合中的对象进行处理。所有这些都是在同一个方法中发生的。下例展示了如何返回

“文档”文件夹中过去 7 天修改过的所有文件：

```
String myDocuments = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
var query =
       from pathname in Directory.GetFiles(myDocuments)
       let LastWriteTime = File.GetLastWriteTime(pathname)
       where LastWriteTime > (DateTime.Now - TimeSpan.FromDays(7))
       orderby LastWriteTime
       select new { Path = pathname, LastWriteTime };  // 匿名类型的对象构成的集合

foreach (var file in query)
  Console.WriteLine("LastWriteTime={0}, Path={1}", file.LastWriteTime, file.Path);
```

匿名类型的实例不能泄露到方法外部。方法原型不能接受匿名类型的参数，因为无法指定匿名类型(的名称)。类似地，方法也不能返回对匿名类型的引用。虽然可以将匿名类型的实例视为一个 `Object`(所有匿名类型都从 `Object` 派生)，但没办法将 `Object` 类型的变量转型回匿名类型，因为不知道在匿名类型在编译时的名称。要传递元组，应考虑使用下一节讨论的 `System.Tuple` 类型。

10.1.5 System.Tuple 类型

微软在 `System` 命名空间中定义了几个泛型 `Tuple` 类型，它们全都从 `Object` 派生，区别只在于元数[①](泛型参数的个数)。下面演示了最简单和最复杂的 `Tuple` 类型：

```
// 这是最简单的：
[Serializable]
public class Tuple<T1> {
    private T1 m_Item1;
    public Tuple(T1 item1) { m_Item1 = item1; }
    public T1 Item1 { get { return m_Item1; } }
}

// 这是最复杂的：
[Serializable]
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> {
    private T1 m_Item1; private T2 m_Item2; private T3 m_Item3; private T4 m_Item4;
    private T5 m_Item5; private T6 m_Item6; private T7 m_Item7; private TRest m_Rest;
    public Tuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5, T6 item6, T7 item7,
        TRest rest) {
        m_Item1 = item1; m_Item2 = item2; m_Item3 = item3; m_Item4 = item4;
        m_Item5 = item5; m_Item6 = item6; m_Item7 = item7; m_Rest = rest;
    }

    public T1 Item1 { get { return m_Item1; } }
```

① 元数的英文是 arity。在计算机编程中，一个函数或运算(操作)的元数是指函数获取的实参或操作数的个数。它源于像 unary(arity=1)、binary(arity=2)、ternary(arity=3)这样的单词。——译注

```
        public T2 Item2 { get { return m_Item2; } }
        public T3 Item3 { get { return m_Item3; } }
        public T4 Item4 { get { return m_Item4; } }
        public T5 Item5 { get { return m_Item5; } }
        public T6 Item6 { get { return m_Item6; } }
        public T7 Item7 { get { return m_Item7; } }
        public TRest Rest { get { return m_Rest; } }
    }
```

和匿名类型相似，Tuple 创建好之后就不可变了(所有属性都只读)。虽然这里没有显示，但 Tuple 类还提供了 CompareTo，Equals，GetHashCode 和 ToString 方法以及 Size 属性。此外，所有 Tuple 类型都实现了 IStructuralEquatable，IStructuralComparable 和 IComparable 接口，所以可以比较两个 Tuple 对象，逐个比对它们的字段。请参考文档更多地了解这些方法和接口。

下面的示例泛型方法 MinMax 用一个 Tuple 类型向调用者返回两样信息：

```
// 用 Item1 返回最小值 & 用 Item2 返回最大值
private static Tuple<Int32, Int32>MinMax(Int32 a, Int32 b) {
    return new Tuple<Int32, Int32>(Math.Min(a, b), Math.Max(a, b));
}

// 下面展示了如何调用方法，以及如何使用返回的 Tuple
private static void TupleTypes() {
    var minmax = MinMax(6, 2);

    // Min=2, Max=6
    Console.WriteLine("Min={0}, Max={1}", minmax.Item1, minmax.Item2);
}
```

当然，很重要的一点是 Tuple 的生产者(写它的人)和消费者(用它的人)必须对 Item#属性返回的内容有一个清楚的理解。对于匿名类型，属性的实际名称是根据定义匿名类型的源代码来确定的。对于 Tuple 类型，属性一律被微软称为 Item#，我们无法对此进行任何改变。遗憾的是，这种名字没有任何实际的含义或意义，所以要由生产者和消费者为它们分配具体含义。这还影响了代码的可读性和可维护性。所以，应该在自己的代码添加详细的注释，使生产者和消费者取得共识。

编译器只能在调用泛型方法时推断泛型类型，调用构造器时则不能。因此，System 命名空间还包含了一个非泛型静态 Tuple 类，其中包含一组静态 Create 方法，能根据实参推断泛型类型。这个类扮演了创建 Tuple 对象的一个“工厂”的角色，它存在的唯一意义便是简化你的代码。下面用静态 Tuple 类重写了刚才的 MinMax 方法：

```
// 用 Item1 返回最小值 & 用 Item2 返回最大值
private static Tuple<Int32, Int32>MinMax(Int32 a, Int32 b) {
    return Tuple.Create(Math.Min(a, b), Math.Max(a, b));  // 更简单的语法
}
```

要创建 8 个或 8 个以上元素的 Tuple，可为 Rest 参数传递另一个 Tuple，如下所示：

```
var t = Tuple.Create(0, 1, 2, 3, 4, 5, 6, Tuple.Create(7, 8));
Console.WriteLine("{0}, {1}, {2}, {3}, {4}, {5}, {6}, {7}, {8}",
    t.Item1, t.Item2, t.Item3, t.Item4, t.Item5, t.Item6, t.Item7,
    t.Rest.Item1.Item1, t.Rest.Item1.Item2);
```

注意：除了匿名类型和 Tuple 类型，还可研究一下 System.Dynamic.ExpandoObject 类(在 System.Core.dll 程序集中定义)。这个类和 C# dynamic 类型(参见 5.5 节“dynamic 基元类型”)配合使用，就可以采取另一种方式将一系列属性(键/值对)组合到一起。虽然实现不了编译时的类型安全性，而且得不到“智能感知”的支持，但语法看起来不错，而且还可以在 C#和 Python 这样的动态语言之间传递 ExpandoObject 对象。以下是使用了一个 ExpandoObject 的示例代码：

```
dynamic e = new System.Dynamic.ExpandoObject();
e.x = 6;      // 添加一个 Int32 'x'属性，其值为 6
e.y = "Jeff"; // 添加一个 String 'y'属性，其值为"Jeff"
e.z = null;   // 添加一个 Object 'z'属性，其值为 null

// 查看所有属性及其值：
foreach (var v in (IDictionary<String, Object>)e)
    Console.WriteLine("Key={0}, V={1}", v.Key, v.Value);

// 删除'x'属性及其值：
var d = (IDictionary<String, Object>)e;
d.Remove("x");
```

10.2 有参属性

在上一节中，属性的 get 访问器方法不接受参数，所以我把它们称为**无参属性**。由于用起来就像访问字段，所以很容易理解。除了这些与字段相似的属性，编程语言还支持我所谓的**有参属性**，其 get 访问器方法接收一个或多个参数，set 访问器方法则接收两个或多个参数。不同编程语言以不同方式公开有参属性。另外，不同编程语言对有参属性的称呼也不同。C#称为**索引器**，Visual Basic 则称为**默认属性**。本节主要讨论 C#如何使用有参属性来公开索引器。

C#使用数组风格的语法来公开有参属性(索引器)。换言之，可将索引器看成是 C#开发人员对[]操作符的重载。下面是一个示例 BitArray 类，它允许用数组风格的语法来索引由该类的实例维护的一组二进制位。

```
using System;

public sealed class BitArray {
    // 容纳了二进制位的私有字节数组
    private Byte[] m_byteArray;
```

```
        private Int32 m_numBits;

        // 下面的构造器用于分配字节数组，并将所有位设为0
        public BitArray(Int32 numBits) {
            // 先验证实参
            if (numBits <= 0)
                throw new ArgumentOutOfRangeException("numBits must be > 0");

            // 保存位的个数
            m_numBits = numBits;

            // 为位数组分配字节
            m_byteArray = new Byte[(numBits + 7) / 8];
        }

        // 下面是索引器(有参属性)
        public Boolean this[Int32 bitPos] {

            // 下面是索引器的 get 访问器方法
            get {
                // 先验证实参
                if ((bitPos < 0) || (bitPos >= m_numBits))
                    throw new ArgumentOutOfRangeException("bitPos");

                // 返回指定索引处的位的状态
                return (m_byteArray[bitPos / 8] & (1 << (bitPos % 8))) != 0;
            }

            // 下面是索引器的 set 访问器方法
            set {
                if ((bitPos < 0) || (bitPos >= m_numBits))
                    throw new ArgumentOutOfRangeException("bitPos",
                                                          bitPos.ToString());
                if (value) {
                    // 将指定索引处的位设为 true
                    m_byteArray[bitPos / 8] = (Byte)
                         (m_byteArray[bitPos / 8] | (1 << (bitPos % 8)));
                } else {
                    // 将指定索引处的位设为 false
                    m_byteArray[bitPos / 8] = (Byte)
                         (m_byteArray[bitPos / 8] & ~(1 << (bitPos % 8)));
                }
            }
        }
    }
```

BitArray 类的索引器用起来很简单：

```
    // 分配含 14 个位的 BitArray 数组
    BitArray ba = new BitArray(14);
```

```
// 调用 set 访问器方法，将编号为偶数的所有位都设为 true
for (Int32 x = 0; x < 14; x++) {
    ba[x] = (x % 2 == 0);
}

// 调用 get 访问器方法显示所有位的状态
for (Int32 x = 0; x < 14; x++) {
    Console.WriteLine("Bit " + x + " is " + (ba[x] ? "On" : "Off"));
}
```

在这个示例 BitArray 类中，索引器获取 Int32 类型的参数 bitPos。所有索引器至少要有一个参数，但可以有更多。这些参数(和返回类型)可以是除了 void 之外的任意类型。在 System.Drawing.dll 程序集的 System.Drawing.Imaging.ColorMatrix 类中，提供了有多个参数的一个索引器的例子。

经常要创建索引器来查询关联数组[①]中的值。System.Collections.Generic.Dictionary 类型就提供了这样的一个索引器，它获取一个键，并返回与该键关联的值。和无参属性不同，类型可提供多个重载的索引器，只要这些索引器的签名不同。

和无参属性的 set 访问器方法相似，索引器的 set 访问器方法同样包含了一个隐藏参数，在 C#中称为 value。该参数代表想赋给“被索引元素”的新值。

CLR 本身并不区分无参属性和有参属性。对 CLR 来说，每个属性都只是类型中定义的一对方法和一些元数据。如前所述，不同编程语言要求用不同的语法来创建和使用有参属性。将 this[...]作为表达索引器的语法，这纯粹是 C#团队自己的选择。也正是因为这个选择，所以 C#只允许在对象的实例上定义索引器。C#不支持定义静态索引器属性，虽然 CLR 是支持静态有参属性的。

由于 CLR 以相同的方式对待有参和无参属性，所以编译器会在托管程序集中生成以下两项或三项。

- 代表有参属性 get 访问器的方法。仅在属性定义了 get 访问器方法时生成。
- 代表有参属性 set 访问器的方法。仅在属性定义了 set 访问器方法时生成。
- 托管程序集元数据中的属性定义。这一项必然生成。没有专门的有参属性元数据定义表，因为对于 CLR 来说，有参属性和普通的属性无异。

对于前面的 BitArray 类，编译器在编译索引器时，原始代码似乎是像下面这样写的：

```
public sealed class BitArray {

    // 下面是索引器的 get 访问器方法
    public Boolean get_Item(Int32 bitPos) { /* ... */ }
```

① 关联数组(associative array)使用字符串索引(称为键)来访问存储在数组中的值(称为值)。——译注

```
    // 下面是索引器的 set 访问器方法
    public void set_Item(Int32 bitPos, Boolean value) { /* ... */ }
}
```

编译器在索引器名称之前附加 get_或者 set_前缀，从而自动生成这些方法的名称。由于 C#的索引器语法不允许开发人员指定索引器名称，所以 C#编译器团队不得不为访问器方法选择一个默认名称；他们最后选择了 Item。因此，编译器生成的方法名就是 get_Item 和 set_Item。

查看文档时，留意类型是否提供了名为 Item 的属性，从而判断该类型是否提供了索引器。例如，System.Collections.Generic.List 类型提供了名为 Item 的公共实例属性，它就是 List 的索引器。

用 C#编程时，永远看不到 Item 这个名称，所以一般无需关心编译器在幕后选择的是什么名称。但是，如果为类型设计的索引器要由其他语言的代码访问，就可能需要更改索引器的 get 和 set 访问器方法所用的默认名称 Item。C#允许向索引器应用定制特性 System.Runtime.CompilerServices.IndexerNameAttribute 来重命名这些方法，如下所示：

```
using System;
using System.Runtime.CompilerServices;

public sealed class BitArray {

    [IndexerName("Bit")]
    public Boolean this[Int32 bitPos] {
        // 这里至少要定义一个访问器方法
    }
}
```

现在，编译器将生成名为 get_Bit 和 set_Bit 的方法，而不是 get_Item 和 set_Item。编译时，C#编译器会注意到 IndexerName 特性，它告诉编译器如何对方法和属性的元数据进行命名。特性本身不会进入程序集的元数据①。

以下 Visual Basic 代码演示了如何访问这个 C#索引器：

```
' 构造 BitArray 类型的实例
Dim ba as New BitArray(10)

' Visual Basic 用()而不是[]指定数组元素
Console.WriteLine(ba(2)) ' 显示 True 或 False

' Visual Basic 还允许通过索引器的名称来访问它
Console.WriteLine(ba.Bit(2)) ' 显示的内容和上一行代码相同
```

C#允许一个类型定义多个索引器，只要索引器的参数集不同。在其他语言中，IndexerName 特性允许定义多个相同签名的索引器，因为索引器各自可以有不同的名称。C#不允许这样做，是因为它的语法不是通过名称来引用索引器，编译器不知道你引用的是哪个索引器。编译以下 C#代码将导致编译器报错：error C0111: 类型"SomeType"已定义了一个名为"this"的具有相同参数类型的成员。

```
using System;
using System.Runtime.CompilerServices;

public sealed class SomeType {
    // 定义 get_Item 访问器方法
    public Int32 this[Boolean b] {
        get { return 0; }
    }

    // 定义 get_Jeff 访问器方法
    [IndexerName("Jeff")]
    public String this[Boolean b] {
        get { return null; }
    }
```

① 正是因为这个原因，才造成了 IndexerNameAttribute 类不是 CLI 和 C#语言的 ECMA 标准的一部分。

```
    }
```

显然，C#将索引器看成是对[]操作符的重载，而[]操作符不能用来消除具有不同方法名和相同参数集的有参属性的歧义。

顺便说一句，System.String 类型是改变了索引器名称的一个例子。String 的索引器的名称是 Chars，而不是 Item。这个只读属性允许从字符串中获得一个单独的字符。对于不用[]操作符语法来访问这个属性的编程语言，Chars 这个名称更有意义。

选择主要有参属性

C#对索引器的限制会引起以下两个问题。

- 如果定义类型的语言允许多个有参属性应该怎么办？
- 从 C#中如何使用这个类型？

这两个问题的答案是类型必须选择其中一个有参属性名来作为默认(主要)属性，这要求向类本身应用 System.Reflection.DefaultMemberAttribute 的一个实例。要说明的是，该特性可应用于类、结构或接口。在 C#中编译定义了有参属性的类型时，编译器会自动向类型应用该特性的实例，同时会考虑到你可能应用了 IndexerName 特性的情况。在 DefaultMemberAttribute 特性的构造器中，指定了要作为类型的默认有参属性使用的名称。

因此，如果在 C#中定义包含有参属性的类型，但没有指定 IndexerName 特性，那么在向该类型应用的 DefaultMember 特性中，会将默认成员的名称指定为 Item。如果向有参属性应用了 IndexerName 特性，那么在向该类型应用的 DefaultMember 特性中，会将默认成员的名称指定为由 IndexerName 特性指定的字符串名称。记住，如果代码中含有多个名称不一样的有参属性，C#将无法编译。

而对于支持多个有参属性的编程语言，必须从中选择一个属性方法名，并用 DefaultMember 特性进行标识。这是 C#代码唯一能访问的有参属性。

C#编译器发现代码试图获取或设置索引器时，会生成对其中一个方法的调用。有的语言不支持有参属性。要从这种语言中访问有参属性，必须显式调用需要的索引器方法。对于 CLR，无参和有参属性没有区别，所以可用相同的 System.Reflection.PropertyInfo 类来发现有参属性和它的访问器方法之间的关联。

10.3 调用属性访问器方法时的性能

对于简单的 get 和 set 访问器方法，JIT 编译器会将代码内联(inline，或者说嵌入)。这样一来，使用属性(而不是使用字段)就没有性能上的损失。内联是指将方法(目前说的是访问

器方法)的代码直接编译到调用它的方法中。这就避免了在运行时发出调用所产生的开销，代价是编译好的方法变得更大。由于属性访问器方法包含的代码一般很少，所以对内联会使生成的本机代码变得更小，而且执行得更快。

注意：JIT 编译器在调试代码时不会内联属性方法，因为内联的代码会变得难以调试。这意味着在程序的 Release 版本中，访问属性时的性能可能比较快；而在程序的 Debug 版本中，则可能比较慢。相应地，在 Debug 和 Release 版本中，字段的访问速度都很快。

10.4 属性访问器的可访问性

有的时候，我们希望为 get 访问器方法指定一种可访问性，为 set 访问器方法指定另一种可访问性。最常见的一种情形是提供公共 get 访问器和受保护 set 访问器。

```
public class SomeType {
    private String m_name;
    public String Name {
        get { return m_name; }
        protected set { m_name = value;}
    }
}
```

如上述代码所示，Name 属性本身声明为 public 属性，意味着 get 访问器方法是公共的，所有代码都能调用。而 set 访问器方法声明为 protected，只能从 SomeType 内部定义的代码中调用，或者从 SomeType 的派生类的代码中调用。

定义属性时，如果两个访问器方法需要不同的可访问性，C#要求必须为属性本身指定限制最小的可访问性。然后，两个访问器只能选择一个来使用限制较大的。在前面的例子中，属性本身声明为 public，而 set 访问器方法声明为 protected(限制较 public 大)。

10.5 泛型属性访问器方法

既然属性本质上是方法，而 C#和 CLR 允许泛型方法，所以有时可能想在定义属性时引入它自己的泛型类型参数(而非使用当前包容它的类型的泛型类型参数)，但 C#不允许。之所以属性不能引入它自己的泛型类型参数，最主要的原因是概念上说不通。属性本应表示可供查询或设置的一个对象特征。一旦引入泛型类型的参数，就意味着有可能改变查询/设置的行为。但属性不应该和行为沾边。要想公开对象的行为(无论是不是泛型)，应该定义方法而非属性。

第 11 章 事件

本章内容：

- 设计要公开事件的类型
- 编译器如何实现事件
- 设计侦听事件的类型
- 显式实现事件

本章讨论可以在类型中定义的最后一种成员：事件。定义了事件成员的类型允许类型(或类型的实例)通知其他对象发生了特定的事情。例如，`Button` 类提供了 `Click` 事件。应用程序中的一个或多个对象可接收关于该事件的通知，以便在 `Button` 被单击(click)之后采取特定操作。我们用“事件”这种类型成员来实现这种交互。具体地说，定义了事件成员的类型能提供以下能力。

- 方法能登记(register)它对事件的关注。
- 方法能注销(unregister)它对事件的关注。
- 事件发生时，登记了的方法将收到通知。

类型之所以能提供事件通知功能，是因为类型维护了一个已登记方法的列表。事件发生后，类型将通知列表中所有已登记的方法。

CLR 事件模型以**委托**为基础。委托本质上是一种类型，提供了调用[①]回调方法的一种类型安全的方式。对象凭借回调方法接收它们订阅的通知。委托虽然会在本章开始使用，但它的完整细节是在第 17 章“委托”中讲述的。

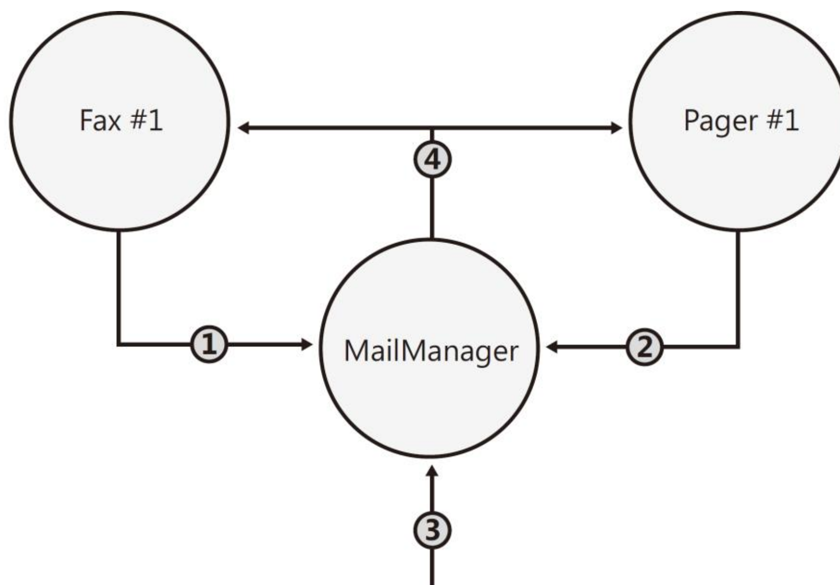
为了帮你完整地理解事件在 CLR 中的工作机制，先来描述事件很有用的一个场景。假定

---

① 这个“调用”(invoke)理解为“唤出”更恰当。它和普通的“调用”(call)稍有不同。在英语的语境中，invoke 和 call 的区别在于，在执行一个所有信息都已知的方法时，用 call 比较恰当。这些信息包括要引用的类型、方法的签名以及方法名。但是，在需要先“唤出”某个东西来帮你调用一个信息不明的方法时，用 invoke 就比较恰当。但是，由于两者均翻译为“调用”不会对读者的理解造成太大的困扰，所以本书仍然采用约定俗成的方式来进行翻译，只是在必要的时候附加英文原文提醒你区分。——译注

要设计一个电子邮件应用程序。电子邮件到达时，用户可能希望将该邮件转发给传真机或寻呼机。先设计名为 `MailManager` 的类型来接收传入的电子邮件，它公开 `NewMail` 事件。其他类型(如 `Fax` 和 `Pager`)的对象登记对于该事件的关注。`MailManager` 在收到新电子邮件时会引发(raise)该事件，造成邮件分发给每一个已登记的对象。每个对象都以它们自己的方式处理邮件。

应用程序初始化时只实例化一个 `MailManager` 实例，然后可以实例化任意数量的 `Fax` 和 `Pager` 对象。图 11-1 展示了应用程序如何初始化，以及新电子邮件到达时发生的事情。



1. Fax对象中的一个方法登记对MailManager的 NewMail事件的关注。
2. Pager对象中的一个方法登记对MailManager的NewMail事件的关注。
3. 一封新邮件到达MailManager。
4. MailManager对象将事件通知发送给所有已登记的方法，这些方法以自己的方式处理邮件。

图 11-1 设计使用了事件的应用程序

图 11-1 的应用程序首先构造 `MailManager` 的一个实例。`MailManager` 提供了 `NewMail` 事件。构造 `Fax` 和 `Pager` 对象时，它们向 `MailManager` 的 `NewMail` 事件登记自己的一个实例方法。这样当新邮件到达时，`MailManager` 就知道通知 `Fax` 和 `Pager` 对象。`MailManager` 将来在收到新邮件时会引发 `NewMail` 事件，使所有已登记的方法都有机会以自己的方式处理邮件。

11.1 设计要公开事件的类型

开发人员通过连续几个步骤定义公开了一个或多个事件成员的类型。本节详细描述了每个

必要的步骤。MailManager 示例应用程序(参见本书配套代码)展示了 MailManager 类型、Fax 类型和 Pager 类型的所有源代码。注意，Pager 类型和 Fax 类型几乎完全相同。

11.1.1 第一步：定义类型来容纳所有需要发送给事件通知接收者的附加信息

事件引发时，引发事件的对象可能希望向接收事件通知的对象传递一些附加信息。这些附加信息需要封装到它自己的一个类中。该类通常包含一组私有字段，以及一些用于公开这些字段的只读公共属性。根据约定，这种类应该从 System.EventArgs 派生，而且类名以 EventArgs 结束。本例将该类命名为 NewMailEventArgs 类，它的各个字段分别标识了发件人(m_from)、收件人(m_to)和主题(m_subject)。

```
// 第一步：定义一个类型来容纳所有应该发送给事件通知接收者的附加信息
internal class NewMailEventArgs : EventArgs {

    private readonly String m_from, m_to, m_subject;

    public NewMailEventArgs(String from, String to, String subject) {
        m_from = from; m_to = to; m_subject = subject;
    }

    public String From { get { return m_from; } }
    public String To { get { return m_to; } }
    public String Subject { get { return m_subject; } }
}
```

注意：EventArgs 类在 Microsoft .NET Framework 类库(FCL)中定义，这是这样实现的：

```
[ComVisible(true), Serializable]
public class EventArgs {
    public static readonly EventArgs Empty = new EventArgs();
    public EventArgs() { }
}
```

可以看出，该类型的实现非常简单，就是一个让其他类型继承的基类型。许多事件都没有附加信息需要传递。例如，当一个 Button 向已登记的接收者通知自己被单击时，直接调用回调方法即可。定义不需要传递附加数据的事件时，可以直接使用 EventArgs.Empty，不用构造新的 EventArgs 对象。

11.1.2 第二步：定义事件成员

事件成员使用 C#关键字 event 定义。每个事件成员都要指定以下内容：可访问性标识符(几乎肯定是 public，这样其他代码才能访问该事件成员)；委托类型，指出要调用的方法

的原型；以及名称(可为任何有效的标识符)。以下是我们的 MailManager 类中的事件成员：

```
internal class MailManager {

    // 第二步：定义事件成员
    public event EventHandler<NewMailEventArgs> NewMail;
    ...
}
```

NewMail 是事件名称。事件成员的类型是 EventHandler<NewMailEventArgs>，意味着“事件通知”的所有接收者都必须提供一个原型和 EventHandler<NewMailEventArgs>委托类型匹配的回调方法。由于泛型 System.EventHandler 委托类型的定义如下：

```
public delegate void EventHandler<TEventArgs>(Object sender, TEventArgs e);
```

所以方法原型必须具有以下形式：

```
void MethodName(Object sender, NewMailEventArgs e);
```

注意：许多人奇怪事件模式为什么要求 sender 参数是 Object 类型。毕竟，只有 MailManager 才会引发传递了 NewMailEventArgs 对象的事件，所以回调方法更合适的原型似乎是下面这个：

```
void MethodName(MailManager sender, NewMailEventArgs e);
```

要求 sender 是 Object 主要是因为继承。例如，假定 MailManager 成为 SmtpMailManager 的基类，那么回调方法的 sender 参数应该是 SmtpMailManager 类型而不是 MailManager 类型。但这不可能发生，因为 SmtpMailManager 继承了 NewMail 事件。所以，如果代码需要由 SmtpMailManager 引发事件，还是要将 sender 实参转型为 SmtpMailManager。反正都要进行类型转换，这和将 sender 定为 Object 类型没什么两样。

将 sender 参数的类型定为 Object 的另一个原因是灵活性。它使委托能由多个类型使用，只要求类型提供一个会传递 NewMailEventArgs 对象的事件。例如，即使 PopMailManager 类不是从 MailManager 类派生的，也能使用这个委托。

此外，事件模式要求委托定义和回调方法将派生自 EventArgs 的参数命名为 e。这个要求唯一的作用就是加强事件模式的一致性，使开发人员更容易学习和实现这个模式。注意，能自动生成源代码的工具(比如 Microsoft Visual Studio)也知道将参数命名为 e。

最后，事件模式要求所有事件处理程序[1]的返回类型都是 void。这很有必要，因为引发事件后可能要调用好几个回调方法，但没办法获得所有方法的返回值。将返回类型定为 void，

[1] 本书按约定俗成的译法将 event handler 翻译成“事件处理程序”，但请把它理解成“事件处理方法”(在 VB 中，则理解成“事件处理 Sub 过程”)。——译注

就不允许回调(方法)返回一个值。遗憾的是，FCL 中的一些事件处理程序没有遵循微软自己规定的模式。例如，ResolveEventHandler 事件处理程序会返回 Assembly 类型的一个对象。

11.1.3 第三步：定义负责引发事件的方法来通知事件的登记对象

按照约定，类要定义一个受保护的虚方法。引发事件时，类及其派生类中的代码会调用该方法。方法只获取一个参数，即一个 NewMailEventArgs 对象，其中包含了要传给接收通知的对象的信息。方法的默认实现只是检查一下是否有对象登记了对事件的关注。如果有，就引发事件来通知事件的登记对象。该方法在 MailManager 类中看起来像下面这样：

```
internal class MailManager {
    ...
    // 第三步：定义负责引发事件的方法来通知已登记的对象。
    // 如果类是密封的，该方法要声明为私有和非虚
    protected virtual void OnNewMail(NewMailEventArgs e) {

        // 出于线程安全的考虑，现在将对委托字段的引用复制到一个临时变量中
        EventHandler<NewMailEventArgs> temp = Volatile.Read(ref NewMail);

        // 任何方法登记了对事件的关注，就通知它们
        if (temp != null) temp(this, e);
    }
    ...
}
```

以线程安全的方式引发事件

.NET Framework 刚发布时建议开发者用以下方式引发事件：

```
// 版本 1
protected virtual void OnNewMail(NewMailEventArgs e) {
    if (NewMail != null) NewMail(this, e);
}
```

OnNewMail 方法的问题在于，虽然线程检查出 NewMail 不为 null，但就在调用 NewMail 之前，另一个线程可能从委托链中移除一个委托，使 NewMail 变成了 null。这会抛出 NullReferenceException 异常。为了修正这个竞态问题，许多开发者都像下面这样写 OnNewMail 方法

```
// 版本 2
protected virtual void OnNewMail(NewMailEventArgs e) {
    EventHandler<NewMailEventArgs> temp = NewMail;
    if (temp != null) temp(this, e);
}
```

它的思路是，将对 NewMail 的引用复制到临时变量 temp 中，后者引用赋值发生时的委托链。然后，方法比较 temp 和 null，并调用(invoke)temp；所以，向 temp 赋值后，即使另一个线程更改了 NewMail 也没有关系。委托是不可变的(immutable)，所以这个技术理论上行得通。但许多开发者没有意识到的是，编译器可能“擅做主张”，通过完全移除局部变量 temp 的方式对上述代码进行优化。如果发生这种情况，版本 2 就和版本 1 就没有任何区别。所以，仍有可能抛出 NullReferenceException 异常。

要想彻底修正这个问题，应该像下面这样重写 OnNewMail：

```
// 版本 3
protected virtual void OnNewMail(NewMailEventArgs e) {
   EventHandler<NewMailEventArgs> temp = Volatile.Read(ref NewMail);
   if (temp != null) temp(this, e);
}
```

对 Volatile.Read 的调用强迫 NewMail 在这个调用发生时读取，引用真的必须复制到 temp 变量中(编译器别想走捷径)。然后，temp 变量只有在不为 null 时才会被调用(invoke)。第 29 章“基元线程同步构造”将详细讨论 Volatile.Read 方法。

虽然最后一个版本很完美，是技术正确的版本，但版本 2 实际也是可以使用的，因为 JIT 编译器理解这个模式，知道自己不该将局部变量 temp“优化”掉。具体地说，微软的所有 JIT 编译器都“尊重”那些不会造成对堆内存的新的读取动作的不变量(invariant)。所以，在局部变量中缓存一个引用，可确保堆引用只被访问一次。这一点并未在文档中反映，理论上说将来可能改变，这正是为什么应该使用最后一个版本的原因。但实际上，Microsoft 的 JIT 编译器永远没有可能真的进行修改来破坏这个模式，否则太多的应用程序都会“遭殃”[①]。此外，事件主要在单线程的情形 (WPF 和 Windows Store 应用)中使用，所以线程安全不是问题。

另外，非常重点的一点在于，考虑到线程竞态条件[②]，方法有可能在从事件的委托链中移除之后得到调用(invoke)。

为方便起见，可以定义一个扩展方法(参见第 8 章“方法”)来封装这个线程安全逻辑。如下所示：

```
public static class EventArgExtensions {
   public static void Raise<TEventArgs>(this TEventArgs e,
      Object sender, ref EventHandler<TEventArgs> eventDelegate) {
```

[①] 这是 Microsoft 的 JIT 编译器团队的人告诉我的。

[②] 文档翻译成“争用状态”或“争用条件”。——译注

```
            // 出于线程安全的考虑，现在将对委托字段的引用复制到临时字段
            EventHandler<TEventArgs> temp = Volatile.Read(ref eventDelegate);

            // 任何方法登记了对事件的关注就通知它们
            if (temp != null) temp(sender, e);
        }
    }
```

现在可以像下面这样重写 OnNewMail 方法：

```
    protected virtual void OnNewMail(NewMailEventArgs e) {
        e.Raise(this, ref m_NewMail);
    }
```

以 MailManager 为基类的类可以自由地重写 OnNewMail 方法。这使派生类能控制事件的引发，以自己的方式处理新邮件。一般情况下，派生类会调用基类的 OnNewMail 方法，使登记的方法能收到通知。但是，派生类也可以决定不再允许事件被进一步转发(已登记的方法将不会收到事件通知)。

11.1.4 第四步：定义方法将输入转化为期望事件

类还必须有一个方法获取输入并转化为事件的引发。在 MailManager 的例子中，是调用 SimulateNewMail 方法来指出一封新的电子邮件已到达 MailManager：

```
    internal class MailManager {

        //第四步：定义一个方法，将输入转化为期望的事件
        public void SimulateNewMail(String from, String to, String subject) {

            // 构造一个对象来容纳想传给通知接收者的信息
            NewMailEventArgs e = new NewMailEventArgs(from, to, subject);

            // 调用虚方法通知对象事件已发生，
            // 如果没有类型重写该方法，我们的对象将通知事件的所有登记对象
            OnNewMail(e);
        }
    }
```

SimulateNewMail 接收关于邮件的信息并构造 NewMailEventArgs 对象，将邮件的信息传给它的构造器。然后调用 MailManager 自己的虚方法 OnNewMail 来正式通知 MailManager 对象收到了新的电子邮件。这通常会导致事件的引发，从而通知所有已登记的方法。(如前所述，以 MailManager 为基类的类可能重写这个行为。)

11.2 编译器如何实现事件

知道如何定义提供了事件成员的类之后，接着研究一下事件到底是什么，以及它是如何工作的。MailManager 类用一行代码定义了事件成员本身：

```
public event EventHandler<NewMailEventArgs> NewMail;
```

C#编译器在编译时，会把它转换为以下三个构造：

```
// 1. 一个被初始化为 null 的私有委托字段
private EventHandler<NewMailEventArgs> NewMail = null;

// 2. 一个公共 add_Xxx 方法(其中 Xxx 是事件名)
// 允许方法登记对事件的关注
public void add_NewMail(EventHandler<NewMailEventArgs> value) {
    // 通过循环和对 CompareExchange 的调用，可以
    // 以一种线程安全的方式向事件添加委托
    EventHandler<NewMailEventArgs> prevHandler;
    EventHandler<NewMailEventArgs> newMail = this.NewMail;
    do {
        prevHandler = newMail;
        EventHandler<NewMailEventArgs> newHandler =
            (EventHandler<NewMailEventArgs>) Delegate.Combine(prevHandler, value);
        newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
             ref this.NewMail, newHandler, prevHandler);
    } while (newMail != prevHandler);
}

// 3. 一个公共 remove_Xxx 方法(其中 Xxx 是事件名)
// 允许方法注销对事件的关注
public void remove_NewMail(EventHandler<NewMailEventArgs> value) {
    // 通过循环和对 CompareExchange 的调用，可以
    // 以一种线程安全的方式从事件中移除一个委托
    EventHandler<NewMailEventArgs> prevHandler;
    EventHandler<NewMailEventArgs> newMail = this.NewMail;
    do {
        prevHandler = newMail;
        EventHandler<NewMailEventArgs> newHandler =
           (EventHandler<NewMailEventArgs>) Delegate.Remove(prevHandler, value);
        newMail = Interlocked.CompareExchange<EventHandler<NewMailEventArgs>>(
           ref this.NewMail, newHandler, prevHandler);
    } while (newMail != prevHandler);
}
```

第一个构造是具有恰当委托类型的字段。该字段是对一个委托列表的头部的引用。事件发生时会通知这个列表中的委托。字段初始化为 null，表明暂无侦听者(listener)登记对该事件的关注。一旦有一个方法登记了对事件的关注，该字段就会引用 EventHandler<NewMailEventArgs> 委托的一个实例，后者可能引用更多的 EventHandler<NewMailEventArgs>委托。侦听者在登记对事件的关注时，只需将委托类型的一个实例添加到列表中。显然，注销(对事件的关注)意味着从列表中移除委托。

注意，尽管原始代码行将事件定义为 public，但委托字段(本例是 NewMail)始终是 private 的。将委托字段定义为 private，目的是防止类外部的代码不正确地操纵它。如果字段定义为 public，那么任何代码都能更改字段中的值，而且可能删除已登记了对事件的关注的

委托。

C#编译器生成的第二个构造是一个方法，允许其他对象登记对事件的关注。C#编译器在事件名(NewMail)之前附加 add_前缀，从而自动命名该方法。C#编译器还自动为方法生成代码。生成的代码总是调用 System.Delegate 的静态 Combine 方法，它将委托实例添加到委托列表中，返回新的列表头(地址)，并将这个地址存回字段。

C#编译器生成的第三个构造是一个方法，允许对象注销对事件的关注。同样地，C#编译器在事件名(NewMail)之前附加 remove_前缀，从而自动命名该方法。方法中的代码总是调用Delegate的静态Remove方法，将委托实例从委托列表中删除，返回新的列表头(地址)，并将这个地址存回字段。

警告：如果试图删除从未添加过的方法，那么 Delegate 的 Remove 方法在内部不做任何事情。也就是说，不会抛出任何异常，也不会显示任何警告；事件的方法集合保持不变。

注意：add 和 remove 方法以线程安全的一种模式更新值。该模式的详情将在 29.3.4 节“Interlocked Anything 模式”中讨论。

在本例中，add 和 remove 方法的可访问性都是 public。这是因为源代码将事件声明为 public。如果事件声明为 protected，编译器生成的 add 和 remove 方法也会被声明为 protected。因此，在类型中定义事件时，事件的可访问性决定了什么代码能登记和注销对事件的关注。但无论如何，只有类型本身才能直接访问私有的委托字段。事件成员也可声明为 static 或 virtual。在这种情况下，编译器生成的 add 和 remove 方法分别标记为 static 或 virtual。

除了生成上述三个构造，编译器还会在托管程序集的元数据中生成一个事件定义记录项。这个记录项包含了一些标志(flag)和基础委托类型(underlying delegate type)，还引用了 add 和 remove 访问器方法。这些信息的作用很简单，就是建立“事件”的抽象概念和它的访问器方法之间的联系。编译器和其他工具可以利用这些元数据信息，并且可以通过 System.Reflection.EventInfo 类获取这些信息。但是，CLR 本身并不使用这些元数据信息，它在运行时只需要访问器方法。

11.3 设计侦听事件的类型

最难的部分已经完成了，接下来是一些较为简单的事情。本节将演示如何定义一个类型来

使用另一个类型提供的事件。先来看看 Fax 类型的代码：

```
internal sealed class Fax {
    // 将 MailManager 对象传给构造器
    public Fax(MailManager mm) {

        // 构造 EventHandler<NewMailEventArgs>委托的一个实例，
        // 使它引用我们的 FaxMsg 回调方法。
        // 向 MailManager 的 NewMail 事件登记我们的回调方法。
        mm.NewMail += FaxMsg;
    }

    // 新电子邮件到达时，MailManager 将调用这个方法
    private void FaxMsg(Object sender, NewMailEventArgs e) {

        // 'sender'表示 MailManager 对象，便于将信息传回给它
        // 'e'表示 MailManager 对象想传给我们的附加事件信息

        // 以下代码正常情况下应该传真电子邮件，
        // 但这个测试性的实现只是在控制台上显示邮件
        Console.WriteLine("Faxing mail message:");
        Console.WriteLine(" From={0}, To={1}, Subject={2}",
            e.From, e.To, e.Subject);
    }

    // 执行这个方法，Fax 对象将向 NewMail 事件注销自己对它的关注，
    // 以后不再接收通知
    public void Unregister(MailManager mm) {
        // 向 MailManager 的 NewMail 事件注销自己对这个事件的关注
        mm.NewMail -= FaxMsg;
    }
}
```

电子邮件应用程序初始化时首先构造一个 MailManager 对象，并将对该对象的引用保存到变量中。然后构造 Fax 对象，并将 MailManager 对象引用作为实参传递。在 Fax 构造器中，Fax 对象使用 C# +=操作符登记它对 MailManager 的 NewMail 事件的关注：

```
mm.NewMail += FaxMsg;
```

C#编译器内建了对事件的支持，会将+=操作符翻译成以下代码来添加对象对事件的关注：

```
mm.add_NewMail(new EventHandler<NewMailEventArgs>(this.FaxMsg));
```

C#编译器生成的代码构造一个 EventHandler<NewMailEventArgs>委托对象，其中包装了 Fax 类的 FaxMsg 方法。接着，C#编译器调用 MailManager 类的 add_NewMail 方法，向它传递新的委托对象。为了对此进行验证，可以编译代码并用 ILDasm.exe 这样的工具查看 IL 代码。

即使使用的编程语言不直接支持事件，也可以显式调用 add 访问器方法向事件登记委托。两者效果一样，只是后者的源代码看起来没那么优雅。两者最终都是用 add 访问器将委托

添加到事件的委托列表中，从而完成委托向事件的登记。

MailManager 对象引发事件时，Fax 对象的 FaxMsg 方法会被调用。调用这个方法时，会传递 MailManager 对象引用作为它的第一个参数，即 sender。该参数大多数时候会被忽略。但是，如果 Fax 对象希望在响应事件时访问 MailManager 对象的成员，它就能派上用场了。第二个参数是 NewMailEventArgs 对象引用。对象中包含 MailManager 和 NewMailEventArgs 的设计者认为对事件接收者来说有用的附加信息。

FaxMsg 方法可以从 NewMailEventArgs 对象中轻松访问邮件的发件人、收件人以及主题。真实的 Fax 对象应将这些信息传真到某处。本例只是在控制台窗口显示。

对象不再希望接收事件通知时，应注销对事件的关注。例如，如果不再希望将电子邮件转发到一台传真机，Fax 对象就应该注销它对 NewMail 事件的关注。对象只要向事件登记了它的一个方法，便不能被垃圾回收。所以，如果你的类型要实现 IDisposable 的 Dispose 方法，就应该在实现中注销对所有事件的关注。IDisposable 的详情参见第 21 章“托管堆和垃圾回收”。

Fax 的 Unregister 方法示范了如何注销对事件的关注。该方法和 Fax 构造器中的代码十分相似。唯一区别是使用-=而不是+=。C#编译器看到代码使用-=操作符向事件注销委托时，会生成对事件的 remove 方法的调用：

```
mm.remove_NewMail(new EventHandler<NewMailEventArgs>(FaxMsg));
```

和+=操作符一样，即使编程语言不直接支持事件，也可显式调用 remove 访问器方法向事件注销委托。remove 方法为了向事件注销委托，需要扫描委托列表来寻找一个恰当的委托(其中包装的方法和传递的方法相同)。一旦找到匹配，现有委托就会从事件的委托列表中删除。没有找到也不会报错，列表不发生任何变动。

顺便说一下，C#要求代码使用+=和-=操作符在列表中增删委托。如果显式调用 add 和 remove 方法，C#编译器会报告以下错误消息：CS0571: 无法显式调用运算符或访问器。

11.4 显式实现事件

System.Windows.Forms.Control 类型定义了大约 70 个事件。假如 Control 类型在实现事件时，允许编译器隐式生成 add 和 remove 访问器方法以及委托字段，那么每个 Control 对象仅为事件就要准备 70 个委托字段！由于大多数程序员只关心少数几个事件，所以每个从 Control 派生类型创建的对象都会浪费大量内存。顺便说一下，ASP.NET 的 System.Web.UI.Control 类型和 WPF 的 System.Windows.UIElement 类型也提供了大多数程序员都用不上的大量事件。

本节将讨论 C#编译器如何允许类的开发人员显式实现一个事件，使开发人员能够控制 add 和 remove 方法处理回调委托的方式。我要演示如何通过显式实现事件来高效率地实现提供了大量事件的类，但肯定还有其他情形也需要显式实现事件。

为了高效率存储事件委托，公开了事件的每个对象都要维护一个集合(通常是字典)。集合将某种形式的事件标识符作为键(key)，将委托列表作为值(value)。新对象构造时，这个集合是空白的。登记对一个事件的关注时，会在集合中查找事件的标识符。如果事件标识符已在其中，新委托就和这个事件的委托列表合并。如果事件标识符不在集合中，就添加事件标识符和委托。

对象需要引发事件时，会在集合中查找事件标识符。如果集合中没有找到事件标识符，表明还没有任何对象登记对这个事件的关注，所以没有任何委托需要回调。如果事件标识符在集合中，就调用与它关联的委托列表。具体怎么实现这个设计模式，是定义事件的那个类型的开发人员的责任；使用类型的开发人员不知道事件在内部如何实现。

下例展示了如何完成这个模式。首先实现一个 `EventSet` 类，它代表一个集合，其中包含事件以及每个事件的委托列表。

```
using System;
using System.Collections.Generic;
using System.Threading;

// 这个类的目的是在使用 EventSet 时，提供
// 多一点的类型安全性和代码可维护性
public sealed class EventKey { }

public sealed class EventSet {
    // 该私有字典用于维护 EventKey -> Delegate 映射
private readonly Dictionary<EventKey, Delegate> m_events =
    new Dictionary<EventKey, Delegate>();

// 添加 EventKey -> Delegate 映射(如果 EventKey 不存在)，
// 或者将委托和现有的 EventKey 合并
public void Add(EventKey eventKey, Delegate handler) {
    Monitor.Enter(m_events);
    Delegate d;
    m_events.TryGetValue(eventKey, out d);
    m_events[eventKey] = Delegate.Combine(d, handler);
    Monitor.Exit(m_events);
}

// 从 EventKey(如果它存在)删除委托，并且
// 在删除最后一个委托时删除 EventKey -> Delegate 映射
public void Remove(EventKey eventKey, Delegate handler) {
    Monitor.Enter(m_events);
    // 调用 TryGetValue，确保在尝试从集合中删除不存在的 EventKey 时不会抛出异常
    Delegate d;
    if (m_events.TryGetValue(eventKey, out d)) {
        d = Delegate.Remove(d, handler);

        // 如果还有委托，就设置新的头部(地址)，否则删除 EventKey
        if (d != null) m_events[eventKey] = d;
```

```
            else m_events.Remove(eventKey);
        }
        Monitor.Exit(m_events);
    }

    // 为指定的 EventKey 引发事件
    public void Raise(EventKey eventKey, Object sender, EventArgs e) {
        // 如果 EventKey 不在集合中，不抛出异常
        Delegate d;
        Monitor.Enter(m_events);
        m_events.TryGetValue(eventKey, out d);
        Monitor.Exit(m_events);

        if (d != null) {
            // 由于字典可能包含几个不同的委托类型，
            // 所以无法在编译时构造一个类型安全的委托调用。
            // 因此，我调用 System.Delegate 类型的 DynamicInvoke
            // 方法，以一个对象数组的形式向它传递回调方法的参数。
            // 在内部，DynamicInvoke 会向调用的回调方法查证参数的
            // 类型安全性，并调用方法。
            // 如果存在类型不匹配的情况，DynamicInvoke 会抛出异常。
            d.DynamicInvoke(new Object[] { sender, e });
       }
    }
```

接着定义一个类来使用 EventSet 类。在这个类中，一个字段引用了一个 EventSet 对象，而且这个类的每个事件都是显式实现的，使每个事件的 add 方法都将指定的回调委托存储到 EventSet 对象中，而且每个事件的 remove 方法都删除指定的回调委托(如果找得到的话)。

```
    using System;

    // 为这个事件定义从 EventArgs 派生的类型
    public class FooEventArgs : EventArgs { }

    public class TypeWithLotsOfEvents {
        // 定义私有实例字段来引用集合
        // 集合用于管理一组“事件/委托”对
        // 注意：EventSet 类型不是 FCL 的一部分，它是我自己的类型
        private readonly EventSet m_eventSet = new EventSet();

        // 受保护的属性使派生类型能访问集合
        protected EventSet EventSet { get { return m_eventSet; } }

        #region 用于支持 Foo 事件的代码(为附加的事件重复这个模式)
        // 定义 Foo 事件必要的成员
        // 2a. 构造一个静态只读对象来标识这个事件.
        // 每个对象都有自己的哈希码，以便在对象的集合中查找这个事件的委托链表
        protected static readonly EventKey s_fooEventKey = new EventKey();
```

```
    // 2b. 定义事件的访问器方法，用于在集合中增删委托
    public event EventHandler<FooEventArgs> Foo {
        add    { m_eventSet.Add(s_fooEventKey, value); }
        remove { m_eventSet.Remove(s_fooEventKey, value); }
    }

    // 2c. 为这个事件定义受保护的虚方法 OnFoo
    protected virtual void OnFoo(FooEventArgs e) {
        m_eventSet.Raise(s_fooEventKey, this, e);
    }

    // 2d. 定义将输入转换成这个事件的方法
    public void SimulateFoo() {OnFoo(new FooEventArgs());}
    #endregion
}
```

使用 TypeWithLotsOfEvents 类型的代码不知道事件是由编译器隐式实现，还是由开发人员显式实现。它们只需用标准的语法向事件登记即可。以下代码进行了演示：

```
public sealed class Program {
    public static void Main() {
        TypeWithLotsOfEvents twle = new TypeWithLotsOfEvents();

        // 添加一个回调
        twle.Foo += HandleFooEvent;

        // 证明确实可行
        twle.SimulateFoo();
    }

    private static void HandleFooEvent(object sender, FooEventArgs e) {
        Console.WriteLine("Handling Foo Event here...");
    }
}
```

第 12 章 泛型

本章内容：

- FCL 中的泛型
- 泛型基础结构
- 泛型接口
- 泛型委托
- 委托和接口的逆变和协变泛型类型实参
- 泛型方法
- 泛型和其他成员
- 可验证性和约束

熟悉面向对象编程(OOP)的开发人员都深谙这种编程方式的好处。其中一个好处是“代码重用”，它极大提高了开发效率。也就是说，可以派生出一个类，让它继承基类的所有能力。派生类只需重写虚方法，或添加一些新方法，就可定制派生类的行为，使之满足开发人员的需求。**泛型**(generic)是 CLR 和编程语言提供的一种特殊机制，它支持另一种形式的代码重用，即“算法重用”。

简单地说，开发人员先定义好算法，比如排序、搜索、交换、比较或者转换等。但是，定义算法的开发人员并不设定该算法具体要操作的数据类型；该算法可广泛地应用于不同类型的对象。然后，另一个开发人员只要指定了算法要操作的具体数据类型，就可以开始使用这个算法了。例如，一个排序算法可以操作 `Int32` 和 `String` 等类型的对象，而一个比较算法可以操作 `DateTime` 和 `Version` 等类型的对象。

大多数算法都封装在一个类型中，CLR 允许创建泛型引用类型和泛型值类型，但不允许创建泛型枚举类型。此外，CLR 还允许创建泛型接口和泛型委托。偶尔也有方法封装了有用的算法，所以 CLR 允许在引用类型、值类型或接口中定义泛型方法。

先来看一个简单的例子。Framework 类库(Framework Class Library，FCL)定义了一个泛型列表算法，它知道如何管理对象集合。泛型算法没有设定对象的数据类型。要在使用这个泛型列表算法时指定具体数据类型。

封装了泛型列表算法的 FCL 类称为 `List<T>`(读作 List of Tee)。这个类是在 `System.Collections.Generic` 命名空间中定义的。下面展示了类定义 (进行了大幅简化)：

```
[Serializable]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
  IList, ICollection, IEnumerable {

  public List();
  public void Add(T item);
  public Int32 BinarySearch(T item);
  public void Clear();
  public Boolean Contains(T item);
  public Int32 IndexOf(T item);
  public Boolean Remove(T item);
  public void Sort();
  public void Sort(IComparer<T> comparer);
  public void Sort(Comparison<T> comparison);
  public T[] ToArray();

  public Int32 Count { get; }
  public T this[Int32 index] { get; set; }
}
```

泛型 List 类的设计者紧接在类名后添加了一个<T>，表明它操作的是一个未指定的数据类型。定义泛型类型或方法时，为类型指定的任何变量(比如 T)都称为**类型参数**(type parameter)。T 是变量名，源代码中凡是能使用数据类型的地方都能使用 T。例如，在 List 类定义中，T 被用于方法参数(Add 方法接受一个 T 类型的参数)和返回值(ToArray 方法返回 T 类型的一维数组)。另一个例子是索引器方法(在 C#中称为 this)。索引器有一个 get 访问器方法，它返回 T 类型的值；一个 set 访问器方法，它接收 T 类型的参数。由于凡是能指定一个数据类型的地方都能使用 T 变量，所以在方法内部定义一个局部变量时，或者在类型中定义字段时，也可以使用 T。

注意：按照微软的设计原则，泛型参数变量要么称为 T，要么至少以大写 T 开头(如 TKey 和 TValue)。大写 T 代表类型(Type)，就像大写 I 代表接口(Interface)一样，比如 IComparable。

定义好泛型 List<T>类型之后，其他开发人员为了使用这个泛型算法，要指定由算法操作的具体数据类型。使用泛型类型或方法时指定的具体数据类型称为**类型实参**(type argument)。例如，开发人员可以指定一个 DateTime 类型实参来使用 List 算法。以下代码对此进行了演示：

```
private static void SomeMethod() {
  // 构造一个 List 来操作 DateTime 对象
  List<DateTime> dtList = new List<DateTime>();

  // 向列表添加一个 DateTime 对象
  dtList.Add(DateTime.Now);                // 不进行装箱

  // 向列表添加另一个 DateTime 对象
```

```
    dtList.Add(DateTime.MinValue);          // 不进行装箱

    // 尝试向列表中添加一个 String 对象
    dtList.Add("1/1/2004");                 // 编译时错误

    // 从列表提取一个 DateTime 对象
    DateTime dt = dtList[0];                // 不需要转型
  }
```

从以上代码可以看出，泛型为开发人员提供了以下优势。

- **源代码保护**

  使用泛型算法的开发人员不需要访问算法的源代码。然而，使用 C++模板时，算法源代码必须提供给准备使用算法的用户。

- **类型安全**

  将泛型算法应用于一个具体的类型时，编译器和 CLR 能理解开发人员的意图，并保证只有与指定数据类型兼容的对象才能用于算法。试图使用不兼容类型的对象会造成编译时错误，或在运行时抛出异常。在上例中，试图将 String 对象传给 Add 方法造成编译器报错。

- **更清晰的代码**

  由于编译器强制类型安全性，所以减少了源代码中必须进行的强制类型转换次数，使代码更容易编写和维护。在 SomeMethod 的最后一行，开发人员不需要进行(DateTime)强制类型转换，就能将索引器的结果(查询在索引位置 0 处的元素)存储到 dt 变量中。

- **更佳的性能**

  没有泛型的时候，要想定义常规化的算法，它的所有成员都必须定义成操作 Object 数据类型。要用这个算法来操作值类型的实例，CLR 必须在调用算法的成员之前对值类型实例进行装箱。正如第 5 章“基元类型、引用类型和值类型”讨论的那样，装箱造成在托管堆上进行内存分配，造成更频繁的垃圾回收，从而损害应用程序的性能。由于现在能创建一个泛型算法来操作一种具体的值类型，所以值类型的实例能以传值方式传递，CLR 不再需要执行任何装箱操作。此外，由于不再需要进行强制类型转换(参见上一条)，所以 CLR 无需验证这种转型是否类型安全，这同样提高了代码的运行速度。

为了理解泛型的性能优势，我写了一个程序来比较泛型 List 算法和 FCL 的非泛型 ArrayList 算法的性能。我打算同时使用值类型的对象和引用类型的对象来测试这两个算法的性能。下面是程序本身：

```
using System;
using System.Collections;
using System.Collections.Generic;
```

```
using System.Diagnostics;

public static class Program {
   public static void Main() {
      ValueTypePerfTest();
      ReferenceTypePerfTest();
   }

   private static void ValueTypePerfTest() {
      const Int32 count = 100000000;


      using (new OperationTimer("List<Int32>")) {
         List<Int32> l = new List<Int32>();
         for (Int32 n = 0; n < count; n++) {
            l.Add(n);       // 不发生装箱
            Int32 x = l[n]; // 不发生拆箱
         }
         l = null; //确保进行垃圾回收
      }

      using (new OperationTimer("ArrayList of Int32")) {
         ArrayList a = new ArrayList();
         for (Int32 n = 0; n < count; n++) {
            a.Add(n);                    // 发生装箱
            Int32 x = (Int32) a[n];      // 发生拆箱
         }
         a = null;                       // 确保进行垃圾回收
      }
   }

   private static void ReferenceTypePerfTest() {
      const Int32 count = 100000000;

      using (new OperationTimer("List<String>")) {
         List<String> l = new List<String>();
         for (Int32 n = 0; n < count; n++) {
            l.Add("X");                  // 复制引用
            String x = l[n];             // 复制引用
         }
         l = null;                       // 确保进行垃圾回收
      }

      using (new OperationTimer("ArrayList of String")) {
         ArrayList a = new ArrayList();
         for (Int32 n = 0; n < count; n++) {
            a.Add("X");                  // 复制引用
            String x = (String) a[n];    // 检查强制类型转换 & 复制引用
         }
         a = null;  // 确保进行垃圾回收
      }
```

```
    }
}

// 这个类用于进行运算性能计时
internal sealed class OperationTimer : IDisposable {
    private Stopwatch m_stopwatch;
    private String m_text;
    private Int32 m_collectionCount;

    public OperationTimer(String text) {
        PrepareForOperation();

        m_text = text;
        m_collectionCount = GC.CollectionCount(0);

        // 这应该是方法的最后一个语句，从而最大程度保证计时的准确性
        m_stopwatch = Stopwatch.StartNew();
    }

    public void Dispose() {
        Console.WriteLine("{0} (GCs={1,3}) {2}", (m_stopwatch.Elapsed),
            GC.CollectionCount(0) - m_collectionCount, m_text);
    }

    private static void PrepareForOperation() {
        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();
    }
}
```

在我的机器上编译并运行这个程序 (打开优化开关)，得到以下输出：

```
00:00:01.2848275 (GCs=  6) List<Int32>
00:00:21.3052240 (GCs=221) ArrayList of Int32
00:00:01.6353842 (GCs=  6) List<String>
00:00:01.4143716 (GCs=  0) ArrayList of String
```

这证明在操作 `Int32` 类型时，泛型 `List` 算法比非泛型 `ArrayList` 算法快得多。1.3 秒和 21 秒，约 16 倍的差异！此外，用 `ArrayList` 操作值类型(`Int32`)会造成大量装箱，最终要进行 221 次垃圾回收。对应地，`List` 算法只需进行 6 次。

不过，用引用类型来测试，差异就没那么明显了，用时非常接近。所以，泛型 `List` 算法此时表面上没什么优势。但要注意的是，泛型算法的优势还包括更清晰的代码和编译时的类型安全。所以，虽然性能提升不是很明显，但泛型算法的其他优势也不容忽视。

注意：应该意识到，首次为一个特定的数据类型调用泛型方法时，CLR 都会为该方法生成本机代码。这会增大应用程序的工作集(working set)大小，从而损害性能。12.2 节

“泛型基础结构”将进一步探讨这个问题。

12.1 FCL 中的泛型

泛型最明显的应用就是集合类。FCL 在 System.Collections.Generic 和 System.Collections.ObjectModel 命名空间中提供了多个泛型集合类。System.Collections.Concurrent 命名空间则提供了线程安全的泛型集合类。Microsoft 建议使用泛型集合类，不建议使用非泛型集合类。这是出于几方面的考虑。首先，使用非泛型集合类，无法像使用泛型集合类那样获得类型安全性、更清晰的代码以及更佳的性能。其次，泛型类具有比非泛型类更好的对象模型。例如，虚方法数量显著变少，性能更好。另外，泛型集合类增添了一些新成员，为开发人员提供了新的功能。

集合类实现了许多接口，放入集合中的对象可通过实现接口来执行排序和搜索等操作。FCL 包含许多泛型接口定义，所以使用接口时也能享受到泛型带来的好处。常用接口在 System.Collections.Generic 命名空间中提供。

新的泛型接口不是为了替代旧的非泛型接口。许多时候两者都要使用(为了向后兼容)。例如，如果 List<T>类只实现了 IList<T>接口，代码就不能将一个 List<DateTime>对象当作一个 IList 来处理。

还要注意，System.Array 类(所有数组类型的基类)提供了大量静态泛型方法，比如 AsReadOnly，BinarySearch，ConvertAll，Exists，Find，FindAll，FindIndex，FindLast，FindLastIndex，ForEach，IndexOf，LastIndexOf，Resize，Sort 和 TrueForAll 等。下面展示了部分方法：

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable,
  IStructuralComparable, IStructuralEquatable {

  public static void Sort<T>(T[] array);
  public static void Sort<T>(T[] array, IComparer<T> comparer);

  public static Int32 BinarySearch<T>(T[] array, T value);
  public static Int32 BinarySearch<T>(T[] array, T value,
     IComparer<T> comparer);
  ...
}
```

以下代码展示了如何使用其中的一些方法：

```
public static void Main() {
   // 创建并初始化字节数组
   Byte[] byteArray = new Byte[] { 5, 1, 4, 2, 3 };

   // 调用 Byte[]排序算法
   Array.Sort<Byte>(byteArray);
```

```
    // 调用Byte[]二分搜索算法
    Int32 i = Array.BinarySearch<Byte>(byteArray, 1);
    Console.WriteLine(i); // 显示"0"
}
```

12.2 泛型基础结构

泛型在 CLR 2.0 中加入。为了在 CLR 中加入泛型，许多人花费了大量时间来完成这个大型任务。具体地说，为了使泛型能够工作，微软必须完成以下工作。

- 创建新的 IL 指令，使之能够识别类型实参。
- 修改现有元数据表的格式，以便表示具有泛型参数的类型名称和方法。
- 修改各种编程语言(C#，Microsoft Visual Basic .NET 等)来支持新语法，允许开发人员定义和引用泛型类型和方法。
- 修改编译器，使之能生成新的 IL 指令和修改的元数据格式。
- 修改 JIT 编译器，以便处理新的支持类型实参的 IL 指令来生成正确的本机代码。
- 创建新的反射成员，使开发人员能查询类型和成员，以判断它们是否具有泛型参数。另外，还必须定义新的反射成员，使开发人员能在运行时创建泛型类型和方法定义。
- 修改调试器以显示和操纵泛型类型、成员、字段以及局部变量。
- 修改 Microsoft Visual Studio 的“智能感知”(IntelliSense)功能。将泛型类型或方法应用于特定数据类型时能显示成员的原型。

现在让我们花些时间讨论 CLR 内部如何处理泛型。这一部分的知识可能影响你构建和设计泛型算法的方式。另外，还可能影响你是否使用一个现有泛型算法的决策。

12.2.1 开放类型和封闭类型

贯穿全书，我讨论了 CLR 如何为应用程序使用的各种类型创建称为**类型对象**(type object)的内部数据结构。具有泛型类型参数的类型仍然是类型，CLR 同样会为它创建内部的类型对象。这一点适合引用类型(类)、值类型(结构)、接口类型和委托类型。然而，具有泛型类型参数的类型称为**开放类型**，CLR 禁止构造开放类型的任何实例。这类似于 CLR 禁止构造接口类型的实例。

代码在引用泛型类型时可以指定一组泛型类型实参。为所有类型参数都传递了实际的数据类型，类型就成为**封闭类型**。CLR 允许构造封闭类型的实例。然而，代码引用泛型类型的时候，可能留下一些泛型类型实参未指定。这会在 CLR 中创建新的开放类型对象，而且不能创建该类型的实例。以下代码更清楚地说明了这一点：

```
using System;
using System.Collections.Generic;

// 一个部分指定的开放类型
internal sealed class DictionaryStringKey<TValue> :
   Dictionary<String, TValue> {
}

public static class Program {
   public static void Main() {
      Object o = null;

      // Dictionary<,>是开放类型，有 2 个类型参数
      Type t = typeof(Dictionary<,>);

      // 尝试创建该类型的实例(失败)
      o = CreateInstance(t);
      Console.WriteLine();

      // DictionaryStringKey<>是开放类型，有一个类型参数
      t = typeof(DictionaryStringKey<>);

      // 尝试创建该类型的实例(失败)
      o = CreateInstance(t);
      Console.WriteLine();

      // DictionaryStringKey<Guid>是封闭类型
      t = typeof(DictionaryStringKey<Guid>);

      // 尝试创建该类型的一个实例(成功)
      o = CreateInstance(t);

      // 证明它确实能够工作
      Console.WriteLine(" 对象类型 = " + o.GetType());
   }
```

```
        private static Object CreateInstance(Type t) {
            Object o = null;
            try {
                o = Activator.CreateInstance(t);
                Console.Write("已创建 {0} 的实例。", t.ToString());
            }
            catch (ArgumentException e) {
                Console.WriteLine(e.Message);
            }
            return o;
        }
    }
```

编译并运行上述代码得到以下输出：

```
无法创建 System.Collections.Generic.Dictionary`2[TKey,TValue] 的实例，
因为 Type.ContainsGenericParameters 为 True。

无法创建 DictionaryStringKey`1[TValue] 的实例，
因为 Type.ContainsGenericParameters 为 True。

已创建 DictionaryStringKey`1[System.Guid] 的实例。
对象类型 = DictionaryStringKey`1[System.Guid]
```

可以看出，Activator 的 CreateInstance 方法会在试图构造开放类型的实例时抛出 ArgumentException 异常。注意，异常的字符串消息指明类型中仍然含有一些泛型参数。

从输出可以看出，类型名以“`”字符和一个数字结尾。数字代表类型的元数，也就是类型要求的类型参数个数。例如，Dictionary 类的元数为 2，要求为 TKey 和 TValue 这两个类型参数指定具体类型。DictionaryStringKey 类的元数为 1，只要求为 TValue 指定具体类型。

还要注意，CLR 会在类型对象内部分配类型的静态字段(本书第 4 章“类型基础”对此进行了讨论)。因此，每个封闭类型都有自己的静态字段。换言之，假如 List<T>定义了任何静态字段，这些字段不会在一个 List<DateTime>和一个 List<String>之间共享；每个封闭类型对象都有自己的静态字段。另外，假如泛型类型定义了静态构造器(参见第 8 章“方法”)，那么针对每个封闭类型，这个构造器都会执行一次。泛型类型定义静态构造器的目的是保证传递的类型实参满足特定条件。例如，我们可以像下面这样定义只能处理枚举类型的泛型类型：

```
internal sealed class GenericTypeThatRequiresAnEnum<T> {
    static GenericTypeThatRequiresAnEnum() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T 必须是某个枚举类型");
        }
    }
}
```

CLR 提供了名为**约束**的功能，可以更好地指定有效的类型实参。本章稍后会详细讨论。遗憾的是，约束无法将类型实参限制为“仅枚举类型”。正是因为这个原因，所以上例需要用静态构造器来保证类型是一个枚举类型。

12.2.2 泛型类型和继承

泛型类型仍然是类型，所以能从其他任何类型派生。使用泛型类型并指定类型实参时，实际是在 CLR 中定义一个新的类型对象，新的类型对象从泛型类型派生自的那个类型派生。换言之，由于List<T>从Object派生，所以List<String>和List<Guid>也从Object派生。类似地，由于 DictionaryStringKey<TValue>从 Dictionary<String，TValue>派生，所以DictionaryStringKey<Guid>也从 Dictionary<String，Guid>派生。指定类型实参不影响继承层次结构——理解这一点，有助于你判断哪些强制类型转换是允许的，哪些不允许。

假定像下面这样定义一个链表节点类：

```
internal sealed class Node<T> {
   public T m_data;
   public Node<T> m_next;

   public Node(T data) : this(data, null) {
   }

   public Node(T data, Node<T> next) {
      m_data = data; m_next = next;
   }

   public override String ToString() {
      return m_data.ToString() +
         ((m_next != null) ? m_next.ToString() : String.Empty);
   }
}
```

那么可以写代码来构造链表，例如：

```
private static void SameDataLinkedList() {
   Node<Char> head = new Node<Char>('C');
   head = new Node<Char>('B', head);
   head = new Node<Char>('A', head);
   Console.WriteLine(head.ToString());  // 显示"ABC"
}
```

在这个Node类中，对于m_next字段引用的另一个节点来说，它的m_data字段必须包含相同的数据类型。这意味着在链表包含的节点中，所有数据项都必须具有相同的类型(或派生类型)。例如，不能使用Node类来创建这样一个链表：其中一个元素包含Char值，另一个包含DateTime值，另一个元素则包含String值。当然，如果到处都用Node<Object>，那么确实可以做到，但会丧失编译时类型安全性，而且值类型会被装箱。

所以，更好的办法是定义非泛型 Node 基类，再定义泛型 TypedNode 类(用 Node 类作为基

类)。这样就可以创建一个链表，其中每个节点都可以是一种具体的数据类型(不能是Object)，同时获得编译时的类型安全性，并防止值类型装箱。下面是新的类型定义：

```
internal class Node {
   protected Node m_next;

   public Node(Node next) {
      m_next = next;
   }
}

internal sealed class TypedNode<T> : Node {
   public T m_data;

   public TypedNode(T data) : this(data, null) {
   }

   public TypedNode(T data, Node next) : base(next) {
      m_data = data;
   }

   public override String ToString() {
      return m_data.ToString() +
         ((m_next != null) ? m_next.ToString() : String.Empty);
   }
}
```

现在可以写代码来创建一个链表，其中每个节点都是不同的数据类型。例如：

```
private static void DifferentDataLinkedList() {
   Node head = new TypedNode<Char>('.');
   head = new TypedNode<DateTime>(DateTime.Now, head);
   head = new TypedNode<String>("Today is ", head);
   Console.WriteLine(head.ToString());
}
```

12.2.3 泛型类型同一性

泛型语法有时会将开发人员弄糊涂，因为源代码中可能散布着大量“<”和“>”符号，这有损可读性。为了对语法进行增强，有的开发人员定义了一个新的非泛型类类型，它从一个泛型类型派生，并指定了所有类型实参。例如，为了简化下面这样的代码：

```
List<DateTime> dtl = new List<DateTime>();
```

一些开发人员可能首先定义下面这样的类：

```
internal sealed class DateTimeList : List<DateTime> {
    // 这里无需放入任何代码!
}
```

然后就可以简化创建列表的代码(没有了“<”和“>”符号)：

```
DateTimeList dt1 = new DateTimeList();
```

这样做表面上是方便了(尤其是要为参数、局部变量和字段使用新类型的时候)，但是，绝对不要单纯出于增强源码可读性的目的来定义一个新类。这样会丧失类型同一性(identity)和相等性(equivalence)，如以下代码所示：

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

上述代码运行时，sameType 会被初始化为 false，因为比较的是两个不同类型的对象。这也意味着如果方法的原型接受一个DateTimeList，那么不可以将一个List<DateTime>传给它。然而，如果方法的原型接受一个 List<DateTime>，那么可以将一个 DateTimeList 传给它，因为 DateTimeList 从 List<DateTime>派生。开发人员很容易被所有这一切搞糊涂。

幸好，C#允许使用简化的语法来引用泛型封闭类型，同时不会影响类型的相等性。这个语法要求在源文件顶部使用传统的 using 指令，例如：

```
using DateTimeList = System.Collections.Generic.List<System.DateTime>;
```

using 指令实际定义的是名为 DateTimeList 的符号。代码编译时，编译器将代码中出现的所有DateTimeList 替换成System.Collections.Generic.List<System.DateTime>。这样就允许开发人员使用简化的语法，同时不影响代码的实际含义。所以，类型的同一性和相等性得到了维持。现在，执行下面这行代码时，sameType 会被初始化为 true：

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

另外，可以利用 C#的“隐式类型局部变量”功能，让编译器根据表达式的类型来推断方法的局部变量的类型：

```
using System;
using System.Collections.Generic;
...
internal sealed class SomeType {
    private static void SomeMethod () {

        // 编译器推断出 dtl 的类型
        // 是 System.Collections.Generic.List<System.DateTime>
        var dtl = new List<DateTime>();
        ...
    }
}
```

12.2.4 代码爆炸

使用泛型类型参数的方法在进行 JIT 编译时，CLR 获取方法的 IL，用指定的类型实参替换，然后创建恰当的本机代码(这些代码为操作指定的数据类型“量身定制”)。这正是你希望的，也是泛型的重要特点。但它有一个缺点：CLR 要为每种不同的方法/类型组合生成本机代码。我们将这个现象称为**代码爆炸**。它可能造成应用程序的工作集显著增大，从而损害性能。

幸好，CLR 内建了一些优化措施能缓解代码爆炸。首先，假如为特定的类型实参调用了一个方法，以后再用相同的类型实参调用这个方法，CLR 只会为这个方法/类型组合编译一次代码。所以，如果一个程序集使用 `List<DateTime>`，一个完全不同的程序集(加载到同一个 AppDomain 中)也使用 `List<DateTime>`，那么 CLR 只为 `List<DateTime>`编译一次方法。这样就显著缓解了代码爆炸。

CLR 还有另一个优化，它认为所有引用类型实参都完全相同，所以代码能够共享。例如，CLR 为 `List<String>`的方法编译的代码可直接用于 `List<Stream>`的方法，因为 `String` 和 `Stream` 均为引用类型。事实上，对于任何引用类型，都会使用相同的代码。CLR 之所以能执行这个优化，是因为所有引用类型的实参或变量实际只是指向堆上对象的指针(32 位 Windows 系统上是 32 位指针；64 位 Windows 系统上是 64 位指针)，而所有对象指针都以相同方式操纵。

但是，假如某个类型实参是值类型，CLR 就必须专门为那个值类型生成本机代码。这是因为值类型的大小不定。即使两个值类型大小一样(比如 `Int32` 和 `UInt32`，两者都是 32 位)，CLR 仍然无法共享代码，因为可能要用不同的本机 CPU 指令来操纵这些值。

12.3 泛型接口

显然，泛型的主要作用就是定义泛型的引用类型和值类型。然而，对泛型接口的支持对

CLR 来说也很重要。没有泛型接口，每次用非泛型接口(如 `IComparable`)来操纵值类型都会发生装箱，而且会失去编译时的类型安全性。这将严重制约泛型类型的应用范围。因此，CLR 提供了对泛型接口的支持。引用类型或值类型可以指定类型实参，从而实现一个泛型接口。也可以保持类型实参的未指定状态来实现泛型接口。下面来看一些例子。

以下泛型接口定义是 FCL 的一部分(在 System.Collections.Generic 命名空间中)：

```
public interface IEnumerator<T> : IDisposable, IEnumerator {
   T Current { get; }
}
```

下面的示例类型实现了上述泛型接口，而且指定了类型实参。注意，Triangle 对象可以枚举一组 Point 对象(三角形的顶点)。还要注意，Current 属性具有 Point 数据类型：

```
internal sealed class Triangle : IEnumerator<Point> {
   private Point[] m_vertices;

   // IEnumerator<Point>的 Current 属性的类型是 Point
    public Point Current { get { ... } }

    ...
}
```

下例实现了相同的泛型接口，但保持类型实参的未指定状态：

```
internal sealed class ArrayEnumerator<T> : IEnumerator<T> {
   private T[] m_array;

   // IEnumerator<T>的 Current 属性的类型是 T
   public T Current { get { ... } }

   ...
}
```

注意，ArrayEnumerator 对象可以枚举一组 T 对象(T 未指定，允许使用 ArrayEnumerator 泛型类型的代码以后再为 T 指定具体类型)。还要注意，Current 属性现在具有未指定的数据类型 T。第 13 章“接口”将更深入地讨论泛型接口。

12.4 泛型委托

CLR 支持泛型委托，目的是保证任何类型的对象都能以类型安全的方式传给回调方法。此外，泛型委托允许值类型实例在传给回调方法时不进行任何装箱。第 17 章“委托”会讲到，委托实际只是提供了 4 个方法的一个类定义。4 个方法包括一个构造器、一个 Invoke 方法，一个 BeginInvoke 方法和一个 EndInvoke 方法。如果定义的委托类型指定了类型参数，编译器会定义委托类的方法，用指定的类型参数替换方法的参数类型和返回值类型。

例如，假定像下面这样定义泛型委托：

```
public delegate TReturn CallMe<TReturn, TKey, TValue>(TKey key, TValue value);
```

编译器会将它转换成如下所示的类：

```
public sealed class CallMe<TReturn, TKey, TValue> : MulticastDelegate {
    public CallMe(Object object, IntPtr method);
```

```
        public virtual TReturn Invoke(TKey key, TValue value);
        public virtual IAsyncResult BeginInvoke(TKey key, TValue value,
            AsyncCallback callback, Object object);
        public virtual TReturn EndInvoke(IAsyncResult result);
    }
```

注意：建议尽量使用 FCL 预定义的泛型 Action 和 Func 委托。这些委托的详情将在 17.6 节“委托定义不要太多(泛型委托)”讲述。

12.5 委托和接口的逆变和协变泛型类型实参

委托的每个泛型类型参数都可以标记为协变量或逆变量[①]。利用这个功能，可将泛型委托类型的变量转换为相同的委托类型(但泛型参数类型不同)。泛型类型参数可以是以下任何一种形式。

- **不变量(invariant)** 意味着泛型类型参数不能更改。到目前为止，你在本章看到的全是不变量形式的泛型类型参数。
- **逆变量(contravariant)** 意味着泛型类型参数可以从一个类更改为它的某个派生类。在 C#中，是用 in 关键字标记逆变量形式的泛型类型参数。逆变量泛型类型参数只出现在输入位置，比如作为方法的参数。
- **协变量(covariant)** 意味着泛型类型参数可以从一个类更改为它的某个基类。C#是用 out 关键字标记协变量形式的泛型类型参数。协变量泛型类型参数只能出现在输出位置，比如作为方法的返回类型。

例如，假定存在以下委托类型定义(顺便说一下，它真的存在)：

```
public delegate TResult Func<in T, out TResult>(T arg);
```

其中，泛型类型参数 T 用 in 关键字标记，这使它成为逆变量；泛型类型参数 TResult 用 out 关键字标记，这使它成为协变量。

所以，如果像下面这样声明一个变量：

```
Func<Object, ArgumentException> fn1 = null;
```

就可以将它转型为另一个泛型类型参数不同的 Func 类型：

```
Func<String, Exception> fn2 = fn1;  // 不需要显式转型
Exception e = fn2("");
```

上述代码的意思是说：fn1 变量引用一个方法，获取一个 Object，返回一个 ArgumentException。而 fn2 变量引用另一个方法，获取一个 String，返回一个 Exception。

---

① 本书采用目前约定俗成的译法：covariant=协变量，contravariant=逆变量；covariance=协变性，contravariance=逆变性。另外，variance=可变性。至于两者更详细的区别，推荐阅读 Eric Lippert 的 Covariance and Contravariance in C#系列博客文章(https://tinyurl.com/5cx8nz33)。简而言之，协变性指定返回类型的兼容性，而逆变性指定参数的兼容性。——译注

由于可以将一个 String 传给期待 Object 的方法(因为 String 从 Object 派生)，而且由于可以获取返回 ArgumentException 的一个方法的结果，并将这个结果当成一个 Exception(因为 Exception 是 ArgumentException 的基类)，所以上述代码能正常编译，而且编译时能维持类型安全性。

注意：只有编译器能验证类型之间存在引用转换，这些可变性才有用。换言之，由于需要装箱，所以值类型不具有这种可变性。我个人认为，正是因为存在这个限制，所以这些可变性的用处不是特别大。例如，假定定义了以下方法：

```
void ProcessCollection(IEnumerable<Object> collection) { ... }
```

我不能在调用它时传递一个 List<DateTime>对象引用，因为 DateTime 值类型和 Object 之间不存在引用转换——虽然 DateTime 是从 Object 派生的。为了解决这个问题，可以像下面这样声明 ProcessCollection：

```
void ProcessCollection<T>(IEnumerable<T> collection) { ... }
```

另外，ProcessCollection(IEnumerable<Object> collection)最大的好处是 JIT 编译得到的代码只有一个版本。但如果使用 ProcessCollection<T> (IEnumerable<T> collection)，那么只有在 T 是引用类型的前提下，才可共享同一个版本的 JIT 编译代码。对于 T 是值类型的情况，每个值类型都有一份不同的 JIT 编译代码；不过，起码能在调用方法时传递一个值类型集合了。

另外，对于泛型类型参数，如果要将该类型的实参传给使用了 out 或 ref 关键字的方法，便不允许可变性。例如，以下代码会造成编译器报告错误消息：无效的可变性：类型参数"T"在"SomeDelegate<T>.Invoke(ref T)"中必须是不变量。现在的"T"是逆变量。[①]

```
delegate void SomeDelegate<in T>(ref T t);
```

使用要获取泛型参数和返回值的委托时，建议尽量为逆变性和协变性指定 in 和 out 关键字。这样做不仅没有不好的后果，还能使你的委托适用于更多的情形。

和委托相似，具有泛型类型参数的接口也可以将类型参数标记为逆变量和协变量。下面的示例接口有一个协变量泛型类型参数：

```
public interface IEnumerator<out T> : IEnumerator {
    Boolean MoveNext();
```

[①] Visual Studio 中文版实际显示的消息有点让人摸不着头脑：error CS1961：变型无效：类型参数“T”必须是在“Program.Node.SomeDelegate<T>.Invoke(ref T)”上有效的固定式。“T”为逆变。——译注

```
        T Current { get; }
    }
```

由于 T 是协变量，所以以下代码能顺利编译和运行：

```
// 这个方法接受任意引用类型的一个 IEnumerable 对象
Int32 Count(IEnumerable<Object> collection) { ... }

...
// 以下调用向 Count 传递一个 IEnumerable<String>对象
Int32 c = Count(new[] { "Grant" });
```

重要提示：开发人员有时会问："为什么必须显式用 in 或 out 标记泛型类型参数？"他们认为，编译器应该能检查委托或接口声明，并自动检测哪些泛型类型参数能够逆变和协变。虽然编译器确实能，但 C#团队认为必须由你订立协定(contract)，明确说明想允许什么。例如，假定编译器判断一个泛型类型参数是逆变量(用在输入位置)，但你将来向某个接口添加了成员，并将类型参数用在了输出位置。下次编译时，编译器就会认为该类型参数是不变量。但在引用了其他成员的所有地方，只要还以为"类型参数是逆变量"，那么就可能出错。

因此，编译器团队决定，在声明泛型类型参数时，必须由你显式使用 in 或 out 来标记可变性。以后使用这个类型参数时，假如用法与声明时指定的不符，编译器就会报错，提醒你违反了自己订立的协定。如果为泛型类型参数添加 in 或 out 来打破原来的协定，就必须修改使用旧协定的代码。

12.6 泛型方法

定义泛型类、结构或接口时，类型中定义的任何方法都可以引用类型指定的类型参数。类型参数可以作为方法参数的类型、方法返回值的类型或者方法内部定义的局部变量的类型使用。然而，CLR 还允许方法指定它自己的类型参数。这些类型参数也可以作为参数、返回值或局部变量的类型使用。

在下例中，类型定义了一个类型参数，方法也定义了自己的：

```
internal sealed class GenericType<T> {
   private T m_value;

   public GenericType(T value) { m_value = value; }

   public TOutput Converter<TOutput>() {
      TOutput result = (TOutput) Convert.ChangeType(m_value, typeof(TOutput));
      return result;    // 返回类型转换之后的结果
   }
}
```

在这个例子中，GenericType 类定义了类型参数(T)，Converter 方法也定义了自己的类型参数(TOutput)。这样的 GenericType 可以处理任意类型。Converter 方法能将 m_value 字段引用的对象转换成任意类型——具体取决于调用时传递的类型实参是什么。泛型方法的存在，为开发人员提供了极大的灵活性。

泛型方法的一个很好的例子是 Swap 方法：

```
private static void Swap<T>(ref T o1, ref T o2) {
   T temp = o1;
   o1 = o2;
   o2 = temp;
}
```

现在可以这样调用 Swap：

```
private static void CallingSwap() {
   Int32 n1 = 1, n2 = 2;
   Console.WriteLine("n1={0}, n2={1}", n1, n2);
   Swap<Int32>(ref n1, ref n2);
   Console.WriteLine("n1={0}, n2={1}", n1, n2);

   String s1 = "Aidan", s2 = "Grant";
   Console.WriteLine("s1={0}, s2={1}", s1, s2);
   Swap<String>(ref s1, ref s2);
   Console.WriteLine("s1={0}, s2={1}", s1, s2);
}
```

为获取 out 和 ref 参数的方法使用泛型类型很有意思，因为作为 out/ref 实参传递的变量必须具有与方法参数相同的类型，以防止损害类型安全性。涉及 out/ref 参数的这个问题已在 9.3 节“以传引用的方式向方法传递参数”讨论。事实上，Interlocked 类的 Exchange 和 CompareExchange 方法就是因为这个原因才提供泛型重载的[①]：

```
public static class Interlocked {
   public static T Exchange<T>(ref T location1, T value) where T: class;
   public static T CompareExchange<T>(
      ref T location1, T value, T comparand) where T: class;
}
```

泛型方法和类型推断

C#泛型语法因为涉及大量“<”和“>”符号，所以开发人员很容易被弄得晕头转向。为了改进代码的创建，增强可读性和可维护性，C#编译器支持在调用泛型方法时进行**类型推断**。这意味着编译器会在调用泛型方法时自动判断(或者说推断)要使用的类型。以下代码对类型推断进行了演示：

① where 子句将在本章稍后的 12.8 节“可验证性和约束”讨论。

```
private static void CallingSwapUsingInference() {
   Int32 n1 = 1, n2 = 2;
   Swap(ref n1, ref n2);    // 调用 Swap<Int32>

   String s1 = "Aidan";
   Object s2 = "Grant";
   Swap(ref s1, ref s2);    // 错误，不能推断类型
}
```

在上述代码中，对 Swap 的调用没有在一对“<”和“>”中指定类型实参。在第一个 Swap 调用中，C#编译器推断 n1 和 n2 都是 Int32，所以应该使用 Int32 类型实参来调用 Swap。

推断类型时，C#使用变量的数据类型，而不是变量引用的对象的实际类型。所以在第二个 Swap 调用中，C#发现 s1 是 String，而 s2 是 Object(即使它恰好引用了一个 String)。由于 s1 和 s2 是不同数据类型的变量，编译器拿不准要为 Swap 传递什么类型实参，所以会报告以下消息：error CS0411: 无法从用法中推断出方法"Program.Swap<T>(ref T, ref T)"的类型参数。请尝试显式指定类型参数。

类型可以定义多个方法，让其中一个方法接受具体数据类型，让另一个接受泛型类型参数，如下例所示：

```
private static void Display(String s) {
   Console.WriteLine(s);
}

private static void Display<T>(T o) {
   Display(o.ToString());       // 调用 Display(String)
}
```

下面展示了 Display 方法的一些调用方式：

```
Display("Jeff");                // 调用 Display(String)
Display(123);                   // 调用 Display<T>(T)
Display<String>("Aidan");       // 调用 Display<T>(T)
```

在第一个调用中，编译器可以调用接受一个 String 参数的 Display 方法，也可以调用泛型 Display 方法(将 T 替换成 String)。但是，C#编译器的策略是先考虑较明确的匹配，再考虑泛型匹配。所以，它会生成对非泛型 Display 方法的调用，也就是接收一个 String 参数的版本。对于第二个调用，编译器不能调用接收 String 参数的非泛型 Display 方法，所以必须调用泛型 Display 方法。顺便说一句，编译器优先选择较明确的匹配，开发人员应对此感到庆幸。假如编译器优先选择泛型方法，那么由于泛型 Display 方法会再次调用 Display(但传递由 ToString 返回的一个 String)，所以会造成无限递归。

对 Display 的第三个调用明确指定了泛型类型实参 String。这告诉编译器不要尝试推断类型实参。相反，应使用显式指定的类型实参。在本例中，编译器还假定我肯定是想调用泛型 Display 方法，所以会毫不犹豫地调用泛型 Display 方法。在内部，泛型 Display 方法会为传入的字符串调用 ToString 方法，然后将转换所得的字符串传给非泛型 Display 方

法。

12.7 泛型和其他成员

在 C#中，属性、索引器、事件、操作符方法、构造器和终结器本身不能有类型参数。但它们能在泛型类型中定义，而且这些成员中的代码能使用类型的类型参数。

C#之所以不允许这些成员指定自己的泛型类型参数，是因为微软 C#团队认为，开发人员很少需要将这些成员作为泛型使用。除此之外，为这些成员添加泛型支持的代价是相当高的，因为必须为语言设计足够的语法。例如，在代码中使用一个+操作符时，编译器可能要调用一个操作符重载方法。而在代码中，没有任何办法能伴随+操作符指定类型实参。

12.8 可验证性和约束

编译泛型代码时，C#编译器会进行分析，确保代码适用于当前已有或将来可能定义的任何类型。看看下面这个方法：

```
private static Boolean MethodTakingAnyType<T>(T o) {
   T temp = o;
   Console.WriteLine(o.ToString());
   Boolean b = temp.Equals(o);
   return b;
}
```

这个方法声明了 T 类型的临时变量(temp)。然后，方法执行一些变量赋值和方法调用。这个方法适用于任何类型。无论 T 是引用类型，是值类型或枚举类型，还是接口或委托类型，它都能工作。这个方法适用于当前存在的所有类型，也适用于将来可能定义的其他任何类型，因为所有类型都支持对 Object 类型的变量的赋值，也支持对 Object 类型定义的方法的调用(比如 ToString 和 Equals)。

再来看看下面这个方法：

```
private static T Min<T>(T o1, T o2) {
   if (o1.CompareTo(o2) < 0) return o1;
   return o2;
}
```

Min 方法试图使用 o1 变量来调用 CompareTo 方法。但是，许多类型都没有提供 CompareTo 方法，所以 C#编译器不能编译上述代码，它不能保证这个方法适用于所有类型。强行编译上述代码会报告消息：

```
error  CS1061:“T”未包含“CompareTo”的定义，并且找不到可接受第一个“T”类型参数的可访问扩展
方法“CompareTo”(是否缺少 using 指令或程序集引用?)。
```

所以从表面看，使用泛型似乎做不了太多事情。只能声明泛型类型的变量，执行变量赋值，再调用 Object 定义的方法，如此而已！显然，假如泛型只能这么用，可以说它几乎没有任何用。幸好，编译器和 CLR 支持称为**约束**的机制，可通过它使泛型变得真正有用！

约束的作用是限制能指定成泛型实参的类型数量。通过限制类型的数量，可以对那些类型执行更多操作。以下是新版本的 Min 方法，它指定了一个约束(加粗显示)：

```
public static T Min<T>(T o1, T o2) where T : IComparable<T> {
   if (o1.CompareTo(o2) < 0) return o1;
   return o2;
}
```

C# where 关键字告诉编译器，为 T 指定的任何类型都必须实现同类型(T)的泛型 IComparable 接口。有了这个约束，就可以在方法中调用 CompareTo，因为已知 IComparable<T>接口定义了 CompareTo。

现在，当代码引用泛型类型或方法时，编译器要负责保证类型实参符合指定的约束。例如，假如编译以下代码：

```
private static void CallMin() {
  Object o1 = "Jeff", o2 = "Richter";
  Object oMin = Min<Object>(o1, o2); // Error CS0311
}
```

编译器会报告以下消息：

```
error CS0311: 类型"object"不能用作泛型类型或方法"SomeType.Min<T>(T,T)"中的类型参数"T"。没有从"object"到"System.IComparable<object>"的隐式引用转换。
```

编译器报错是因为 `System.Object` 没有实现 `IComparable<Object>`接口。而且事实上，`System.Object` 没有实现任何接口。

对约束及其工作方式有了一个基本的认识后，让我们更深入地研究一下它。约束可应用于泛型类型的类型参数，也可应用于泛型方法的类型参数(如 `Min` 方法所示)。CLR 不允许基于类型参数名称或约束来进行重载；只能基于元数(类型参数个数)对类型或方法进行重载。下例对此进行了演示：

```
// 可以定义以下类型
internal sealed class AType {}
internal sealed class AType<T> {}
internal sealed class AType<T1, T2> {}

// 错误：与没有约束的 AType<T>冲突
internal sealed class AType<T> where T : IComparable<T> {}

// 错误：与 AType<T1, T2>冲突
internal sealed class AType<T3, T4> {}

internal sealed class AnotherType {
   // 可以定义以下方法，参数个数不同：
   private static void M() {}
   private static void M<T>() {}
   private static void M<T1, T2>() {}

   // 错误：与没有约束的 M<T>冲突
   private static void M<T>() where T : IComparable<T> {}

   // 错误：与 M<T1, T2>冲突
   private static void M<T3, T4>() {}
}
```

重写虚泛型方法时，重写的方法必须指定相同数量的类型参数，而且这些类型参数会继承在基类方法上指定的约束。事实上，根本不允许为重写方法的类型参数指定任何约束。但是，类型参数的名称是可以改变的。类似地，实现接口方法时，方法必须指定与接口方法等量的类型参数，这些类型参数将继承由接口方法指定的约束。下例使用虚方法演示了这

一规则：

```
internal class Base {
   public virtual void M<T1, T2>()
      where T1 : struct
      where T2 : class {
   }
}

internal sealed class Derived : Base {
   public override void M<T3, T4>()
      where T3 : EventArgs           // 错误
      where T4 : class               // 错误
      { }
}
```

试图编译上述代码，编译器会报告以下错误：

```
error CS0460: 重写和显式接口实现方法的约束是从基方法继承的，因此不能直接指定这些约束。
```

从 Derived 类的 M<T3, T4>方法中移除两个 where 子句，代码就能正常编译了。注意，类型参数的名称可以更改，比如将 T1 改成 T3，将 T2 改成 T4)；但约束不能更改(甚至不能指定)。

下面讨论编译器/CLR 允许向类型参数应用的各种约束。可用一个主要约束、一个次要约束以及/或者一个构造器约束来约束类型参数。接下来的三个小节将分别讨论这些约束。

12.8.1 主要约束

类型参数可以指定零个或者一个**主要约束**。主要约束可以是代表非密封类的一个引用类型。不能指定以下特殊引用类型：System.Object，System.Array，System.Delegate，System.MulticastDelegate，System.ValueType，System.Enum 或者 System.Void。

指定引用类型约束时，相当于向编译器承诺：一个指定的类型实参要么是与约束类型相同的类型，要么是从约束类型派生的类型。例如以下泛型类：

```
internal sealed class PrimaryConstraintOfStream<T> where T : Stream {
   public void M(T stream) {
      stream.Close();    //正确
  }
}
```

在这个类定义中，类型参数 T 设置了主要约束 Stream(在 System.IO 命名空间中定义)。这就告诉编译器，使用 PrimaryConstraintOfStream 的代码在指定类型实参时，必须指定 Stream 或者从 Stream 派生的类型(比如 FileStream)。如果类型参数没有指定主要约束，就默认为 System.Object。但是，如果在源代码中显式指定 System.Object，那么 C#会报错：error CS0702: 约束不能是特殊类"object"。

有两个特殊的主要约束：class 和 struct。其中，class 约束向编译器承诺类型实参是引用类型。任何类类型、接口类型、委托类型或者数组类型都满足这个约束。例如以下泛型类：

```
internal sealed class PrimaryConstraintOfClass<T> where T : class {
   public void M() {
      T temp = null;  // 允许，因为T肯定是引用类型
   }
}
```

在这个例子中，将 temp 设为 null 是合法的，因为 T 已知是引用类型，而所有引用类型的变量都能设为 null。不对 T 进行约束，上述代码就通不过编译，因为 T 可能是值类型，而值类型的变量不能设为 null。

struct 约束向编译器承诺类型实参是值类型。包括枚举在内的任何值类型都满足这个约束。但是，编译器和 CLR 将任何 System.Nullable<T>值类型视为特殊类型，不满足这个 struct 约束。原因是 Nullable<T>类型将它的类型参数约束为 struct，而 CLR 希望禁止像 Nullable<Nullable<T>>这样的递归类型。可空类型将在第 19 章“可空值类型”讨论。

以下示例类使用 struct 约束来约束它的类型参数：

```
internal sealed class PrimaryConstraintOfStruct<T> where T : struct {
   public static T Factory() {
      // 允许。因为所有值类型都隐式有一个公共无参构造器
      return new T();
   }
}
```

这个例子中的 new T()是合法的，因为 T 已知是值类型，而所有值类型都隐式地有一个公共无参构造器。如果 T 不约束，约束为引用类型，或者约束为 class，那么上述代码将无法通过编译，因为有的引用类型没有公共无参构造器。

12.8.2 次要约束

类型参数可以指定零个或者多个**次要约束**，次要约束代表接口类型。这种约束向编译器承诺类型实参实现了接口。由于能指定多个接口约束，所以类型实参必须实现了所有接口约束(以及主要约束，如果有的话)。第 13 章“接口”将详细讨论接口约束。

还有一种次要约束称为**类型参数约束**，有时也称为**裸类型约束**。这种约束用得比接口约束少得多。它允许一个泛型类型或方法规定：指定的类型实参要么就是约束的类型，要么是约束的类型的派生类。一个类型参数可以指定零个或者多个类型参数约束。下面这个泛型方法演示了如何使用类型参数约束：

```
private static List<TBase> ConvertIList<T, TBase>(IList<T> list)
   where T : TBase {
   List<TBase> baseList = new List<TBase>(list.Count);
   for (Int32 index = 0; index < list.Count; index++) {
```

```
            baseList.Add(list[index]);
        }
        return baseList;
    }
```

ConvertIList 方法指定了两个类型参数，其中 T 参数由 TBase 类型参数约束。意味着不管为 T 指定什么类型实参，都必须兼容于为 TBase 指定的类型实参。下面这个方法演示了对 ConvertIList 的合法调用和非法调用：

```
private static void CallingConvertIList() {
    // 构造并初始化一个 List<String>(它实现了 IList<String>)
    IList<String> ls = new List<String>();
    ls.Add("A String");

    // 1. 将 IList<String>转换成一个 IList<Object>
    IList<Object> lo = ConvertIList<String, Object>(ls);

    // 2. 将 IList<String>转换成一个 IList<IComparable>
    IList<IComparable> lc = ConvertIList<String, IComparable>(ls);

    // 3. 将 IList<String>转换成一个 IList<IComparable<String>>
    IList<IComparable<String>> lcs =
            ConvertIList<String, IComparable<String>>(ls);

    // 4. 将 IList<String>转换成一个 IList<String>
    IList<String> ls2 = ConvertIList<String, String>(ls);

    // 5. 将 IList<String>转换成一个 IList<Exception>
    IList<Exception> le = ConvertIList<String, Exception>(ls); // 错误
}
```

在对 ConvertIList 的第一个调用中，编译器检查 String 是否兼容于 Object。由于 String 从 Object 派生，所以第一个调用满足类型参数约束。在对 ConvertIList 的第二个调用中，编译器检查 String 是否兼容于 IComparable。由于 String 实现了 IComparable 接口，所以第二个调用满足类型参数约束。在对 ConvertIList 的第三个调用中，编译器检查 String 是否兼容于 IComparable<String>。由于 String 实现了 IComparable<String>接口，所以第三个调用满足类型参数约束。在对 ConvertIList 的第 4 个调用中，编译器知道 String 兼容于它自己。在对 ConvertIList 的第 5 个调用中，编译器检查 String 是否兼容于 Exception。由于 String 不兼容于 Exception，所以第 5 个调用不满足类型参数约束，编译器报告以下消息：

```
error CS0311: 类型"string"不能用作泛型类型或方法"Program.ConvertIList<T, TBase>(System.Collections.Generic.IList<T>)"中的类型参数"T"。没有从"string"到"System.Exception"的隐式引用转换。
```

12.8.3 构造器约束

类型参数可以指定零个或一个**构造器约束**，它向编译器承诺类型实参是实现了公共无参构

造器的非抽象类型。注意，如果同时使用构造器约束和 struct 约束，C#编译器会认为这是一个错误，因为这是多余的；所有值类型都隐式提供了公共无参构造器。以下示例类使用构造器约束来约束它的类型参数：

```
internal sealed class ConstructorConstraint<T> where T : new() {
   public static T Factory() {
      // 允许，因为所有值类型都隐式有一个公共无参构造器。
      // 而且，如果指定的是引用类型，约束也要求它提供公共无参构造器。
      return new T();
   }
}
```

这个例子中的 new T()是合法的，因为已知 T 是拥有公共无参构造器的类型。对所有值类型来说，这一点(拥有公共无参构造器)肯定成立。对于作为类型实参指定的任何引用类型，这一点也成立，因为构造器约束要求它必须成立。

开发人员有时想为类型参数指定一个构造器约束，并指定构造器要获取多个参数。目前，CLR(以及 C#编译器)只支持无参构造器。微软认为这已经能满足几乎所有情况，我对此也表示同意。

12.8.4 其他可验证性问题

本节剩下的部分将讨论另外几个特殊的代码构造。由于可验证性问题，这些代码构造在和泛型共同使用时，可能产生不可预期的行为。另外，还讨论了如何利用约束使代码重新变得可以验证。

1. 泛型类型变量的转型

除非转型为与约束兼容的类型，否则将泛型类型的变量转型为其他类型是非法的。

```
private static void CastingAGenericTypeVariable1<T>(T obj) {
   Int32 x = (Int32) obj ;    //错误
   String s = (String) obj;   //错误
}
```

上述两行会造成编译器报错，因为 T 可能是任意类型，无法保证成功转型。为了修改上述代码使其能通过编译，可以先转型为 Object。

```
private static void CastingAGenericTypeVariable2<T>(T obj) {
   Int32 x = (Int32) (Object) obj ;  //无错误
   String s = (String) (Object) obj; //无错误
}
```

虽然代码现在能编译，但 CLR 仍有可能在运行时抛出 InvalidCastException 异常。

转型为引用类型时还可以使用 C# as 操作符。下面对代码进行了修改，为 String 使用了 as 操作符 (Int32 是值类型不能用)：

```
private static void CastingAGenericTypeVariable3<T>(T obj) {
    String s = obj as String; // 不会报错
}
```

2. 将泛型类型变量设为默认值

除非将泛型类型约束成引用类型，否则将泛型类型变量设为 null 是非法的。

```
private static void SettingAGenericTypeVariableToNull<T>() {
   T temp = null;   // error CS0403 – 无法将 null 转换为类型参数"T",
                    // 因为它可能是不可以为 null 的值类型。请考虑改用 default(T)
}
```

由于未对 T 进行约束，所以它可能是值类型，而将值类型的变量设为 null 是不可能的。如果 T 被约束成引用类型，将 temp 设为 null 就是合法的，代码能顺利编译并运行。

微软 C#团队认为有必要允许开发人员将变量设为它的默认值，并专门为此提供了 default 关键字：

```
private static void SettingAGenericTypeVariableToDefaultValue<T>() {
   T temp = default(T); // 正确
}
```

上述代码中的 default 关键字告诉 C#编译器和 CLR 的 JIT 编译器，如果 T 是引用类型，就将 temp 设为 null；如果是值类型，就将 temp 的所有位设为 0。

3. 将泛型类型变量与 null 进行比较

无论泛型类型是否被约束，使用==或!=操作符将泛型类型变量与 null 进行比较都是合法

的：

```
private static void ComparingAGenericTypeVariableWithNull<T>(T obj) {
   if (obj == null) { /* 对于值类型，永远都不会执行 */ }
}
```

由于 T 未进行约束，所以可能是引用类型或值类型。如果 T 是值类型，那么 obj 永远都不会为 null。你或许以为 C#编译器会报错。但 C#编译器并不报错；相反，它能顺利地编译代码。调用这个方法时，如果为类型参数传递值类型，那么 JIT 编译器知道 if 语句永远都不会为 true，所以不会为 if 测试或者大括号内的代码生成本机代码。如果换用!=操作符，那么 JIT 编译器不会为 if 测试生成代码(因为测试结果肯定为 true)，但会为 if 大括号内的代码生成本机代码。

顺便说一句，如果 T 被约束成 struct，C#编译器会报错。值类型的变量不能与 null 进行比较，因为结果始终一样。

4. 两个泛型类型变量相互比较

如果泛型类型参数不能肯定是引用类型，那么对同一个泛型类型的两个变量进行比较是非法的：

```
private static void ComparingTwoGenericTypeVariables<T>(T o1, T o2) {
   if (o1 == o2) { } //错误
}
```

在这个例子中，T 未进行约束。虽然两个引用类型的变量相互比较是合法的，但两个值类型的变量相互比较是非法的，除非值类型重载了==操作符。如果 T 被约束成 class，上述代码能通过编译。如果变量引用同一个对象，==操作符会返回 true。注意，如果 T 被约束成引用类型，而且该引用类型重载了 operator==方法，那么编译器会在看到==操作符时生成对这个方法的调用。显然，所有些讨论也适合!=操作符。

写代码来比较基元值类型(Byte，Int32，Single，Decimal 等)时，C#编译器知道如何生成正确的代码。然而，对于非基元值类型，C#编译器不知道如何生成代码来进行比较。所以，如果 ComparingTwoGenericTypeVariables 方法的 T 被约束成 struct，编译器会报错。

不允许将类型参数约束成具体的值类型，因为值类型隐式密封，不可能存在从值类型派生的类型。如果允许将类型参数约束成具体的值类型，那么泛型方法会被约束为只支持该具体类型，这还不如不要泛型呢！

5. 泛型类型变量作为操作数使用

最后要注意，将操作符应用于泛型类型的操作数会出现大量问题。第 5 章讨论了 C#如何处理它的基元类型：Byte，Int16，Int32，Int64，Decimal 等。我特别指出 C#知道如何解释应用于基元类型的操作符(比如+，-，*和/)。但不能将这些操作符应用于泛型类型的变量。编译器在编译时确定不了类型，所以不能向泛型类型的变量应用任何操作符。所以，不可能写一个能处理任何数值数据类型的算法。下面是我想写的一个示例泛型方法：

```
private static T Sum<T>(T num) where T : struct {
   T sum = default(T) ;
   for (T n = default(T) ; n < num ; n++)
      sum += n;
   return sum;
}
```

可以看出，我千方百计想让这个方法通过编译。我将 T 约束成一个 struct，而且使用 default(T)将 sum 和 n 初始化为 0。但是，编译时还是得到了以下三个错误：

- **error CS0019**：运算符"<"无法应用于"T"和"T"类型的操作数
- **error CS0023**：运算符"++"无法应用于"T"类型的操作数
- **error CS0019**：运算符"+="无法应用于"T"和"T"类型的操作数

这是 CLR 的泛型支持体系的一个严重限制，许多开发人员(尤其是科学、金融和数学领域的开发人员)对这个限制感到很失望。许多人尝试用各种技术来避开这一限制，其中包括反射(参见第 23 章“程序集加载和反射”)、dynamic 基元类型(5.5 节)和操作符重载等。但是，所有这些技术都会严重损害性能或者影响代码的可读性。希望微软在 CLR 和编译器未来的版本中解决这个问题。

第 13 章 接口

本章内容：

- 类和接口继承
- 定义接口
- 继承接口
- 关于调用接口方法的更多探讨
- 隐式和显式接口方法实现(幕后发生的事情)
- 泛型接口
- 泛型和接口约束
- 实现多个具有相同方法名和签名的接口
- 用显式接口方法实现来增强编译时类型安全性
- 谨慎使用显式接口方法实现
- 设计：基类还是接口？

对于多继承(multiple inheritance)的概念，许多程序员并不陌生，它是指一个类从两个或多个基类派生的能力。例如，假定 `TransmitData` 类的作用是发送数据，`ReceiveData` 类的作用是接收数据。现在要创建 `SocketPort` 类，作用是发送和接收数据。在这种情况下，你会希望 `SocketPort` 从 `TransmitData` 和 `ReceiveData` 这两个类继承。

有的编程语言允许多继承，所以能从 `TransmitData` 和 `ReceiveData` 这两个基类派生出 `SocketPort`。但 CLR 不支持多继承(因此所有托管编程语言也支持不了)。CLR 只是通过接口提供了“缩水版”的多继承。本章将讨论如何定义和使用接口，还要提供一些指导性原则，以便你判断何时应该使用接口而不是基类。

13.1 类和接口继承

Microsoft .NET Framework 提供了 `System.Object` 类，它定义了 4 个公共实例方法：`ToString`，`Equals`，`GetHashCode` 和 `GetType`。该类是其他所有类的根或者说终极基类。换言之，所有类都继承了 `Object` 的 4 个实例方法。这还意味着只要代码能操作 `Object` 类的实例，就能操作任何类的实例。

由于微软的开发团队已实现了 `Object` 的方法，所以从 `Object` 派生的任何类实际都继承了以下内容。

- **方法签名**

  使代码认为自己是在操作 `Object` 类的实例，但实际操作的可能是其他类的实例。

- **方法实现**

  使开发人员定义 `Object` 的派生类时不必手动实现 `Object` 的方法。

在 CLR 中，任何类都肯定从一个(而且只能是一个)类派生，后者最终从 `Object` 派生。这个类称为基类。基类提供了一组方法签名和这些方法的实现。你定义的新类可在将来由其他开发人员用作基类——所有方法签名和方法实现都会由新的派生类继承。

CLR 还允许开发人员定义接口，它实际只是对一组方法签名进行了统一命名。这些方法不提供任何实现。类通过指定接口名称来继承接口，而且必须显式实现接口方法，否则 CLR 会认为此类型定义无效。当然，实现接口方法的过程可能比较繁琐，所以我才在前面说接口继承是实现多继承的一种“缩水版”机制。C#编译器和 CLR 允许一个类继承多个接口。当然，继承的所有接口方法都必须实现。

我们知道，类继承的一个重要特点是，凡是能使用基类型实例的地方，都能使用派生类型的实例。类似地，接口继承的一个重要特点是，凡是能使用具名接口类型的实例的地方，都能使用实现了接口的一个类型的实例。下面先看看如何定义接口。

13.2 定义接口

如前所述，接口对一组方法签名进行了统一命名。注意，接口还能定义事件、无参属性和有参属性(C#的索引器)。如前所述，所有这些东西本质上都是方法，它们只是语法上的简化。不过，接口不能定义任何构造器方法，也不能定义任何实例字段。

虽然 CLR 允许接口定义静态方法、静态字段、常量和静态构造器，但符合 CLS 标准的接口绝不允许，因为有的编程语言不能定义或访问它们。事实上，C#禁止接口定义任何一种这样的静态成员。

C#用 `interface` 关键字定义接口。要为接口指定名称和一组实例方法签名。下面是 FCL 中的几个接口的定义：

```
public interface IDisposable {
    void Dispose();
}

public interface IEnumerable {
    IEnumerator GetEnumerator();
```

```
}

public interface IEnumerable<T> : IEnumerable {
    new IEnumerator<T> GetEnumerator();
}

public interface ICollection<T> : IEnumerable<T>, IEnumerable {
    void     Add(T item);
    void     Clear();
    Boolean  Contains(T item);
    void     CopyTo(T[] array, Int32 arrayIndex);
    Boolean  Remove(T item);
    Int32    Count          { get; } // 只读属性
    Boolean  IsReadOnly     { get; } // 只读属性
}
```

在 CLR 看来，接口定义就是类型定义。也就是说，CLR 会为接口类型对象定义内部数据结构，同时可以通过反射机制来查询接口类型所提供的功能。和类型一样，接口可以在文件范围中定义，也可嵌套在另一个类型中。定义接口类型时，可以指定你希望的任何可见性/可访问性(public，protected，internal 等)。

根据约定，接口类型名称以大写字母 I 开头，目的是方便在源代码中辨认接口类型。CLR 支持泛型接口(前面几个例子已进行了演示)和接口中的泛型方法。本章稍后会讨论泛型接口提供的许多功能。另外，第 12 章“泛型”已全面讨论了泛型。

接口定义可以从另一个或多个接口 “继承”。但是，“继承”应打上引号，因为它并不是严格的继承。接口继承的工作方式并不完全和类继承一样。我个人倾向于将接口继承看成是将其他接口的协定(contract)包括到新接口中。例如，ICollection<T>接口定义就包含了 IEnumerable<T>和 IEnumerable 两个接口的协定。这有下面两层含义。

- 继承 ICollection<T>接口的任何类必须实现 ICollection<T>，IEnumerable<T>和 IEnumerable 这三个接口所定义的方法。
- 任何代码在引用对象时，如果期待该对象的类型实现了 ICollection<T>接口，可以认为该类型同时实现了 IEnumerable<T>和 IEnumerable 接口。

13.3 继承接口

本节介绍如何定义实现了接口的类型，然后介绍如何创建该类型的实例，并用这个对象调用接口的方法。C#将这个过程变得很简单，但幕后发生的事情还是有点复杂。本章稍后会详细解释。

下面是在 MSCorLib.dll 中定义的 System.IComparable<T>接口：

```
public interface IComparable<in T> {
```

```
    Int32 CompareTo(T other);
}
```

以下代码展示了如何定义实现了该接口的类型，同时还展示了对两个 Point 对象进行比较的代码：

```
using System;

// Point 从 System.Object 派生，并实现了 IComparable<T>
public sealed class Point : IComparable<Point> {
    private Int32 m_x, m_y;

    public Point(Int32 x, Int32 y) {
        m_x = x;
        m_y = y;
    }

    // 该方法为 Point 实现 IComparable<T>.CompareTo()
    public Int32 CompareTo(Point other) {
        return Math.Sign(Math.Sqrt(m_x * m_x + m_y * m_y)
            - Math.Sqrt(other.m_x * other.m_x + other.m_y * other.m_y));
    }

    public override String ToString() {
        return String.Format("({0}, {1})", m_x, m_y);
    }
}

public static class Program {
    public static void Main() {
        Point[] points = new Point[] {
            new Point(3, 3),
            new Point(1, 2),
        };

        // 下面调用由 Point 实现的 IComparable<T>的 CompareTo 方法
        if (points[0].CompareTo(points[1]) > 0) {
            Point tempPoint = points[0];
            points[0] = points[1];
            points[1] = tempPoint;
        }
        Console.WriteLine("Points from closest to (0, 0) to farthest:");
        foreach (Point p in points)
            Console.WriteLine(p);
    }
}
```

但凡有一个方法实现了某个接口方法签名，C#编译器都要求将它标记为 public。CLR 要求将接口方法标记为 virtual。如果你的代码不将方法显式标记为 virtual，那么编译器会将它们标记为 virtual 和 sealed；这会阻止派生类重写接口方法。但是，如果将方法显

式标记为 virtual，编译器就会将该方法标记为 virtual(并保持它的非密封状态)，使派生类能重写它。

派生类不能重写标记为 sealed 的接口方法。但是，派生类可以重新继承同一个接口，并为接口方法提供自己的实现。在对象上调用接口方法时，调用的是该方法在该对象的类型中的实现。下例对此进行了演示：

```
using System;

public static class Program {
    public static void Main() {
        /************************ 第一个例子************************/
        Base b = new Base();

        // 用 b 的类型来调用 Dispose，显示："Base's Dispose"
        b.Dispose();

        // 用 b 的对象的类型来调用 Dispose，显示："Base's Dispose"
         ((IDisposable)b).Dispose();


        /************************第二个例子************************/
        Derived d = new Derived();

        // 用 d 的类型来调用 Dispose，显示："Derived's Dispose"
        d.Dispose();

        // 用 d 的对象的类型来调用 Dispose，显示："Derived's Dispose"
         ((IDisposable)d).Dispose();


        /************************ 第三个例子 ************************/
        b = new Derived();

        // 用 b 的类型来调用 Dispose，显示："Base's Dispose"
        b.Dispose();

        // 用 b 的对象的类型来调用 Dispose，显示："Derived's Dispose"
         ((IDisposable)b).Dispose();
    }
}

// 这个类派生自 Object，它实现了 IDisposable
internal class Base : IDisposable {
    // 这个方法隐式密封，不能被重写
    public void Dispose() {
        Console.WriteLine("Base's Dispose");
    }
}
```

```
// 这个类派生自 Base，它重新实现了 IDisposable
internal class Derived : Base, IDisposable {
    // 这个方法不能重写 Base 的 Dispose，
    // 'new'表明该方法重新实现了 IDisposable 的 Dispose 方法
    new public void Dispose() {
        Console.WriteLine("Derived's Dispose");

        // 注意，下面这行代码展示了如何调用基类的实现(如果需要的话)
        // base.Dispose();
    }
}
```

13.4 关于调用接口方法的更多探讨

FCL 的 System.String 类型继承了 System.Object 的方法签名及其实现。此外，String 类型还实现了几个接口：IComparable，ICloneable，IConvertible，IEnumerable，IComparable<String>，IEnumerable<Char>和 IEquatable<String>。这意味着 String 类型不需要实现(或重写)其 Object 基类型提供的方法，但必须实现所有接口声明的方法。

CLR 允许定义接口类型的字段、参数或局部变量。使用接口类型的变量可以调用该接口定义的方法。此外，CLR 允许调用 Object 定义的方法，因为所有类都继承了 Object 的方法。以下代码对此进行了演示：

```
// s 变量引用一个 String 对象
String s = "Jeffrey";
// 可以使用 s 调用在 String，Object，IComparable，ICloneable，
// IConvertible，IEnumerable 中定义的任何方法

// cloneable 变量引用同一个 String 对象
ICloneable cloneable = s;
// 使用 cloneable 只能调用 ICloneable 接口声明的
// 任何方法(或 Object 定义的任何方法)

// comparable 变量引用同一个 String 对象
IComparable comparable = s;
// 使用 comparable 只能调用 IComparable 接口声明的
// 任何方法(或 Object 定义的任何方法)

// enumerable 变量引用同一个 String 对象
// 可在运行时将变量从一个接口转换成另一个，只要
// 对象的类型实现了这两个接口
IEnumerable enumerable = (IEnumerable) comparable;
// 使用 enumerable 只能调用 IEnumerable 接口声明的
// 任何方法(或 Object 定义的任何方法)
```

在这段代码中，所有变量都引用同一个“Jeffrey”String 对象。该对象在托管堆中；所以，使用其中任何变量时，调用的任何方法都会影响这个“Jeffrey”String 对象。不过，变量的类型规定了能对这个对象执行的操作。s 变量是 String 类型，所以可以用 s 调用 String

类型定义的任何成员(比如 Length 属性)。还可用变量 s 调用从 Object 继承的任何方法(比如 GetType)。

cloneable 变量是 ICloneable 接口类型。所以，使用 cloneable 变量可以调用该接口定义的 Clone 方法。此外，可以调用 Object 定义的任何方法(比如 GetType)，因为 CLR 知道所有类型都继承自 Object。不过，不能用 cloneable 变量调用 String 本身定义的公共方法，也不能调用由 String 实现的其他任何接口的方法。类似地，使用 comparable 变量可以调用 CompareTo 方法或 Object 定义的任何方法，但不能调用其他方法。

重要提示：和引用类型相似，值类型可以实现零个或多个接口。但是，值类型的实例在转换为接口类型时必须装箱。这是由于接口变量是引用，必须指向堆上的对象，使 CLR 能检查对象的类型对象指针，从而判断对象的确切类型。调用已装箱值类型的接口方法时，CLR 会跟随对象的类型对象指针找到类型对象的方法表，从而调用正确的方法。

13.5 隐式和显式接口方法实现(幕后发生的事情)

类型加载到 CLR 中时，会为该类型创建并初始化一个方法表(参见第 1 章“CLR 的执行模型”)。在这个方法表中，类型引入的每个新方法都有对应的记录项；另外，还为该类型继承的所有虚方法添加了记录项。继承的虚方法既有继承层次结构中的各个基类型定义的，也有接口类型定义的。所以，对于下面这个简单的类型定义：

```
internal sealed class SimpleType : IDisposable {
    public void Dispose() { Console.WriteLine("Dispose"); }
}
```

类型的方法表将包含以下方法的记录项。

- Object(隐式继承的基类)定义的所有虚实例方法。

- IDisposable(继承的接口)定义的所有接口方法。本例只有一个方法，即 Dispose，因为 IDisposable 接口只定义了这个方法。

- SimpleType 引入的新方法 Dispose。

为简化编程，C#编译器假定 SimpleType 引入的 Dispose 方法是对 IDisposable 的 Dispose 方法的实现。之所以这样假定，是由于 Dispose 方法的可访问性是 public，而接口方法的签名和新引入的方法完全一致。也就是说，两个方法具有相同的参数和返回类型。顺便说一句，如果新的 Dispose 方法被标记为 virtual，C#编译器仍然认为该方法匹配接口方法。

C#编译器将新方法和接口方法匹配起来之后，会生成元数据，指明 SimpleType 类型的方法表中的两个记录项应引用同一个实现。为了更清楚地理解这一点，下面的代码演示了如何调用类的公共 Dispose 方法以及如何调用 IDisposable 的 Dispose 方法在类中的实现：

```
public sealed class Program {
    public static void Main() {
        SimpleType st = new SimpleType();

        // 调用公共 Dispose 方法实现
        st.Dispose();

        // 调用 IDisposable 的 Dispose 方法的实现
        IDisposable d = st;
        d.Dispose();
    }
}
```

在第一个 Dispose 方法调用中，调用的是 SimpleType 定义的 Dispose 方法。然后定义 IDisposable 接口类型的变量 d，它引用 SimpleType 对象 st。调用 d.Dispose()时，调用的是 IDisposable 接口的 Dispose 方法。由于 C#要求公共 Dispose 方法同时是 IDisposable 的 Dispose 方法的实现，所以会执行相同的代码。在这个例子中，两个调用你看不出任何区别。输出结果如下所示：

```
Dispose
Dispose
```

现在重写 SimpleType，以便于看出区别：

```
internal sealed class SimpleType : IDisposable {
    public void Dispose() { Console.WriteLine("public Dispose"); }
    void IDisposable.Dispose() { Console.WriteLine("IDisposable Dispose"); }
}
```

在不改动前面的 Main 方法的前提下，重新编译并再次运行程序，输出结果如下所示：

```
public Dispose
IDisposable Dispose
```

在 C#中，将定义方法的那个接口的名称作为方法名前缀(例如 IDisposable.Dispose)，就会创建**显式接口方法实现**(Explicit Interface Method Implementation，EIMI①)。注意，C#中不允许在定义显式接口方法时指定可访问性(比如 public 或 private)。但是，编译器生成方法的元数据时，可访问性会自动设为 private，防止其他代码在使用类的实例时直接调用接口方法。只有通过接口类型的变量才能调用接口方法。

还要注意，EIMI 方法不能标记为 virtual，所以不能被重写。这是由于 EIMI 方法并非真的是类型的对象模型的一部分，它只是将接口(一组行为或方法)和类型连接起来，同时避免公开行为/方法。如果觉得这一点不好理解，那么你的感觉没有错！它就是不太好理解。本章稍后会介绍 EIMI 有用的一些场合。

13.6 泛型接口

C#和 CLR 所支持的泛型接口为开发人员提供了许多非常出色的功能。本节要讨论泛型接口提供的一些好处。

首先，泛型接口提供了出色的编译时类型安全性。有的接口(比如非泛型 IComparable 接口)定义的方法使用了 Object 参数或 Object 返回类型。在代码中调用这些接口方法时，可传递对任何类型的实例的引用。但这通常不是我们期望的。下面的代码对此进行了演示：

```
private void SomeMethod1() {
    Int32 x = 1, y = 2;
    IComparable c = x;

    // CompareTo 期待 Object，传递 y(一个 Int32)没有问题
    c.CompareTo(y); // y 在这里装箱

    // CompareTo 期待 Object，传递"2" (一个 String)虽然可以编译，
    // 但会在运行时抛出 ArgumentException 异常
    c.CompareTo("2");
}
```

接口方法理想情况下应该使用强类型。这正是 FCL 为什么包含泛型 IComparable<in T>接口的原因。下面修改代码来使用泛型接口：

```
private void SomeMethod2() {
    Int32 x = 1, y = 2;
    IComparable<Int32> c = x;

    // CompareTo 期待 Int32，传递 y(一个 Int32)没有问题
    c.CompareTo(y);   // y 在这里不装箱
```

---

① 请记住 EIMI 的意思，本书后面会大量使用这个缩写词。——译注

```
        // CompareTo 期待 Int32，传递"2"(一个 String)造成编译错误，
        // 指出 String 不能被转换为 Int32
        c.CompareTo("2");  // 错误
    }
```

泛型接口的第二个好处在于，处理值类型时装箱次数会少很多。在 SomeMethod1 中，非泛型 IComparable 接口的 CompareTo 方法期待获取一个 Object；传递 y(Int32 值类型)会造成 y 中的值装箱。但在 SomeMethod2 中，泛型 IComparable<in T>接口的 CompareTo 方法本来期待的就是 Int32；y 以传值的方式传递，无需装箱。

注意：

FCL 定义了 IComparable，ICollection，IList 和 IDictionary 等接口的泛型和非泛型版本。定义类型时如果要实现其中任何一个接口，那么一般应该实现泛型版本。FCL 保留非泛型版本是为了向后兼容，照顾在.NET Framework 支持泛型之前写的代码。非泛型版本还允许用户以较常规的、类型较不安全(more general, less type-safe)的方式处理数据。

有的泛型接口继承了非泛型版本，所以必须同时实现接口的泛型和非泛型版本。例如，泛型 IEnumerable<out T>接口继承了非泛型 IEnumerable 接口，所以实现 IEnumerable<out T>就必须实现 IEnumerable。

和其他代码集成时，有时必须实现非泛型接口，因为接口的泛型版本并不存在。这时，如果接口的任何方法获取或返回 Object，就会失去编译时的类型安全性，而且值类型将发生装箱。可利用本章 13.9 节“用显式接口方法实现来增强编译时类型安全性”介绍的技术来缓解该问题。

泛型接口的第三个好处在于，类可以实现同一个接口若干次，只要每次使用不同的类型参数。以下代码对此进行了演示：

```
using System;

// 该类实现泛型 IComparable<T>接口两次
public sealed class Number: IComparable<Int32>, IComparable<String> {
    private Int32 m_val = 5;

    // 该方法实现 IComparable<Int32>的 CompareTo 方法
    public Int32 CompareTo(Int32 n) {
        return m_val.CompareTo(n);
    }

    // 该方法实现 IComparable<String>的 CompareTo 方法
    public Int32 CompareTo(String s) {
```

```
            return m_val.CompareTo(Int32.Parse(s));
        }
    }

    public static class Program {
        public static void Main() {
            Number n = new Number();

            // 将 n 中的值和一个 Int32(5)比较
            IComparable<Int32> cInt32 = n;
            Int32 result = cInt32.CompareTo(5);

            //将 n 中的值和一个 String("5")比较
            IComparable<String> cString = n;
            result = cString.CompareTo("5");
        }
    }
```

接口的泛型类型参数可以标记为逆变和协变，为泛型接口的使用提供更大的灵活性。欲知协变和逆变的详情，请参见 12.5 节“委托和接口的逆变和协变泛型类型实参”。

13.7 泛型和接口约束

上一节讨论了泛型接口的好处。本节要讨论将泛型类型参数约束为接口的好处。

第一个好处在于，可以将泛型类型参数约束为多个接口。这样一来，传递的参数的类型必须实现全部接口约束。例如：

```
    public static class SomeType {
        private static void Test() {
            Int32 x = 5;
            Guid g = new Guid();

            // 对 M 的调用能通过编译，因为 Int32 实现了
            // IComparable 和 IConvertible
            M(x);

            // 这个 M 调用导致编译错误，因为 Guid 虽然
            // 实现了 IComparable，但没有实现 IConvertible
            M(g);
        }

        // M 的类型参数 T 被约束为只支持同时实现了
        // IComparable 和 IConvertible 接口的类型
        private static Int32 M<T>(T t) where T : IComparable, IConvertible {
            ...
        }
    }
```

这真的很“酷”！定义方法参数时，参数的类型规定了传递的实参必须是该类型或者它的派生类型。如果参数的类型是接口，那么实参可以是任意类类型，只要该类实现了接口。使用多个接口约束，实际是表示向方法传递的实参必须实现多个接口。

事实上，如果将 T 约束为一个类和两个接口，就表示传递的实参类型必须是指定的基类(或者它的派生类)，而且必须实现两个接口。这种灵活性使方法能细致地约束调用者能传递的内容。调用者不满足这些约束，就会产生编译错误。

接口约束的第二个好处是传递值类型的实例时减少装箱。上述代码向 M 方法传递了 x(值类型 Int32 的实例)。x 传给 M 方法时不会发生装箱。如果 M 方法内部的代码调用 t.CompareTo(…)，这个调用本身也不会引发装箱(但传给 CompareTo 的实参可能发生装箱)。

另一方面，如果 M 方法像下面这样声明：

```
private static Int32 M(IComparable t) {
    ...
}
```

那么 x 要传给 M 就必须装箱。

C#编译器为接口约束生成特殊 IL 指令，导致直接在值类型上调用接口方法而不装箱。不用接口约束便没有其他办法让 C#编译器生成这些 IL 指令；如此一来，在值类型上调用接口方法总是发生装箱。一个例外是如果值类型实现了一个接口方法，在值类型的实例上调用这个方法不会造成值类型的实例装箱。

13.8 实现多个具有相同方法名和签名的接口

定义实现多个接口的类型时，这些接口可能定义了具有相同名称和签名的方法。例如，假定有以下两个接口：

```
public interface IWindow {
    Object GetMenu();
}

public interface IRestaurant {
    Object GetMenu();
}
```

要定义实现这两个接口的类型，必须使用“显式接口方法实现”来实现这个类型的成员，如下所示：

```
// 这个类型派生自 System.Object，
// 并实现了 IWindow 和 IRestaurant 接口。
// (Mario Pizzeria 是一家比萨连锁餐厅)
public sealed class MarioPizzeria : IWindow, IRestaurant {
```

```
    // 这是IWindow的GetMenu方法的实现
    Object IWindow.GetMenu() { ... }

    // 这是IRestaurant的GetMenu方法的实现
    Object IRestaurant.GetMenu() { ... }

    // 这个GetMenu方法是可选的，与接口无关
    public Object GetMenu() { ... }
}
```

由于这个类型必须实现多个接口的GetMenu方法，所以要告诉C#编译器每个GetMenu方法对应的是哪个接口的实现。

代码在使用MarioPizzeria对象时必须将其转换为具体的接口才能调用所需的方法。例如：

```
MarioPizzeria mp = new MarioPizzeria();

// 以下代码调用MarioPizzeria的公共GetMenu方法
mp.GetMenu();

// 以下代码调用MarioPizzeria的IWindow.GetMenu方法
IWindow window = mp;
window.GetMenu();

// 以下代码调用MarioPizzeria的IRestaurant.GetMenu方法
IRestaurant restaurant = mp;
restaurant.GetMenu();
```

13.9 用显式接口方法实现来增强编译时类型安全性

接口很好用，它们定义了在类型之间进行沟通的标准方式。前面曾讨论了泛型接口，讨论了它们如何增强编译时的类型安全性和减少装箱操作。遗憾的是，有时由于不存在泛型版本，所以仍需实现非泛型接口。接口的任何方法接收 System.Object 类型的参数或返回 System.Object 类型的值，就会失去编译时的类型安全性，装箱也会发生。本节将介绍如何用“显式接口方法实现”(EIMI)在某种程度上改善这个局面。

下面是极其常用的 IComparable 接口：

```
public interface IComparable {
    Int32 CompareTo(Object other);
}
```

该接口定义了接收一个 System.Object 参数的方法。可以像下面这样定义实现了该接口的类型：

```
internal struct SomeValueType : IComparable {
    private Int32 m_x;
    public SomeValueType(Int32 x) { m_x = x; }
    public Int32 CompareTo(Object other) {
        return(m_x - ((SomeValueType) other).m_x);
    }
}
```

可用 SomeValueType 写下面这样的代码：

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    Object o = new Object();
    Int32 n = v.CompareTo(v);    // 不希望的装箱操作
    n = v.CompareTo(o);          // InvalidCastException 异常
}
```

上述代码存在两个问题。

- **不希望的装箱操作**

  v 作为实参传给 CompareTo 方法时必须装箱，因为 CompareTo 期待的是一个 Object。

- **缺乏类型安全性**

  代码能通过编译，但 CompareTo 方法内部试图将 o 转换为 SomeValueType 时抛出 InvalidCastException 异常。

这两个问题都可以用 EIMI 解决。下面是 SomeValueType 的修改版本，这次添加了一个 EIMI：

```
internal struct SomeValueType : IComparable {
    private Int32 m_x;
    public SomeValueType(Int32 x) { m_x = x; }

    public Int32 CompareTo(SomeValueType other) {
        return(m_x - other.m_x);
    }

    // 注意以下代码没有指定 public/private 可访问性
    Int32 IComparable.CompareTo(Object other) {
        return CompareTo((SomeValueType) other);
    }
}
```

注意新版本的几处改动。现在有两个 CompareTo 方法。第一个 CompareTo 方法不是获取一个 Object 作为参数，而是获取一个 SomeValueType。这样就不必将 other 的类型转换为 SomeValueType 了，所以用于强制类型转换的代码被去掉了。修改了第一个 CompareTo 方法使其变得类型安全之后，SomeValueType 还必须实现一个 CompareTo 方法来满足 IComparable 的协定。这正是第二个 IComparable.CompareTo 方法的作用，它是一个 EIMI。

经过这两处改动之后，就获得了编译时的类型安全性，而且不会发生装箱：

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    Object o = new Object();
    Int32 n = v.CompareTo(v);    // 不发生装箱
    n = v.CompareTo(o);          // 编译时错误
}
```

不过，定义接口类型的变量会再次失去编译时的类型安全性，而且会再次发生装箱：

```
public static void Main() {
    SomeValueType v = new SomeValueType(0);
    IComparable c = v;      // 装箱!

    Object o = new Object();
    Int32 n = c.CompareTo(v);    // 不希望的装箱操作
    n = c.CompareTo(o);          // InvalidCastException 异常
}
```

事实上，如本章前面所述，将值类型的实例转换为接口类型时，CLR 必须对值类型的实例进行装箱。因此，前面的 Main 方法中会发生两次装箱。

实现 IConvertible，ICollection，IList 和 IDictionary 等接口时 EIMI 很有用。可利用它为这些接口的方法创建类型安全的版本，并减少值类型的装箱。

13.10 谨慎使用显式接口方法实现

使用 EIMI 也可能造成一些严重后果，所以应该尽量避免使用 EIMI。幸好，泛型接口可以帮助我们在大多数时候避免使用 EIMI。但有时(比如实现具有相同名称和签名的两个接口方法时)仍然需要它们。EIMI 最主要的问题如下。

- 没有文档解释类型具体如何实现一个 EIMI 方法，也没有 Microsoft Visual Studio“智能感知”支持。
- 值类型的实例在转换成接口时装箱。
- EIMI 不能由派生类型调用。

下面详细讨论这些问题。

文档在列出一个类型的方法时，会列出显式接口方法实现(EIMI)，但没有提供类型特有的帮助，只有接口方法的常规性帮助。例如，Int32 类型的文档只是说它实现了 IConvertible 接口的所有方法。能做到这一步已经不错，它使开发人员知道存在这些方法。但也使开发人员感到困惑，因为不能直接在一个 Int32 上调用一个 IConvertible 方法。例如，下面的代码无法编译：

```
public static void Main() {
    Int32 x = 5;
    Single s = x.ToSingle(null);   // 试图调用一个 IConvertible 方法
}
```

编译这个方法时，C#编译器会报告以下消息：error CS0117: "int"未包含"ToSingle"的定义。这个错误信息使开发人员感到困惑，因为它明显是说 Int32 类型没有定义 ToSingle 方法，但实际上定义了。

要在一个 Int32 上调用 ToSingle，首先必须将其转换为 IConvertible，如下所示：

```
public static void Main() {
    Int32 x = 5;
    Single s = ((IConvertible) x).ToSingle(null);
}
```

对类型转换的要求不明确，而许多开发人员自己看不出来问题出在哪里。还有一个更让人烦恼的问题：Int32 值类型转换为 IConvertible 会发生装箱，既浪费内存，又损害性能。这是本节开头提到的 EIMI 存在的第二个问题。

EIMI 的第三个也可能是最大的问题是，它们不能被派生类调用。下面是一个例子：

```
internal class Base : IComparable {

    // 显式接口方法实现
    Int32 IComparable.CompareTo(Object o) {
```

```
            Console.WriteLine("Base's CompareTo");
            return 0;
        }
    }

    internal sealed class Derived : Base, IComparable {

        // 一个公共方法，也是接口的实现
        public Int32 CompareTo(Object o) {
            Console.WriteLine("Derived's CompareTo");

            // 试图调用基类的 EIMI 导致编译错误：
            // error CS0117: “Base” 不包含“CompareTo”的定义
            base.CompareTo(o);
            return 0;
        }
    }
```

在Derived的CompareTo方法中调用base.CompareTo导致C#编译器报错。现在的问题是，Base 类没有提供一个可供调用的公共或受保护 CompareTo 方法，它提供的是一个只能用 IComparable 类型的变量来调用的 CompareTo 方法。可将 Derived 的 CompareTo 方法修改成下面这样：

```
    // 一个公共方法，也是接口的实现
    public Int32 CompareTo(Object o) {
        Console.WriteLine("Derived's CompareTo");

        // 试图调用基类的 EIMI 导致无穷递归
        IComparable c = this;
        c.CompareTo(o);

        return 0;
    }
```

这个版本将 this 转换成 IComparable 变量 c，然后用 c 调用 CompareTo。但 Derived 的公共 CompareTo 方法充当了 Derived 的 IComparable.CompareTo 方法的实现，所以造成了无穷递归。这可以通过声明没有 IComparable 接口的 Derived 类来解决：

```
    internal sealed class Derived : Base /*, IComparable */ { ... }
```

现在，前面的CompareTo方法将调用Base中的CompareTo方法。但有时不能因为想在派生类中实现接口方法就将接口从类型中删除。解决这个问题的最佳方法是在基类中除了提供一个被选为显式实现的接口方法，还要提供一个虚方法。然后，Derived 类可以重写虚方法。下面展示了如何正确定义 Base 类和 Derived 类：

```
    internal class Base : IComparable {

        //显式接口方法实现(EIMI)
        Int32 IComparable.CompareTo(Object o) {
```

```
            Console.WriteLine("Base's IComparable.CompareTo");
            return CompareTo(o); // 调用虚方法
        }

        // 用于派生类的虚方法(该方法可为任意名称)
        public virtual Int32 CompareTo(Object o) {
            Console.WriteLine("Base's virtual CompareTo");
            return 0;
        }
    }

    internal sealed class Derived : Base, IComparable {

        // 一个公共方法，也是接口的实现
        public override Int32 CompareTo(Object o) {
            Console.WriteLine("Derived's CompareTo");

            // 现在可以调用 Base 的虚方法
            return base.CompareTo(o);
        }
    }
```

注意，这里是将虚方法定义成公共方法，但有时可能需要定义成受保护方法。把方法定义为受保护(而不是公共)是可以的，但必须进行另一些小的改动。我们的讨论清楚证明了务必谨慎使用 EIMI。许多开发人员在最初接触 EIMI 时，认为 EIMI 非常“酷”，于是开始肆无忌惮地使用。千万不要这样做！EIMI 在某些情况下确实有用，但应该尽量避免使用，因为它们导致类型变得很不好用。

13.11 设计：基类还是接口

经常有人问："应该设计基类还是接口？"这个问题不能一概而论，以下设计规范或许能帮你理清思路。

- **IS-A 对比 CAN-DO 关系**[①]

  类型只能继承一个实现。如果派生类型和基类型建立不起 IS-A 关系，就不用基类而用接口。接口意味着 CAN-DO 关系。如果多种对象类型都"能"做某事，就为它们创建接口。例如，一个类型能将自己的实例转换为另一个类型(`IConvertible`)，一个类型能序列化自己的实例(`ISerializable`)。注意，值类型必须从 `System.ValueType` 派生，所以不能从一个任意的基类派生。这时必须使用 CAN-DO 关系并定义接口。

- **易用性**

  对于开发人员，定义从基类派生的新类型通常比实现接口的所有方法容易得多。基类型可提供大量功能，所以派生类型可能只需稍做改动。而提供接口的话，新类型必须实现所有成员。

- **一致性实现**

  无论接口协定(contract)订立得有多好，都无法保证所有人百分之百正确实现它。事实上，COM 就颇受该问题之累，导致有的 COM 对象只能正常用于 Microsoft Office Word 或 Microsoft Internet Explorer。而如果为基类型提供良好的默认实现，那么一开始得到的就是能正常工作并经过良好测试的类型。以后根据需要修改就可以了。

- **版本控制**

  向基类型添加一个方法，派生类型将继承新方法。所以，如果一开始使用的就是一个能正常工作的类型，那么用户的源代码甚至不需要重新编译。相反，向接口添加新成员，会强迫接口的继承者更改其源代码并重新编译。

在 FCL 中，涉及数据流处理(streaming data)的类采用的是"实现继承"方案[②]。例如，`System.IO.Stream` 是抽象基类，提供了包括 `Read` 和 `Write` 在内的一组方法。其他类(`System.IO.FileStream`，`System.IO.MemoryStream` 和 `System.Net.Sockets.NetworkStream`)都从 `Stream` 派生。在这三个类中，每一个和 `Stream` 类的关系都是 IS-A 关系，这使具体类[③]的实现变得更容易。例如，派生类只需实现同步 I/O

---

① IS-A 是指"属于"，例如，汽车属于交通工具；CAN-DO 是指"能做某事"，例如，一个类型能将自己的实例转换为另一个类型。——译注

② 即继承基类的实现。——译注

③ 对应于"抽象类"。——译注

操作，异步 I/O 操作则已经从 Stream 基类继承了。

必须承认，为流类(*XXX*Stream)选择继承的理由不是特别充分，因为 Stream 基类实际只提供了很少的实现。那么就以 Microsoft Windows 窗体控件类为例好了。Button，CheckBox，ListBox 和其他所有窗体控件都从 System.Windows.Forms.Control 派生。Control 实现了大量代码，各种控件类简单继承一下即可正常工作。这时选择继承应该没有疑问了吧？

相反，微软采用基于接口的方式来设计 FCL 中的集合。System.Collections.Generic 命名空间定义了几个与集合有关的接口：IEnumerable<out T>，ICollection<T>，IList<T>和 IDictionary<TKey，TValue>。然后，微软提供了大量类来实现这些接口组合，包括 List<T>，Dictionary<TKey, TValue>，Queue<T>和 Stack<T>等等。设计者之所以在类和接口之间选择 CAN-DO 关系，是因为不同集合类的实现迥然有异。换句话说，List<T>，Dictionary<TKey，TValue>和 Queue<T>之间没有多少能共享的代码。

不过，这些集合类提供的操作相当一致。例如，都维护了一组可枚举的元素，而且都允许添加和删除元素。假定有一个对象引用，对象的类型实现了 IList<T>接口，就可在不需要知道集合准确类型的前提下插入、删除和搜索元素。这个机制太强大了！

最后要说的是，两件事情实际能同时做：定义接口，*同时*提供实现该接口的基类。例如，FCL 定义了 IComparer<in T>接口，任何类型都可选择实现该接口。此外，FCL 提供了抽象基类 Comparer<T>，它实现了该接口，同时为非泛型 IComparer 的 Compare 方法提供了默认实现。接口定义和基类同时存在带来了很大的灵活性，开发人员可根据需要选择其中一个。

第Ⅲ部分 基本类型

- 第 14 章 字符、字符串和文本处理
- 第 15 章 枚举类型和位标志
- 第 16 章 数组
- 第 17 章 委托
- 第 18 章 定制特性
- 第 19 章 可空值类型

译者后记

从事软件开发的人，都是耐得住寂寞的人。Jeffrey 不仅耐得住寂寞，还在自己的专业领域取得了很高的造诣。取得了很高的造诣不说，他还愿意将自己的心得与大家分享。愿意和大家分享不说，他还非常实诚，真心想把自己的全部知识都清楚地交待给读者。字里行间，全是殷殷叮嘱。无浮夸之文字，倾心血而写就，近十年之所悟，尽呈现于本书。

读完这本书，你的心灵会受到极大的震撼。原因很简单，以前许多似懂非懂的概念，现在变得清晰明了；以前自以为是的做法，现在得到彻底纠正；以前艰苦摸索的编程技巧，现在如同 1+1 一样简单。

Jeffrey 最擅长的就是把最基本的东西讲清楚。你以前或许知道 1+1 等于 2，但他会把 1+1 为什么等于 2 讲得明明白白。最终你会有一种顿悟的感觉，然后自动地知道 1+2 等于几，2+2 等于几。不需要翻阅其他书籍来查询结果。

没有后期维护的书不算是好书。即使是本书英文原版，也维护了一份很长的勘误表，我本人也为其贡献良多。本书中文版将延续我一直以来坚持的风格，建立专门的页面进行维护，提供资源下载和勘误等服务。请大家继续前往我的博客(*https://bookzhou.com*)，发表关于本书的意见和建议。

本书于 2023 年底进行了全面修订：

- 全面提升了可读性：a)使用了更清晰的代码字体。b)进行了大量细节修订，以前一些不是特别清楚或者容易产生歧义的地方，都进行了澄清。c)新增多处“译注”。
- 新的中文版反映了到目前为止所有读者（包括英文版读者）反馈的勘误。
- 标注了一些不适合.NET Core 的内容（数量极少）。
- 更新了书中引用的 URL，有的进行了缩短(短网址)
- 修订了正文中对部分章节标题的引用
- 增补了术语表
- 现在，绝大多数屏幕截图基于 Windows 10/11 和 Visual Studio 2022

注意，原书虽然基于 Visual Studio 2012/2013，.NET Framework 4.5.x 和 C# 5.0，但在目前最新的 Visual Studio 2022(.NET Framework 4.8.x 和 C# 10)上，除了关于 AppDomain 的部分内容，本书绝大多数内容都适合当前的情况。毕竟，整本书讲的都是基础。所有“语法糖”在 ILDasm.exe 的面前，都是一样的。

关于阅读方式和顺序，请参考知乎文章：*https://www.zhihu.com/question/27283360*。感谢赵劼。

最后，如同往常一样，我要说所有的功劳都要归于作者，所有的错误都要归于译者。欢迎大家批评指正。下表列出了本书中文版使用的一些术语，有异于 MSDN 文档(已于 2019 年迁移至 Microsoft Docs，后者现已更名为 Microsoft Learn)的会专门指出。

术语	说明
(S)Byte	等同于“SByte 和 Byte”，这是作者喜欢的说法。类似的还有(U)Int16，(U)Int32，(U)IntPtr 等
AppDomain	(保留原文)
Compute-Bound 和 I/O-Bound	计算限制和 I/O 限制(一个操作如果因为处理器和 I/O 的限制而不得不等待，就称为计算限制或 I/O 限制的操作)
JIT(just-in-time)	文档中喜欢把它翻译为“实时”，例如“实时编译”。但不要忽略这里“just”的含义。它的意思是“刚好”在需要的时候我才做这个事情。不需要我就不做。所以，我们把它翻译为“即时”，以示跟“real-time”的区别。事实上，程序员之间的交流都是用 JIT“某某”。
Windows Store app	Windows Store 应用(文档中翻译成“Windows 应用商店应用”，显得过于冗长)
action method	操作方法
antecedent task 和 continuation task	前置任务和延续任务
arity	元数。在计算机编程中，一个函数或运算(操作)的元数是指函数获取的实参或操作数的个数。它源于像 unary(arity=1)、binary(arity=2)、ternary(arity=3)这样的单词
asynchronously synchronization	异步地同步(同步对资源的访问，但以异步方式进行，即不阻塞线程)
atomic	原子性(读或写都是一次完成，别的线程看不到中间状态，就说这种读写是原子性的)
attribute	特性(以文档为准)
awaiter	等待者(调用 GetAwaiter 所返回的对象)
bit flag	位标志
block	阻塞(停下来等着)
callback	回调(回调方法简称为“回调”)
calling thread	调用线程(发出调用的线程，也称主调线程)
capture	捕捉(文档中主要用“捕捉”，偶尔用“捕获”)
cast	转型(不用文档的“强制类型转换”是因为太冗长)
Compact	压缩(但此压缩非彼压缩，这里只是按照约定俗成的方式将 compact 翻译成“压缩”。不要以为“压缩”后内存会增多。相反，这里的“压缩”更接近于“碎片整理”。事实上，compact 正确的意思是“变得更紧凑”。但事实上，从上个世纪 80 年开

	始，人们就把它看成是 compress 的近义词而翻译成“压缩”，以讹传讹至今)
compute-bound 和 I/O-bound 的操作	本书分别翻译为“计算限制”和“I/O 限制”的操作。一个操作如果因为处理器或 I/O 的限制而不得不等待，就称为计算限制或 I/O 限制的操作。为什么不是“计算密集型”和“I/O 密集型”？虽然后者更常用，八、九十年代的教科书均采用这种说法，但它们并不准确。这些操作之所以最好以异步的方式来做，不是因为它们很“密集”，而是因为设备(CPU、存储设备等)本身能力有限，不能很快地完成一个操作。例如，某个操作需要长时间“霸占”CPU 来完成冗长的计算任务。这个时候能说它是一个“计算密集型”操作吗？
contract	文档的翻译非常混乱，包括协定、协议、合约、约定和契约等；本书使用“协定”
covariance 和 contravariance	协变和逆变(协变是指在要求使用一个类型的地方，能改为使用它的基类；逆变则是指在要求使用一个类型的地方，能改为使用它的派生类。C#用关键字 `in` 表示逆变量，用在输入位置；用 `out` 表示协变量，用在输出位置。详情参见 12.5 节)
culture	语言文化(而不是文档中的“区域性”)
cyclical reference	循环引用(我引用你，你引用我)
declarative	声明性(文档如此，个人更喜欢“宣告式”)
dispose	文档翻译成“释放”。但“dispose 一个对象”真正的意思是“清理或处置对象中包装的资源(比如它的字段引用的对象)，然后等着在一次垃圾回收之后回收该对象占用的托管堆内存(此时才释放)。”为避免误解，本书将 dispose 翻译成“清理”，偶尔也会保留原文。
entry	记录项(而不是“条目”、“入口”)
flush	文档翻译成“刷新”，本书保留原文。其实 flush 在技术文档中的意思和日常生活中一样，即“冲洗(到别处)”。例如，我们会说“冲水”，不会说“刷新水”
formatter	格式化器(文档是“格式化程序”)
get accessor method	get 访问器方法(取值函数或 getter)
guideline	设计规范(和文档一致，但“指导原则”更佳)
handler	处理程序。文档如此，个人不喜欢“程序”二字。
heap 和 stack	heap 是“堆”，stack 是“栈”，两者有严格的区分。但是，因为长期的“误译”，包括许多教科书和微软自己的文档在内，都将 stack 翻译为“堆栈”，例如“调用堆栈”(图22-2)和“堆栈跟踪”(*https://tinyurl.com/bdzepr9m*)。本书为了保持和文档的一致性，有时不得不将 stack 也说成“堆栈”。但看到“堆栈”，请自动脑补为 stack。

helper method	辅助方法
host	寄宿(动词)或宿主(名词)
invoke 和 call	都翻译成“调用”，但两者是有区别的。执行一个所有信息都已知的方法时，用 call 比较恰当。但在需要先“唤出”(invoke)某个东西来帮你调用一个信息不明的方法时，用 invoke 就比较恰当。阅读关于委托和反射的章节时，可以更好地体会两者的区别
literal	直接在代码中书写的值就是 literal 值，比如字符串值和数值("Hello"和 123)。翻译成什么的都有，包括直接量、字面值、文字常量、常值(台译)等。但实际最容易理解的还是英文原文。本书采用“字面值”
marshal	封送
metadata	元数据
mutex	互斥体
native method	本机方法(其实就是非托管方法)
native	本机(文档如此，个人更喜欢“原生”，比如原生类库、原生 C/C++代码、原生堆。一切非托管的，都是 native 的)
operand	操作数(要操作/运算的目标)
operator	操作符(而不是文档中的“运算符”)
overload 和 override	重载和重写，后者经常说成“覆写”或“覆盖”
preempt	抢占
primitive types	基元类型(文档如此，不是“基本类型”。可以在代码中使用的最简单的构造就称为“基元”，其他构造都是它们复合而成的)
provider	提供程序(文档如此，个人不喜欢“程序”二字)
raise an event	引发事件
recursion count 和 recursive lock	递归计数和递归锁(可重入的锁就是递归锁，重入的次数就是递归计数)
scalability	伸缩性(在少量时间里做更多工作的能力，就是所谓的“伸缩性”。作为一个伸缩性好的服务器，理论上应该 CPU 越多，一个耗时操作所需的时间就越短。通俗地说，在多个 CPU 之间并行执行，执行时间将根据 CPU 的数量成比例地缩短)
self-hosted	自寄宿(应用程序的进程自己容纳 CLR，就是所谓的自寄宿)
semaphore	信号量
set accessor method	set 访问器方法(赋值函数或 setter)
side effect	副作用。在计算机编程中，如果一个函数/方法或表达式除了生成一个值，还会造成状态的改变，就说它会造成副作用；或者说会执行一个副作用
singleton	单实例(例如，如果某类型在每个 AppDomain 中只能有一个实

	例，它就是单实例类型)
spinning	自旋(线程不是阻塞，而是原地“打转”，浪费 CPU 时间。但在用于保护执行得非常快的代码区域时性能比较好)
string interning	字符串留用(而不是文档中的“字符串拘留”)
synchronous 和 asynchronous	同步和异步。同步意味着一个操作开始后必须等待它完成；异步则意味着不用等它完成，可立即返回做其他事情。不要将“同步”理解成“同时”。同步意味着不能同时访问一个资源，只有在你用完了之后，我才能接着用。在多线程编程中，“同步”的定义是：当两个或更多的线程需要存取共同的资源时，必须确定在同一时间点只有一个线程能存取该资源，而实现这个目标的过程就称为“同步”。切记不可将同步理解成能够“同时访问一个资源”。
tap(点击)，press and hold(长按)，slide(滑动)，swipe(轻扫)，turn(转动)，pinch(收缩)和 stretch(拉伸)	Windows 8 开始引入的各种触摸“手势”
throw an exception	抛出异常(而不是文档中的“引发异常”)
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。记住，这个过程是“一圈一圈”地进行的。
volatile	易变(文档将 volatile 翻译为“可变”。其实它是“短暂存在”、“易变”的意思，因为可能有多个线程都对这种字段进行修改，本书采用“易变”)
work item 和 worker thread	工作项和工作者线程(线程池术语。工作项是指要由一个线程池线程调用的方法，代表线程实际要做的工作；处理工作项的线程称为工作者线程。工作项被放到一个队列中，工作者线程将工作项从队列中取出并处理)