微软.NET程序员系列

Wintellect

Raise Your Windows IQ

- 汇集.NET平台思想之大成
- 雄踞亚马逊销售榜首14个月
- .NET核心开发组成员鼎力推荐
- .NET领域当之无愧的圣经教本

Microsoft®

.NET框架程序设计(修订版)

Applied Microsoft .NET Framework Programming

(美) Jeffrey Richter 著

李建忠 译

Microsoft .net

清华大学出版社

Microsoft

.NET 框架程序设计（修订版）

Applied Microsoft .NET Framework Programming

文稿编辑：杨志娟
封面设计：陈刘源

"Jeffrey运用他多年的Windows编程经验和深刻的见解对.NET框架的运作原理做了鞭辟入里的分析。从这本书中，你可以抓住.NET框架的真正精髓。"

——**Brad Abrams**，微软公司，.NET框架组，程序经理主管

"Jeffrey Richter将他一贯擅长用平实的文笔来描述复杂技术的特点精确地应用在了C#语言、.NET框架，以及通用语言运行时上。对于任何一名想要理解.NET技术的运作原理和执行方式的程序员来说，这本书不可或缺。"

——**Jim Miller**，微软公司，通用语言运行时内核组，程序经理主管

"正如Programming Applications for Microsoft Windows已经成为Win32程序员人手一册的宝典一样，Applied Microsoft .NET Framework Programming也必定成为.NET 框架程序员的案头必备。这本书是帮助读者从底层理解.NET框架编程的独一无二的教本。对于程序员来讲，要想轻松快捷地编写出稳定、安全、高效的托管应用程序，非此书莫属。"

——**Steven Pratschner**，微软公司，通用语言运行时组，程序经理

本书内容

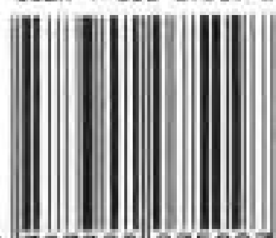
- .NET框架体系结构
- 生成、打包、部署及管理应用程序与类型
- 共享程序集
- 类型基本原理
- 基元类型、引用类型与值类型
- 通用对象操作
- 类型成员及其访问限定
- 常数、字段、方法、属性和事件
- 枚举类型与位标记
- 数组类型
- 接口
- 定制特性
- 委托
- 异常处理
- 自动内存管理
- 应用程序域与反射

作者介绍

Jeffrey Richter 是一位在全球享有盛誉的技术作家，尤其在Windows和.NET领域有着杰出的贡献。他的很多Windows著作畅销全球，其中《Windows高级编程指南》和《Windows核心编程》早已成为Windows程序设计教程的经典之作。

Jeffrey 是Wintellect公司的创始人之一，也是MSDN杂志.NET专栏的特邀编辑。现在他正领导开发Wintellect公司的.NET程序设计课程，向大众推广.NET技术。因为他自1999年开始就参与了微软.NET框架开发组的咨询工作，与这些一线人员一起经历了.NET的孕育与诞生，所以对.NET思想的领悟、对.NET的细节熟稔，是其他的作者难以企及的。

ISBN 7-302-07509-3



新书查询及技术支持：http://www.epress.cn
读者服务邮箱：service@wenyuan.com.cn

定价：68.00元

微软.NET 程序员系列

Microsoft .NET 框架程序设计(修订版)

(美) Jeffrey Richter 著
李建忠 译

清华大学出版社
北 京

内 容 简 介

本书是《微软.NET程序员系列》丛书之一，主要介绍如何开发面向Microsoft .NET框架的各种应用程序。Microsoft .NET框架是微软公司推出的新平台，包含通用语言运行时(CLR)和.NET框架类库(FCL)。本书将深入解释CLR的工作机制及其提供的各种构造，同时还将讨论FCL中一些重要的类型。全书共分为五个部分，包括：.NET框架基本原理、类型和通用语言运行时、类型设计、基本类型，以及类型管理。

本书适用于要了解、掌握.NET平台的读者，尤其适合广大编程爱好者、软件工程师、系统架构师阅读。

Microsoft .NET 框架程序设计(修订版)
Applied Microsoft .NET Framework Programming
Jeffrey Richter

Copyright © 2002 by Microsoft Press.

Original English Language Edition Copyright © 2002 by Microsoft Press.
Published by arrangement with the original publisher, Microsoft Press,
a division of Microsoft Corporation, Redmond, Washington, U.S.A.

本书中文版由 Microsoft Press 授权清华大学出版社出版。

北京市版权局著作权合同登记号 图字 01-2001-2105

版权所有，翻印必究。

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

图书在版编目(CIP)数据

Microsoft .NET框架程序设计(修订版)/(美)里克特(Richter,J.)著；李建忠译.—北京：清华大学出版社，2003

(微软.NET程序员系列)

书名原文：Applied Microsoft .NET Framework Programming

ISBN 7-302-07509-3

Ⅰ.M… Ⅱ.①里…②李… Ⅲ.计算机网络—程序设计 Ⅳ.TP393

中国版本图书馆CIP数据核字(2003)第097582号

出 版 者：清华大学出版社　　地　址：北京清华大学学研大厦
http://www.tup.com.cn　　邮　编：100084
社 总 机：010-62770175　　客户服务：010-62776969

译　者：李建忠
文稿编辑：杨志娟
封面设计：陈刘源
印 刷 者：北京市世界知识印刷厂
装 订 者：三河市李旗庄少明装订厂
发 行 者：新华书店总店北京发行所
开　本：185×230　印张：38.75　插页：5　字数：933千字
版　次：2003年11月第1版　2004年2月第2次印刷
书　号：ISBN 7-302-07509-3/TP·5532
印　数：4001～6000
定　价：68.00元

探微知著

——译序

.NET 平台推出至今已经整整3年了，3年来.NET 从最初的市场噱头走进了绝大多数开发人员的视野，并成为很大一部分开发人员真实生活的一部分。我相信读者在阅读这篇译序时已经不再有".NET 技术有什么优势"或者"我是否需要学习.NET 技术"之类的疑问了。然而.NET 技术浩如烟海，从底层的托管运行时，到五花八门的高级编程语言，从巨量的类库 API，到丰富多彩的应用程序模型，各路技术尽数囊括，多种思想和盘托出。从哪里入手学，怎样学？

是的，Jeffrey Richter 先生在本书中给出了这一问题的答案，这就是.NET 底层框架技术。

我一直认为从微观入手、从底层入手是掌握软件技术的不二法门，我发现我的这种心得和 Jeffrey 先生在本书中的技术思路不谋而合。这也是我在翻译这部名著的高强度劳动中得以保持快乐的一个重要原因。在.NET 平台中，.NET 框架占据着核心的位置，它是整个.NET 平台的关键支撑，是为众多高级语言(如 C#、Managed C++、Visual Basic .NET 等)和应用程序模型(如 Windows 窗体、ASP.NET Web 窗体、XML Web 服务等)提供各种服务的重要基石。脱离.NET 框架来谈.NET 平台，难免不陷入空中楼阁的尴尬境地。实际上，在.NET 平台中，任何一门编程语言提供的功能都只是.NET 框架下一个子集的映射，具体的语言已经退居为一个语法表达的层次了。所以虽然本书的描述语言为 C#，但是任何.NET 语言下的开发人员都可以通过阅读本书来获得教益。除了编程语言外，各种类型的.NET 应用程序在设计、开发、测试、部署、运行等诸多环节也和.NET 框架密切相关。虽然本书并不涉及这些具体.NET 应用程序模型的细节，但是如果没有对.NET 框架的深刻把握，学习再多的.NET 应用程序模型"开发技巧"都将只是徒劳——皮之不存，毛将焉附？因此，不管是学习 Windows 窗体、还是 ASP.NET Web 窗体、抑或是 XML Web 服务，我建议大家首先从.NET 框架开始迈出坚实的一步——探微而知著，这也是 Jeffrey 先生写作本书的初衷。

本书融合了 Jeffrey Richter 先生十几年的开发、顾问经验，对整个.NET 框架技术进行了一次全面的检阅和酣畅淋漓的剖析。全书采用 Jeffrey 先生惯用的"于细微处入手，在平实中参透"的技术文笔，将.NET 框架中的各个技术要点一一呈现给读者。尤其是书中对程序集、元数据、值类型/引用类型(装箱/拆箱)、异常处理、垃圾收集等这些.NET 核心技术的讲解，运思精深，鞭辟入里。对这些主题的掌握是理解整个.NET 框架的关键，也是构建各种高效应用程序的必经路径。在对这些大部头的技术主题进行深入剖析的同时，书中还对读者在.NET 应用程序开发实践中遇到的各种问题给予了很

多有益的忠告和指导。此外，书中也不乏大量经典的可重用代码范例，如 Dispose 资源管理模式、事件设计模式、Equals 的重写实现、对象池的设计等，这些代码像珍珠一样遍布在本书的各个角落。实际上，笔者在日常的开发实践中就重用了其中许多代码。虽然本书是对.NET 框架技术的一次全面的剖析，但它并不是对.NET 框架技术事实的简单堆砌，而是带领读者去探索、去领悟、去构筑一个关于.NET 平台核心技术的思想体系。书中每一章内容都值得读者反复阅读、细细品味，相信每一次重新阅读都会带给读者对.NET 平台更为深刻的理解。

虽然本书涵盖了大量艰深的技术，但是阅读起来却并不困难，这中间除了 Jeffrey 先生优美的技术文笔外，大量短小精悍的示例代码也功不可没。除非读者是久经沙场的软件开发老手，阅读本书只是出于消遣或者对 Jeffrey 先生的一份尊敬，否则笔者强烈建议大家在阅读的同时多通过动手演练这些代码来领悟这本名著的精髓。在阅读本书和进行代码演练时，建议大家到微软 MSDN 网站上下载一份.NET 框架 SDK，同时准备一个源代码编辑器(这方面有 UltraEdit、SharpDevelop、Visual Studio .NET，以及 C# Builder 等)。在.NET 框架 SDK 中，除了最常用的工具 C#编译器(CSC.exe)外，建议读者把 IL 反汇编工具(ildasm.exe)也放在手边——实际上 Jeffrey 先生在本书中就有一个论断“如果你对 IL 反汇编工具显示的内容了解得越多，那么你对.NET 框架的理解也会越深”。另外，读者也要多参阅.NET 框架 SDK 中附带的类库文档。在源代码编辑器方面，如果读者使用的是比较高级的 IDE(比如 Visual Studio .NET)，建议读者刚开始学习的时候避免使用其中的向导、自动代码生成等功能，而应该只把它们作为源代码编辑器来使用。因为不只我一个人发现刚开始学习程序设计就使用大型豪华 IDE 的高级功能往往会把学习者引入一个“神秘的、混沌的、并且不知所措的”状态中，除了打击读者的信心和积极性外，对学习掌握程序设计没有任何帮助。实际上从最简单的源代码编辑开始的学习者到后来往往最容易驾驭这些高级 IDE。工具的学习也有一个探微知著的过程。

再来简单介绍一下本书的作者 Jeffrey Richter 先生。我相信阅读这篇译序的很多读者都比我更了解 Jeffrey 先生，以及他在.NET 技术上所做的令人尊敬的贡献。Jeffrey Richter 先生早在 1999 年.NET 平台的整个开发工作正在进行时便受邀和微软一线的研发人员在一起讨论各种.NET 技术问题，开展.NET 技术的咨询工作。本书就诞生于这个过程。因此如果读者在本书中看到 Jeffrey 先生对.NET 框架中大量技术要点来龙去脉的精辟讲解，各个优缺点的犀利分析，以及各种开发实践的忠告指导时，请不要感到任何的惊奇和突然——Jeffrey 先生的确具有.NET 技术的半官方背景。

我相信对于.NET 领域应该很快会有这样的说法——.NET 程序员将会因为此书而分为两类，一类是读过《Applied Microsoft .NET Framework Programming》的，一类是没有读过《Applied Microsoft .NET Framework Programming》的。

下面简单交代一下译本的几个问题。大家知道.NET 中引入了许多新的术语，很多术语至今在开发界还不能做到耳熟能详，甚至时有混乱。因此，除了译本中附上的术语表外，书中正文在刚开始遇到这些术语时，一般都采用了中英并陈的方式，对于有些读者可能感到比较生疏的术语，还在多个地

方进行了中英并陈，目的无非只有一个——方便读者阅读和理解。再一个问题是勘误。Jeffrey 先生为本书的英文原版在微软出版社的网站上维护有一个勘误表，本书定稿时的最新勘误日期为 2003 年 5 月 30 日，这些勘误直接在译本中进行了修改。所以如果读者进行中英文对照阅读，会发现个别地方讲的不一致，这时读者首先应该去查阅英文版的勘误表。对于那些笔者发现的、但未在原书勘误中列出的错误(其中绝大多数都和 Jeffrey 先生进行了讨论)，笔者采用添加译注的方式进行了说明。译本在出版之后也会长期维护一个勘误表。最后，本书采用了页页对译，书后的索引完全来自本书英文原版。但是由于有些地方添加了 Jeffrey 先生的勘误和笔者的译注，这使得页页对译很难在每一页中都得到严格的保证。因此读者偶尔会发现索引位置不匹配的问题，这时读者应该在索引页码的前后页查找。希望译本的这些做法能够提升读者的阅读体验。

最后，我要感谢所有对本书的翻译给予过帮助和支持的朋友。我首先要感谢本书的原作者 Jeffrey Richter 先生，除了为本书带来大量的精彩华章外，Jeffrey 先生对这本中译本也贡献颇多。在本书的翻译过程中，Jeffrey 先生对我提出的大量不管是琐碎的、还是艰深的问题都给予了耐心的解答，这是这本书的翻译能够成功完成的重要保证。书中很多重要的译注都是我和 Jeffrey 先生交流切磋的结果，如果它们能够给读者的阅读和对.NET 框架的理解带来帮助的话，请把这些功劳记在 Jeffrey 先生的名下。我还要特别感谢清华大学出版社的章忆文女士，是她促成了这次合作，读者才有机会看到这个译本。还有清华大学出版社的杨志娟女士等各位编辑，她们在这部名著的翻译、校对、排版、制图等各个环节上做了大量的工作，读者阅读本书时得到的各种美好体验与享受一定有她们的很多功劳。还有在本书翻译过程中给过我帮助、鼓励与支持的很多朋友和网友，请原谅我不能在这篇短序中一一列出他们的名字，我相信一个好的译本是对大家最好的答谢。

我真诚地希望为这本书付出辛勤劳动的各方没有辜负大家的期望，希望各位读者朋友在美好的阅读享受之中能对.NET 框架技术有一个彻底的、通透的理解。如果你阅读完本书后告诉我你对整个.NET 框架技术有了一种豁然开朗的感觉，并大大提高了你的.NET 应用程序设计和开发能力，甚至因此改变了你的程序生涯——这一点儿都不奇怪，这也是 Jeffrey 先生所有著作的特点——我将会感到十分的荣幸和愉快！如果你发现译本中有任何问题或不满意的地方，请不要吝啬你的 CPU 和 Memory 给我发信，我会非常感谢读者任何的批评与反馈，并及时做出更正与改进。谢谢！

李建忠 2003/07 于上海浦东

jzli@china.com(个人电子邮箱，欢迎读者的批评等阅读反馈)

www.lijianzhong.com(个人网站，提供本书勘误、代码、FAQ 等支持)

前　言

随着时间的推移，我们以计算机为主导的生活方式不断发生着变化。如今，每个人都注意到了互联网的价值，并开始越来越依赖基于 Web 的服务。就个人而言，我喜欢通过互联网来购物、买票、比较产品、获取交通状况、阅读产品评价等。

然而，我发现仍然有许多事情在目前用互联网是无法完成的。比如，我想在我的周围找一家有着特色风味的餐馆。而且，我还希望能够查询一下这个餐馆在晚上七点钟是否有座位。或者，假如我经营着一家商店，我可能想知道哪家厂商库存中有某种商品的现货。如果有多家厂商可以供货，那我希望能找出价格最低，或者是发货最快的那一家。

诸如此类的服务在今天还不存在有两个主要的原因。首先，目前还没有一个标准来集成这些信息。各个厂商都有自己描述产品的方式。旨在描述各类信息的可扩展标记语言(XML)刚刚成为一门新兴标准。其次，开发集成这些服务的复杂性也是一个巨大的挑战。

微软认为销售服务是未来的发展方向。换句话说，就是企业提供服务，而感兴趣的用户消费这些服务。在这之中，许多服务将会是免费的，有一些是可以通过订购计划获得的，还将有一些是按使用情况来付费的。我们可以将这些服务视作某种商业逻辑的执行。下面是一些服务的例子：

- 验证信用卡交易
- 获取从甲地到乙地的方位
- 查看餐馆的菜单
- 预定航班、客房或者一辆出租车
- 更新在线相册里的照片
- 合并家长和孩子的日程表来安排一次家庭度假
- 从支票账户中支付账单
- 跟踪发送给我们的包裹

还可以列出很多类似的服务，任何一家企业都可以实现这些服务。毫无疑问，微软在不远的将来会创建并提供其中一些服务。其他一些公司(例如我们自己的)也将会提供这样的服务，其中一些还可能在自由市场中与微软发生竞争。

那么，怎样才能从我们今天所处的世界中便捷地获得所有这些服务呢？我们又怎样利用并组合这些服务来为用户提供丰富的特色应用(基于 HTML 或者其他技术)呢？举个例子，如果餐馆提供了他

们的菜单访问服务，我们就应该能够写出一个应用程序，它可以查询每个餐馆的菜单，搜索某一特定的口味或菜肴，然后向用户推荐那些离他们最近的餐馆来。

> **注意**　为了创建这样一些丰富的应用程序，企业必须提供针对他们的商业逻辑服务的编程接口，这样的编程接口必须可以通过远程网络(比如互联网)进行调用。这就是整个 Microsoft .NET 平台创新的主旨。简单地讲，.NET 平台创新就是关于人、信息和设备之间的互联。

让我用下面的方式对此做一个解释：我们知道，计算机都有外设(例如鼠标、显示器、键盘、数字相机以及扫描仪等)和它们相连。而操作系统，例如微软的 Windows，则将应用程序对这些外设的访问抽象化后，为我们提供了一个开发平台。我们完全可以将这些外设看作.NET 平台中的服务。

在.NET 平台所描述的这个崭新的世界里，服务(或者外设)将和互联网络相连接。开发人员需要一种便捷的方式来访问这些服务，而 Microsoft .NET 创新的一部分便是提供一个这样的开发平台。图 0.1 展示了它们之间的一个类比。在左边，从开发人员来看，Windows 是一个抽象了硬件外设的开发平台。在右边，从开发人员来看，Microsoft .NET 框架则是一个抽象了 XML Web 服务的开发平台。

图 0.1

虽然是一个在相关标准的开发和定义方面的领导者——正是这些标准使得这样的新世界成为可能，但微软并不拥有其中任何一个标准。客户机通过创建特殊格式的 XML 来描述一个服务器请求，然后通过企业内部网或者互联网来发送它(典型地利用 HTTP 协议)。服务器知道怎样分析这些 XML 数据，处理客户的请求，然后再以 XML 作为响应传回客户机。其中 SOAP(简单对象访问协议)在这里用来描述通过 HTTP 协议发送的特殊格式的 XML。

图 0.2 描述了一组 XML Web 服务相互之间通过带 XML 负载的 SOAP 进行通信的情形。另外，图中还向我们描述了客户端应用程序可能和 Web 服务，甚至其他客户端(通过 SOAP 或者 XML)进行通信的情形。图中向我们展示的是客户通过 HTML 从一个 Web 服务器获得请求结果的情形。这时，用户可能会首先在一个 Web 窗体上填写自己的请求，然后将这个 Web 窗体发送回 Web 服务器。接着

Web 服务器处理用户的请求(包括和一些 Web 服务进行通信)，最后将结果以 HTML 页面的形式返回给终端用户。

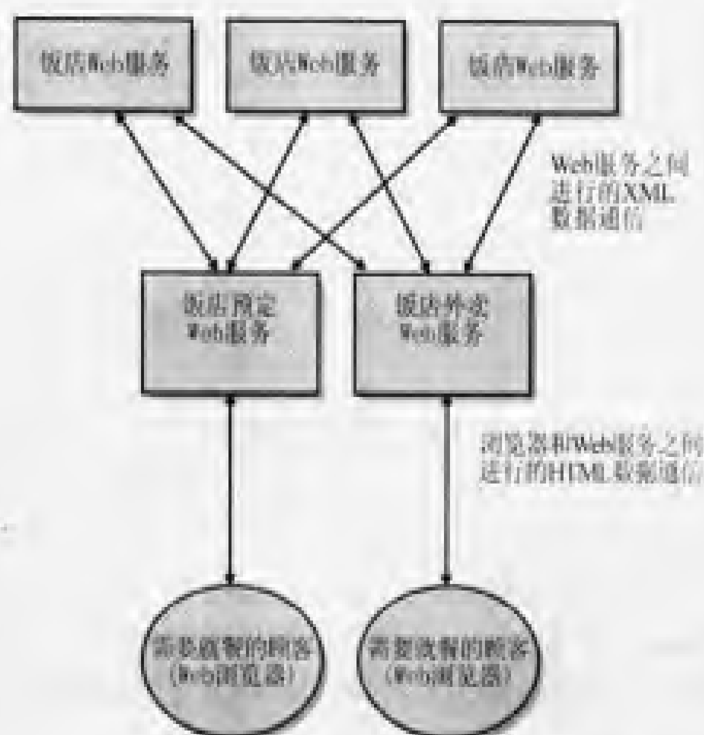


图 0.2

提供 Web 服务的计算机必须运行在一个能够侦听 SOAP 请求的操作系统上。微软希望这样的操作系统是 Windows，但这也并非必须。任何能够侦听 TCP/IP 套接字(socket)端口、并能对该端口读写字节的操作系统都可以胜任这样的工作。在不远的将来，移动电话、寻呼机、汽车、微波炉、电冰箱、手表、音响设备、游戏控制台，以及各种其他的设备都将能够参与到 Web 服务的新世界中。

在客户端，操作系统必须能够在套接字端口读取字节以发送服务请求。客户端的计算机当然要能满足用户应用程序的需求。如果用户想创建一个窗口、或者一个菜单，那么客户端的操作系统必须能够提供这样的功能，否则应用程序开发人员必须自己手动来实现。当然，微软希望人们在构建 Web 服务客户端应用程序时也能够使用 Windows。但同样，Windows 只是一个推荐，而非必须。

不管微软是否参与其中，充满 Web 服务的全新世界都终将到来。微软整个.NET 平台创新的目的就是帮助开发人员来创建和访问这些服务。

当然，如果愿意，我们也可以编写自己的操作系统，创建自己的 Web 服务器来侦听和处理 SOAP 请求。但是这通常很困难，并且要花费很长的时间。微软已经替我们完成了这些复杂的工作。我们可以直接把我们的注意力放在真正的商业逻辑和服务上，而把底层的通信协议和基础构造交给微软来做——微软有众多热衷于此的开发人员。

Microsoft .NET 平台构成

我和微软以及他们的各种技术打交道已经有很多年了。多年来，我有幸目睹了微软公司的各种技术创新：MS-DOS、Windows、Windows CE、OLE、COM、ActiveX、COM+、Windows DNA 等。当我第一次听到 Microsoft .NET 平台时，我就知道它将续写微软不败的神话。它使我由衷地感到微软的确是一个非常有远见卓识的公司。而且，看得出来，整个微软军团在实现他们这一远景规划方面也是个个雄心壮志。

我们不妨来将 Microsoft .NET 平台和下面几项微软的技术做一对比。首先是 ActiveX，这只不过是 COM 的另一个好听的名字罢了。ActiveX 本身所涵盖的东西有限，而且和这个名字扯在一起的 ActiveX 控件从来就没有获得真正意义上的成功。另一个是 Windows DNA(Distributed InterNet Architecture)，这也是一个漂亮的市场标签而已，其实质内容仍是一大堆现存的技术。和前面两项技术相比，我显然对 Microsoft .NET 最有信心。写作本书也是对我这种信心的一种佐证。那么，到底整个 Microsoft .NET 创新包含有哪些内容呢？在下面的各个小节中我将向大家一一道来。

底层操作系统：Windows

由于 Web 服务和使用 Web 服务的应用程序仍然运行在计算机上，而且既然是计算机都要有外设，所以我们仍然需要一个操作系统。当然，微软建议人们选用 Windows。实际上，微软为整个 Windows 产品线都添加了 XML Web 服务支持。在它们中间，Windows XP 和 Windows .NET 服务器家族产品将会为这种服务驱动(service-driven)的世界提供最好的支持。

特别地，Windows XP 和 Windows .NET 服务器家族产品已经集成了 Microsoft .NET Passport XML Web 服务支持。Passport 是一种用户认证服务。为了保证信息访问的安全，许多 Web 服务都需要用户认证。当用户登录到一台运行有 Windows XP 或者 Windows .NET 服务器家族中的产品时，用户在登录使用 Passport 认证的 Web 站点和 Web 服务时的效率将会大大提升。换句话说，用户在访问不同的互联网站点时将不再需要每次都输入用户名和密码。可以想见，Passport 会为我们带来巨大的便利。因为我们只需保留一个身份标识和一个密码，而且只需输入一次！

另外，Windows XP 和 Windows .NET 服务器家族产品对使用.NET 框架技术实现的应用程序的加载和执行还提供了内置的支持。最后，Windows XP 和 Windows .NET 服务器家族产品还有一个新型的、可扩展的即时消息通知应用程序。该应用程序允许第三方厂商(如 Expedia、United States Postal Service 等)和它们的用户进行无缝的通信。例如，当用户的航班(Expedia)延迟，或者邮包(U.S. Postal Service)送达时，他们可以收到自动通知。

不知道大家对这些怎么看，就个人而言，我对它们的期望已经有很多年了——确切地说，是迫不及待！

辅助产品：.NET 企业服务器

作为.NET 平台创新的一部分，微软提供了一些有价值的服务器产品供各种公司选择。下面是其中的一些企业服务器产品：

- Microsoft Application Center 2000
- Microsoft BizTalk Server 2000
- Microsoft Commerce Server 2000
- Microsoft Exchange 2000
- Microsoft Host Integration Server 2000
- Microsoft Internet Security and Acceleration (ISA) Server 2000
- Microsoft Mobile Information Server 2002
- Microsoft SQL Server 2000

这些产品刚开始很有可能仅仅出于市场目的而被贴上.NET 标签。但是随着时间的推移和.NET 战略的推进，我相信微软会将很多.NET 特性集成到这些产品中。

Microsoft XML Web 服务：.NET My Services

微软当然不会满足于仅仅作为一个 Web 服务的底层技术提供商，他们也希望能在 Web 服务领域内大玩一把。自然，他们也会创建自己的 XML Web 服务：其中有些可能是免费的，但有些可能就要收费。微软的初始计划提供以下一些.NET My Services：

- .NET Alerts
- .NET ApplicationSettings
- .NET Calendar
- .NET Categories
- .NET Contacts
- .NET Devices
- .NET Documents
- .NET FavoriteWebSites
- .NET Inbox
- .NET Lists
- .NET Locations
- .NET Presence

- .NET Profile
- .NET Services
- .NET Wallet

这些面向消费者的 XML Web 服务被微软称作“.NET My Services”。大家可以到 *http://www.Microsoft.com/MyServices/* 站点上获取有关该服务的更多信息。随着时间的推移，微软不仅会创建更多面向消费者的 Web 服务，还会创建一些面向商业用户的 Web 服务。除了这些向外公开的 Web 服务外，微软也会创建一些供内部销售、付账等使用的 Web 服务。当然只有微软的员工才可以访问这些内部服务。我预料 Web 服务首先会在许多公司内部的网络中流行起来。而面向整个互联网公众的 Web 服务和使用这些 Web 服务的应用程序则可能进展要稍微慢一些。

开发平台：.NET 框架

大家目前可能已经看到.NET My Services 中的一些服务了，比如 Passport。这些服务目前都运行在 Windows 上，实现技术也还是如 C/C++、ATL、Win32、COM 等一些传统技术。随着时间的推移，这些服务以及许多新的服务最终必将会采用一些更新的技术来构建，比如 C#(音作“C sharp”)、.NET 框架。

> **重要**　虽然这里介绍的都是创建基于互联网的应用程序和 Web 服务，但是.NET 框架的能力远不至此。实际上，.NET 框架开发平台允许我们创建各种各样的应用程序：XML Web 服务、Web 窗体、Win32 GUI 应用程序、Win32 CUI(控制台 UI)应用程序，Windows 服务(由服务控制管理器控制)、实用程序，以及独立的组件模块。而本书所述的内容适用于上述任何类型的应用程序。

.NET 框架包含两个部分：通用语言运行时(CLR)和.NET 框架类库(FCL)。.NET 框架本身又是.NET 平台创新中一个关键的组成部分。实际上，它也是本书将要谈论的一切：开发面向.NET 框架的应用程序和 XML Web 服务。

刚开始的时候，CLR 和 FCL 还只能运行于各种版本的 Windows 平台上，包括 Windows 98、Window 98 第 2 版、Windows Me、Windows NT 4、Windows 2000，以及 32 位和 64 位的 Windows XP 和 Windows .NET 服务器家族产品。另外还有用于 PDA(如 Windows CE 和 Palm)和家电产品(小型设备)的.NET 微缩框架(.NET Compact Framework)。但是，在 2001 年 12 月 13 日，欧洲计算机制造商协会(European Computer Manufacturers Association，简称 ECMA)已经接受了 C#编程语言、部分的 CLR 以及部分的 FCL 作为标准。可以预见，在不远的将来，与 ECMA 标准兼容的这些技术将出现在各种操作系统和 CPU 上。

注意 .NET框架并没有和Windows XP(包括家庭版和专业版)打包在一起。然而，Windows .NET服务器家族(包括Windows .NET Web服务器、Windows .NET标准服务器、Windows .NET企业服务器以及Windows .NET数据中心服务器)将会包括.NET框架。实际上，这也是Windows .NET服务器家族的名称来由。下一个版本的Windows(代码名为“长角牛”，即Longhorn)将会在所有版本中包括.NET框架。就目前而言，我们还必须将.NET框架和我们的应用程序一起分发给客户，也就是说安装程序需要安装一个.NET框架分发包。微软已经创建了一个免费的.NET框架分发包，大家可以到下面的地址获取它：*http://go.microsoft.com/fwlink/?LinkId=5584*。

绝大多数程序员都对运行时和类库比较熟悉。相信大家很多人对C运行时库、标准模板库(STL)、微软基础类库(MFC)、活动模板库(ATL)、Visual Basic运行时库或者Java虚拟机等都有所涉猎。实际上，Windows操作系统本身也可被看作一个运行时引擎和库。运行时和库为应用程序提供着各项服务，也是很多开发人员的最爱，因为我们不必再一次次地重新设计同样的算法。

Microsoft .NET框架为开发人员提供的技术比任何以前的微软开发平台提供的技术都要多，比如代码重用、代码专业化(code specialization)、资源管理、多语言开发、安全、部署、管理等。在设计.NET框架时，微软还感到有必要改进目前Windows平台的某些缺陷。下面的列表向大家描述了CLR和FCL提供的一部分服务：

- 一致的编程模型

  我们知道对于当前的Windows操作系统而言，某些功能需要通过动态链接库(DLL)来访问，而某些功能则需要通过COM对象来访问。然而，在.NET框架下，所有的应用程序服务都将以一种一致的、面向对象的编程模型提供给开发人员。

- 简化的编程方式

  CLR的其中一个目的就是简化Win32和COM环境下所需要的各种繁杂的基础构造。在CLR下，我们将可以从此远离如下这些概念：注册表、全局惟一标识符(GUID)、IUnknown、AddRef、Release、HRESULT等。注意，CLR并不是简单地对开发人员抽象这些概念；相反，CLR完全抛弃了这些概念。当然，如果我们编写的.NET框架应用程序要和现存的一些非.NET代码进行互操作，我们还必须对这些概念有足够的了解。

- 可靠的版本机制

  相信所有的Windows开发人员都对“DLL hell”版本问题比较熟悉。出现这个问题的根本原因在于为一个新应用程序所安装的组件覆盖了一个现有应用程序正在使用的组件，而其结果往往会导致现有的应用程序出现一些奇怪的行为，甚至不能正常工作。.NET框架采用了一种新型的版本机制来隔离应用程序组件，这种隔离策略可以保证一个应用程序总能加载它当

初生成和测试时所使用的组件。这使得应用程序在安装之后的任何时候，都能按期望的行为运行。新的版本机制彻底关上了“DLL hell”的大门。

- 轻便的部署管理

  当前的 Windows 应用程序都非常难以安装和部署。因为安装一个应用程序要照顾到许多事情：各种文件、注册表设置以及快捷链接。另外，要完全卸载一个应用程序更是几乎不可能完成的任务。虽然 Windows 2000 引入了一种新的安装引擎来帮助解决这些问题，但是发布软件安装包的公司仍然免不了在一些事情上出错。.NET 框架希望将这些问题彻底变成历史。在.NET 框架下，组件(或者说类型)将不再受注册表的任何引用。实际上，大多数.NET 框架应用程序的安装工作所需要的只不过是将文件拷贝到一个目录中，然后添加一个快捷链接到【开始】菜单、桌面以及【快速启动】栏而已。卸载应用程序也相当简单：直接删除它们就可以了。

- 广泛的平台支持

  当编译器编译面向.NET 框架的源代码时，它实际上产生的是通用中间语言(Common Intermediate Language，简称 CIL)。只有到了运行时，CLR 才会将这些 CIL 翻译为 CPU 指令。由于这个过程发生在运行时，所以它是面向特定的宿主 CPU 的。这意味着只要一台机器中包含有与 ECMA 兼容的 CLR 和 FCL，我们便可以将.NET 框架应用程序部署在该机器上。这样的机器可以是 x86、IA64、Alpha、PowerPC 等。当用户改变他们的硬件或者操作系统时，他们便会看到这样的技术带来的价值。

- 无缝的语言集成

  COM 允许不同的语言之间进行互操作。而.NET 框架则允许不同的语言之间进行无缝集成。在.NET 框架下，当我们使用一个类型时，不管它是用何种语言开发的，我们都可以像使用自己的语言开发的类型一样来使用它。例如，我们可以在 Visual Basic 中创建一个类，然后再在 C++中继承它。CLR 允许我们这样做是因为 CLR 要求所有面向它的语言都要遵循一种称作通用类型系统(Common Type System，简称 CTS)的规范。而通用语言规范(Common Language Specification，简称 CLS)则描述了一个语言要和其他的语言很好地集成在一起所必须要遵循的规范。微软自己已经提供了几个面向 CLR 的语言编译器：托管扩展 C++、C#、Visual Basic .NET(包括 Visual Basic 脚本即 VBScript，以及 Visual Basic for Applications 即 VBA)和 JScript。另外，其他一些公司和学术组织也正在开发面向 CLR 的语言编译器。

- 简便的代码重用

  使用上面所述的机制，我们可以很容易地创建一些类型来为第三方应用程序提供服务。.NET 框架使得代码的重用变得非常简单，同时也为组件(类型)厂商创造了一个巨大的市场。

- 自动化的内存管理(垃圾收集)

  程序设计是一项需要大量技巧和规则的活动，尤其在处理诸如文件、内存、屏幕空间、网络连接、数据库等资源的情况下更是如此。在这中间，最常见的一个 bug 就是因忘记释放某些资源而导致的不正常的运行行为。CLR 为此会为我们自动追踪资源的使用情况，从而确保应

用程序不致泄漏资源。实际上，在.NET 框架中，我们甚至没有办法显式“释放”内存。本书第 19 章将对 CLR 的垃圾收集原理有详细的解释。

- 坚实的类型安全

  CLR 可以确保所有的代码都是类型安全的。类型安全确保了系统所分配的对象总能够以正确的方式被访问。例如，假设一个方法声明的输入参数接受一个 4 字节的数值，那么 CLR 将会阻止我们向其传递一个 8 字节的数值。类似地，如果一个对象在内存中占用 10 个字节，那么应用程序将不可能把它当成多于 10 个字节的对象来读取。类型安全还意味着应用程序的执行流程只能向已经确知的位置传递(也就是真正的方法入口点)。换句话说，我们不可能构造一个指向某个内存位置的任意引用，然后便让应用程序从那个地址开始执行。总而言之，这些确保类型安全的措施减少了很多常见的编程错误和一些典型的系统攻击(例如，利用缓冲区溢出进行的攻击)。

- 丰富的调试支持

  许多编程语言都支持 CLR，这使得我们可以很容易采用最合适的语言来做它最擅长的工作。但是调试怎么办呢？答案是 CLR 完全支持跨语言调试。

- 统一的错误报告

  Windows 程序设计中最令人烦恼的一个问题就是各种不同的报告错误的方式。例如，一些函数通过返回 Win32 状态码来报告错误，一些函数通过返回 HRESULT 来报告错误，而还有一些函数则通过抛出异常来报告错误。在 CLR 中，所有失败的调用都是通过异常来报告的。异常使得我们能够将恢复代码和真正的应用程序逻辑代码分离开来实现。这种分离可以极大地简化代码的编写、阅读和维护。另外，CLR 中的异常还具有跨模块和跨语言的特性。而且和状态码与 HRESULT 不同的是，异常不能够被忽略。最后，CLR 还提供了内置的堆栈遍历机制，这使得我们可以很容易定位任何的 bug 和调用失败。

- 全新的安全策略

  传统操作系统的安全机制都是基于用户账号来提供隔离和访问控制的。这种机制虽然很有效，但从其本质上来讲，它假设的是所有的代码都具有相同的信任度。当所有的代码都从物理介质(例如 CD-ROM)、或者可信赖的公司网络上安装时，这种假设是正确的。但是随着当今计算平台对可移动代码(如 Web 脚本、从互联网上下载的应用程序以及电子邮件附件)的依赖的增加，我们就需要一种以代码为中心的控制方式。CLR 中的代码访问安全(CAS)为我们提供了这种方式的实现机制。

- 强大的互操作能力

  微软很清楚很多开发人员都有着无数现存的代码和组件。要重写所有这些代码来利用.NET 框架平台无疑将是一件巨大的工作，其结果往往会阻止开发人员对.NET 框架平台的接受速度。为此，.NET 框架从一开始就对访问现有 COM 组件，以及调用传统 DLL 中的 Win32 函数提供了完全的支持。

软件用户一般不会直接去欣赏 CLR 及其能力，但是他们很快将会注意到采用 CLR 的应用程序所

具有的品质和特性。另外，采用 CLR 的应用程序的开发时间和部署时间都要比原来 Windows 下的应用程序快许多，这一点也可以从用户的反馈和公司的营收报表中看得出来。

集成开发环境：Visual Studio .NET

整个.NET 平台创新的最后一个组成部分就是 Visual Studio .NET。Visual Studio 是微软耕耘多年的开发工具，并且随着.NET 平台的发布，它又引入了许多专门针对.NET 框架而设计的特性。支持 Visual Studio .NET 的操作系统包括 Windows NT 4、Windows 2000、Windows XP、Windows .NET 服务器家族产品，以及 Windows 的后续版本。而 Visual Studio .NET 产生的代码除了可以运行在上述操作系统上之外，还可以运行在 Windows 98、Windows 98 第 2 版以及 Windows Me 上。

和任何一种优秀的开发工具一样，Visual Studio .NET 也包括一个项目管理器、一个源代码编辑器、一个 UI 设计器、许多的向导、编译器、链接器、工具、实用程序、文档，以及调试器。它既支持创建面向 32 位和 64 位的 Windows 应用程序，也支持创建面向.NET 框架平台的应用程序。Visual Studio .NET 另一个重要的改进是对于所有的编程语言，它现在只有一个集成开发环境。

另外，微软还提供了一个.NET 框架 SDK。该 SDK 是免费的，它包括所有的语言编译器，以及很多工具和大量的文档。我们也可以不使用 Visual Studio .NET 而直接利用该 SDK 来开发出面向.NET 框架的应用程序。当然，这时候我们必须使用自己的代码编辑器和项目管理工具。我们也可能会因此而失去 Visual Studio .NET 为 Web 窗体和 Windows 窗体提供的便捷的拖拉式设计。就个人而言，我经常使用 Visual Studio .NET，因此我在本书中多次引用到了 Visual Studio .NET。但是 Visual Studio .NET 对于学习、使用和理解本书的所有概念并非必须，因为本书的主旨在程序设计，而非开发工具。

本书目标

本书的目标是解释如何开发面向.NET 框架的应用程序。具体而言，本书将解释 CLR 的工作机制，以及它提供的各种构造。本书还会讨论到 FCL 中的各个部分。当然，没有哪本书能够完全解释 FCL——它包括的类型有数千种，并且这个数目还在以惊人的速度增长。因此，本书仅将重点放在那些每个.NET 开发人员都需要理解的 FCL 核心类型上。需要提醒大家的是本书并不会讲述 Windows 窗体、XML Web 服务、Web 窗体等这些具体的应用程序模型，本书的内容适合于所有这些应用程序模型。

另外，我也不会在本书中讲述任何具体的编程语言，我假设大家至少熟悉一门编程语言，比如 C++、C#、Visual Basic 或者 Java。我还假设大家熟悉面向对象的一些概念，比如数据抽象、继承和多态。对这些概念的坚实理解对于掌握.NET 框架程序设计来说是至关重要的，因为所有.NET 框架的特性都是通过面向对象的范式来提供的。如果对这些概念还比较陌生，我强烈建议大家先找一本这方面的书来看一看。

虽然本书的目的不是讲述基本的编程技巧，但是我仍会在那些和.NET 框架密切相关的主题上有所着墨。所有的.NET 框架开发人员都要理解这些主题，本章不仅会讲解它们，还会在很多地方用到它们。

最后，虽然本书由于讲述的是.NET 框架中的通用语言运行时而不涉及具体的编程语言，但是为了演示其中的工作机理，我还必须提供许多代码示例。为了保持一个编程语言不可知论者(agnostic)的立场(译注：编程语言不可知论者认为编程语言并不是程序设计的全部，但又不否认编程语言的重要性)，我想最佳的选择就是 IL 汇编语言。IL 是 CLR 惟一理解的编程语言。所有其他的语言编译器都是先将源代码转换为 IL，然后再交由 CLR 处理。使用 IL，我们可以访问 CLR 提供的任何特性。

然而，IL 汇编语言是一种非常低级的语言，用来演示程序设计的概念有时候并不合适。因此我决定使用 C#作为我在本书中主要的编程语言。选择 C#是因为它是微软设计用来专门开发面向.NET 框架的编程语言。如果大家使用的不是 C#语言，那也没什么大碍——只要能读懂演示代码就行了。

系统要求

.NET 框架(译注：指仅支持.NET 应用程序运行的.NET 框架分发包)可以安装在 Windows 98、Windows 98 第 2 版、Windows Me、Windows NT 4、Windows 2000(所有版本)、Windows XP(所有版本)，以及 Windows .NET 服务器家族产品上。大家可以到 *http://go.microsoft.com/fwlink/?LinkId=5584* 上下载它。

.NET 框架 SDK 和 Visual Studio .NET 可以安装在 Windows NT 4(所有版本)、Windows 2000(所有版本)、Windows XP(所有版本)，以及 Windows .NET 服务器家族产品上。大家可以到 *http://go.microsoft.com/fwlink/?LinkId=77* 上下载.NET 框架 SDK。当然，Visual Studio .NET 就必须购买。

另外，大家还可以到 *http://www.Wintellect.com* 上下载本书的源代码。

完美无瑕

这个标题清楚地表达了我对本书的期望。但是大家都知道这是个真实的谎言。尽管如此，我和我的编辑仍然将“深度及时、通俗易懂和准确无误”确定为本书的目标，并为此投入了很多精力。然而既便有着一流的团队，事情总难免会出现纰漏。如果大家发现了书中任何的错误(尤其是 bug)，并能通过 *http://www.Wintellect.com* 发送给我，我将非常感谢。

支持信息

为确保本书的准确无误，各方都付出了很多努力。微软出版社在下面的 Web 站点上提供有图书的勘误服务。

http://www.microsoft.com/mspress/support/

大家也可以直接连到微软出版社的知识库上，来查询相关的问题，知识库的地址为：

http://www.microsoft.com/mspress/support/search.asp

如果大家对本书有任何的评论、问题、或者建议，也可以用下面的方法联系微软出版社：

邮政信箱：

Microsoft Press

Attn: Applied Microsoft .NET Framework Programming Editor

One Microsoft Way

Redmond, WA 98052-6399

电子信箱：

MSPINPUT@MICROSOFT.COM

另外，请注意上述邮件地址并不提供微软的产品支持。如果要获得 C#、Visual Studio、或者.NET 框架的支持信息，可以访问微软的产品标准支持站点：

http://support.microsoft.com

目　录

前言 XI

第Ⅰ部分

Microsoft .NET 框架基本原理

第 1 章　Microsoft .NET 框架开发平台体系架构 3

1.1　将源代码编译为托管模块 3
1.2　将托管模块组合为程序集 7
1.3　加载通用语言运行时 9
1.4　执行程序集代码 11
1.4.1　IL 与代码验证 19
1.5　.NET 框架类库 21
1.6　通用类型系统 24
1.7　通用语言规范 27
1.8　与非托管代码互操作 31

第 2 章　生成、打包、部署及管理应用程序与类型 35

2.1　.NET 框架部署目标 36
2.2　将类型生成为模块 37
2.3　将模块组合为程序集 45
2.3.1　使用 Visual Studio .NETIDE 为项目添加程序集引用 52
2.3.2　使用程序集链接器 53
2.3.3　在程序集中包含资源文件 55
2.4　程序集版本资源信息 56
2.4.1　版本号 59
2.5　语言文化 61
2.6　简单应用程序部署(私有部署程序集) 63
2.7　简单管理控制(配置) 64

第 3 章　共享程序集 71

3.1　两种程序集、两种部署方式 72
3.2　强命名程序集 73
3.3　全局程序集缓存 79
3.3.1　GAC 的内部结构 85
3.4　引用强命名程序集 87
3.5　强命名程序集的防篡改特性 89
3.6　延迟签名 90
3.7　强命名程序集的私有部署 95
3.8　并存执行 96
3.9　CLR 如何解析类型引用 98
3.10　高级管理控制(配置) 101
3.10.1　发布者策略控制 106
3.11　修复错误的应用程序 109

第Ⅱ部分

类型与通用语言运行时

第 4 章　类型基础 115

4.1　所有类型的基类型：System.Object 115
4.2　类型转换 117
4.2.1　使用 is 和 as 操作符转型 119
4.3　命名空间与程序集 121

第 5 章 基元类型、引用类型与值类型 127
5.1 基元类型 127
5.1.1 Checked 与 Unchecked 基元类型操作 131
5.2 引用类型与值类型 134
5.3 值类型的装箱与拆箱 141
第 6 章 通用对象操作 153
6.1 对象的等值性与惟一性 153
6.1.1 为基类没有重写 Object.Equals 方法的引用类型实现 Equals 154
6.1.2 为基类重写了 Object.Equals 方法的引用类型实现 Equals 156
6.1.3 为值类型实现 Equals 方法 157
6.1.4 Equals 方法与 ==/!= 操作符的实现总结 160
6.1.5 对象惟一性识别 161
6.2 对象的散列码 162
6.3 对象克隆 164
第III部分 类型设计
第 7 章 类型成员及其访问限定 169
7.1 类型成员 169
7.2 访问限定修饰符和预定义特性 173
7.2.1 类型预定义特性 174
7.2.2 字段预定义特性 175
7.2.3 方法预定义特性 175
第 8 章 常数与字段 177
8.1 常数 177
8.2 字段 178
第 9 章 方法 181
9.1 实例构造器 181
9.2 类型构造器 187
9.3 操作符重载方法 190
9.3.1 操作符与语言互操作性 193
9.4 转换操作符方法 197
9.5 引用参数 200
9.6 可变数目参数 206
9.7 虚方法的调用机理 209
9.8 虚方法的版本问题 210
第 10 章 属性 215
10.1 无参属性 215
10.2 含参属性 220
第 11 章 事件 227
11.1 发布事件 228
11.2 侦听事件 234
11.3 显式控制事件注册 236
11.4 在一个类型中定义多个事件 238
11.5 设计 EventHandlerSet 类型 243
第IV部分 基本类型
第 12 章 文本处理 249
12.1 字符 249
12.2 System.String 类型 253
12.2.1 创建字符串 253
12.2.2 字符串的恒定性 255
12.2.3 字符串比较 256
12.2.4 字符串驻留 262
12.2.5 字符串池技术 266
12.2.6 查看字符串中的字符 266
12.2.7 其他字符串操作 270
12.3 高效地动态创建字符串 270

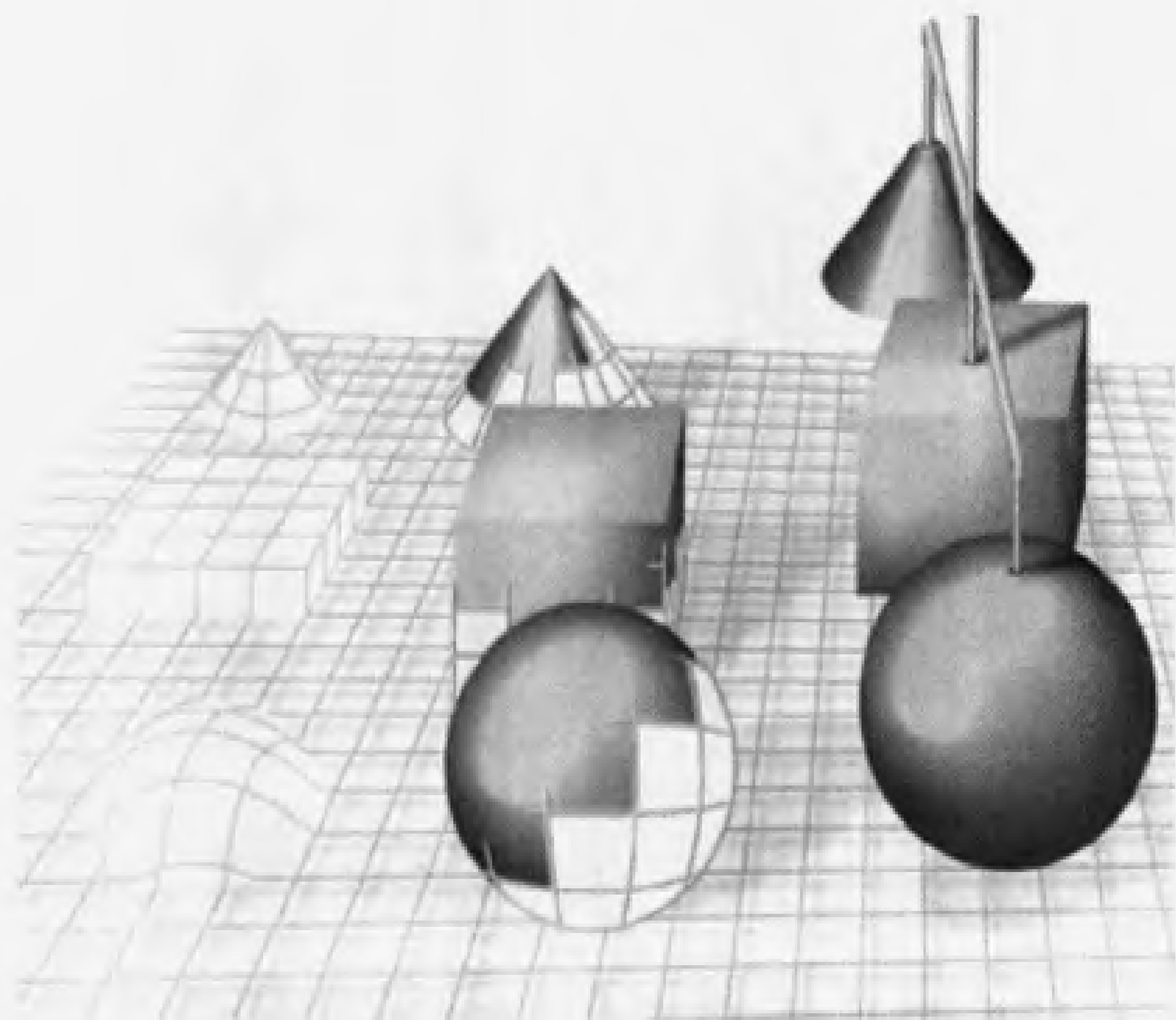
12.3.1 构造 StringBuilder 对象 271
12.3.2 StringBuilder 的成员 272
12.4 获取对象的字符串表达形式 275
12.4.1 特定格式与语言文化 276
12.4.2 将多个对象格式化为一个字符串 280
12.4.3 提供自定义格式化器 282
12.5 通过解析字符串获取对象 285
12.6 编码：字符与字节之间的转换 289
12.6.1 字符与字节的编码/解码流 296
12.6.2 Base-64 字符串编码与解码 298
第 13 章 枚举类型与位标记 299
13.1 枚举类型 299
13.2 位标记 305
第 14 章 数组 309
14.1 所有数组的基类：System.Array 312
14.2 数组的转型 315
14.3 数组的传递与返回 316
14.4 创建下限非 0 的数组 318
14.5 快速数组访问 319
14.6 重新调整数组长度 323
第 15 章 接口 325
15.1 接口与继承 325
15.2 设计支持插件组件的应用程序 331
15.3 使用接口改变已装箱值类型中的字段 333
15.4 实现多个有相同方法的接口 336
15.5 显式接口成员实现 338
第 16 章 定制特性 345
16.1 使用定制特性 345
16.2 定义自己的特性 349
16.3 特性构造器与字段/属性的数据类型 353
16.4 检测定制特性 354
16.5 特性实例间的匹配 359
16.6 伪定制特性 362
第 17 章 委托 365
17.1 认识委托 365
17.2 使用委托回调静态方法 368
17.3 使用委托回调实例方法 370
17.4 委托揭秘 371
17.5 委托史话：System.Delegate 与 System.MulticastDelegate 375
17.6 委托判等 376
17.7 委托链 377
17.8 C#对委托链的支持 383
17.9 对委托链调用施以更多的控制 384
17.10 委托与反射 386
第 V 部分 类型管理
第 18 章 异常 393
18.1 异常处理的演化 394
18.2 异常处理机制 396
18.2.1 try 块 397
18.2.2 catch 块 398
18.2.3 finally 块 400
18.3 异常的本质 401
18.4 System.Exception 类 406
18.5 FCL 定义的异常类 408
18.6 定义自己的异常类 411

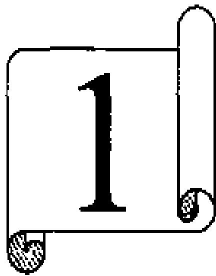
18.7 如何正确使用异常 416
18.7.1 避免过多的 finally 块 416
18.7.2 避免捕获所有异常 418
18.7.3 从异常中顺利地恢复 419
18.7.4 当异常无法修复时，回滚部分完成的操作 420
18.7.5 隐藏实现细节 421
18.8 FCL 中存在的一些问题 424
18.9 性能考虑 426
18.10 捕获筛选器 429
18.11 未处理异常 432
18.11.1 发生未处理异常时的CLR 行为控制 437
18.11.2 未处理异常与Windows 窗体 439
18.11.3 未处理异常与 ASP.NET Web 窗体 440
18.11.4 未处理异常与 ASP .NET XML Web 服务 441
18.12 异常堆栈踪迹 441
18.12.1 远程堆栈踪迹 444
18.13 异常调试 445
18.13.1 告诉 Visual Studio 调试何种代码 448
第 19 章 自动内存管理(垃圾收集) 451
19.1 垃圾收集平台基本原理解析 451
19.2 垃圾收集算法 455
19.3 终止化操作 459
19.3.1 调用 Finalize 方法的条件 467
19.3.2 终止化操作的内部机理 468
19.4 Dispose 模式：强制对象清理资源 471
19.4.1 使用实现了 Dispose 模式的类型 477
19.4.2 C#的 using 语句 482
19.4.3 一个有趣的依赖问题 484
19.5 弱引用 485
19.5.1 弱引用的内部机理 487
19.6 对象复苏 489
19.6.1 利用复苏设计一个对象池 491
19.7 对象的代龄 493
19.8 编程控制垃圾收集器 499
19.9 其他一些与垃圾收集器性能相关的问题 501
19.9.1 省却同步控制的多线程分配 503
19.9.2 可扩展并行收集 503
19.9.3 并发收集 504
19.9.4 大尺寸对象 505
19.10 监视垃圾收集 506
第 20 章 CLR 寄宿、应用程序域、反射 507
20.1 元数据：.NET 框架的基石 507
20.2 CLR 寄宿 508
20.3 应用程序域 510
20.3.1 跨越应用程序域边界访问对象 513
20.3.2 应用程序域事件 515
20.3.3 应用程序及其如何寄宿 CLR 和管理应用程序域 516
20.3.4 Yukon 517
20.4 反射概要 518
20.5 反射一个程序集中的类型 520
20.6 反射一个应用程序域中的程序集 523

20.7 反射一个类型的成员：绑定 523
20.8 显式加载程序集 525
20.8.1 将程序集象“数据文件”一样加载 527
20.8.2 建立一个异常类型的层次结构 529
20.9 显式卸载程序集：卸载应用程序域 532
20.10 获取一个 System.Type 对象的引用 534
20.11 反射一个类型的成员 538
20.11.1 创建一个类型的实例 541
20.11.2 调用一个类型的方法 543
20.11.3 一次绑定、多次调用 548
20.12 反射一个类型的接口 553
20.13 反射的性能 555
索引 557
术语表 592

第 I 部分

Microsoft .NET 框架基本原理





Microsoft .NET 框架开发平台体系架构

Microsoft .NET 框架引入了许多新的概念、技术和术语。本章的目标是让大家对.NET 框架的体系架构有一个总体的认识，并对.NET 框架中出现的一些新的技术和术语有一个基本的了解。此外，本章还会向大家展示将源代码生成(build)为应用程序或一组可分发组件(类型)的全过程，同时还会对这些组件的执行原理做详细的解释。

1.1 将源代码编译为托管模块

很好，大家已经决定使用.NET 框架作为自己的开发平台了。首先，我们需要确定希望创建何种类型的应用程序或组件。假设我们已经完成了这些次要的细节，所有的设计都做好了，规格说明也完成了，一切准备就绪。

现在我们必须决定使用哪种编程语言。这个任务通常很难，因为不同的语言有不同的能力。比如在非托管 C/C++中，我们可以对系统有着相当底层的控制，我们可以以自己喜欢的方式来管理内存，如果需要的话还可以很方便地创建好几个线程，等等。另一方面，利用 Visual Basic 6，我们可以非常高效地创建用户界面(UI)应用程序，控制 COM 对象和数据库也相当容易。

.NET 框架的核心便是通用语言运行时(Common Language Runtime，简称 CLR)，顾名思义它是一个可被各种不同的编程语言所使用的运行时。CLR 的很多特性可用于所有面向它的编程语言。比如，如果 CLR 用异常来报告错误，那么所有面向它的语言都将通过异常来得到错误报告。如果 CLR 允许我们创建线程，那么所有面向它的语言也都可以创建线程。

实际上，CLR 在运行时对开发人员用何种编程语言来完成源代码一无所知。这意味着我们应该选择那些能够最容易表达我们意图的编程语言。我们可以用任何自己喜欢的语言来编写代码，前提是我们使用的编译器能够编译面向 CLR 的代码。

如果上面所说的是正确的，那么在编程语言中做这样那样的选择又有何益处？我们可以将编译器看作是一个语法检查器和“正确代码”的分析器。它们对我们的源代码进行检查，确保我们编写的所有东西都有意义，最后输出描述我们意图的指令序列。不同的编程语言允许我们使用不同的语法进行开发。不要低估这种选择的价值。例如，对于数学或者金融应用，在表达相同意图的情况下，采用 APL 语法相较于采用 Perl 语法能够节省好几天的开发时间。

微软已经创建了几种面向 CLR 的语言编译器：托管扩展 C++、C#(念作“C Sharp”)、Visual Basic、JScript、J#(一个 Java 语言编译器)，以及一个中间语言(Intermediate Language，简称 IL)汇编器。除此之外，其他一些公司也在创建面向 CLR 的编译器，就我所知道的有 Alice、APL、COBOL、Component Pascal、Eiffel、Fortran、Haskell、Mercury、ML、Mondrian、Oberon、Perl、Python、RPG、Scheme，以及 Smalltalk。

图 1.1 演示了源代码文件的编译过程。如图所示，我们可以用任何支持 CLR 的编程语言来创建源代码文件。然后用相应的编译器来做语法检查和源代码分析。但是不管使用的是何种编译器，最后生成的结果都是一个**托管模块**(managed module)。托管模块是一个需要 CLR 才能执行的标准 Windows 可移植可执行(portable executable，简称 PE)文件。随着时间的推移，其他一些操作系统也有可能采用这种 PE 文件格式。

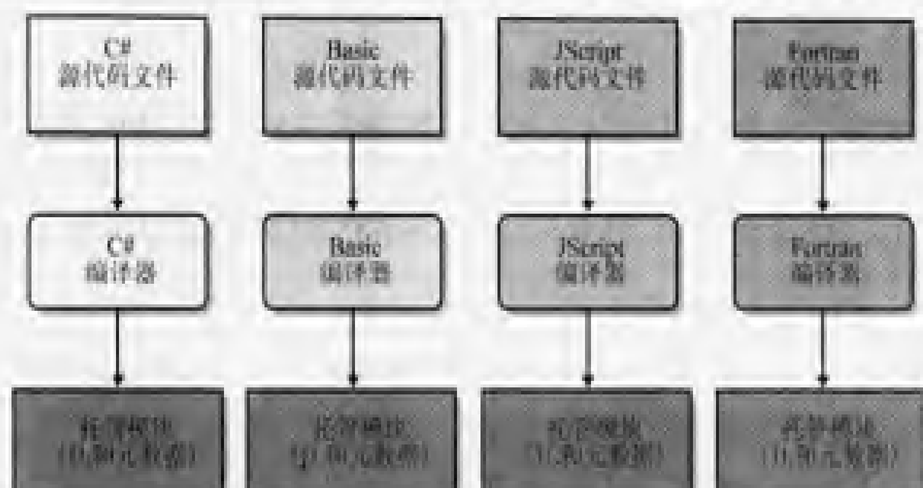


图1.1 将源代码编译为托管模块

表1.1描述了一个托管模块的各个组成部分。

表1.1 托管模块的组成部分

部 分	描 述
PE表头	标准Windows PE文件表头，类似于通用对象文件格式(Common Object File Format，简称COFF)表头。该表头指出了文件的类型：GUI(图形用户界面)，CUI(控制台用户界面)，或者DLL(译注：DLL在以前表示Windows中的动态链接库文件，即Dynamic Link Library，它是以动态链接的方式提供的一组函数库。在.NET平台中，由于和传统的动态链接库文件有着相同的扩展名.dll，所以DLL的名称被沿用了下来，但其含义已有所改变。.NET中的DLL特指程序集文件的一种形式)。另外该表头还包括一个时间标记用于表示文件创建的时间。对于仅包含IL代码的模块，该表头的大多数信息会被忽略。对于包含有本地CPU代码的模块，该表头还会包含有关本地CPU代码的一些信息
CLR表头	包含标识托管模块的一些信息(可以被CLR或者一些实用工具解析)。这些信息包括托管模块所需要的CLR版本号，一些标记，托管模块入口点方法(Main方法)的MethodDef元数据标记，以及有关托管模块的元数据、资源、强命名、标记和其他一些意义不是太大的信息的位置和尺寸
元数据	每个托管模块都包含有一些元数据表。元数据表主要分两种，一种用于描述源代码中定义的类型和成员，一种用于描述源代码中引用的类型和成员
中间语言(IL)代码	编译器在编译源代码时产生的指令。CLR在运行时会将IL代码编译成本地CPU指令

大多数早先的编译器产生的代码都是面向特定 CPU 体系的，例如 X86、IA64、Alpha、或 PowerPC。而所有与 CLR 兼容的编译器产生的都是 IL 代码(本章稍后将会深入探究 IL 代码的一些细节)。由于生存期和执行行为受 CLR 管理的缘故，IL 代码有时也被称作**托管代码**(managed code)。

除了产生 IL 外，所有面向 CLR 的编译器都需要为托管模块产生完整的**元数据**(metadata)。简单地说，元数据就是一个数据表的集合，在这些表中，其中一些用于描述托管模块中所定义的内容(比如所定义的类型和它们的成员)，另外还有一些用于描述托管模块中所引用的内容(比如被引用的类型和它们的成员)。元数据是一些早先的技术如类型库、接口定义语言(IDL)文件的一个超集。需要指出的是 CLR 的元数据远比它们完整。而且不像类型库和 IDL，元数据总是和包含 IL 代码的文件相关联。实际上，元数据总是和这些代码一起被嵌入到同一个 EXE/DLL 文件中，两者根本不可能分离。因为编译器总是同时产生元数据和 IL 代码，并且总是同时将它们嵌入到生成的托管模块中，所以元数据和它所描述的 IL 代码之间总能保持同步。

元数据有很多用处。下面列出了其中的一些：

- 元数据省去了源代码编译时对头文件和库文件的需求，这是因为在含有实现类型和成员的 IL 代码文件中，已经包含了所有被引用的类型和成员的信息。编译器可以直接从托管模块中读取元数据来获得这些信息。
- Visual Studio .NET 可以利用元数据来辅助我们编写代码。它的智能感知(IntelliSense)特性就是通过分析元数据来告诉我们某个类型提供了哪些方法，以及某个方法有哪些参数。
- CLR 的代码验证过程可以利用元数据来确保代码仅执行“安全”的操作。(稍后将会探讨代码验证。)
- 利用元数据，我们可以将一个对象的字段序列化到一个内存块中，然后远程传送给另一台机器，最后再在远程机器上执行反序列化，从而重新创建对象和它的状态。
- 利用元数据，垃圾收集器可以追踪对象的生存期。对于任何对象，垃圾收集器都能够通过元数据来确定该对象的类型，并且可以获知该对象的哪些字段引用了其他对象。

本书第 2 章将探讨有关元数据的更多细节。

微软的 C#、Visual Basic、JScript、J# 编译器和 IL 汇编器总是产生需要 CLR 才能执行的托管模块。要执行这些托管模块，终端用户必须在他们的机器上安装 CLR。这和我们需要安装 MFC(Microsoft Foundation Class)库或者 Visual Basic DLL 来运行 MFC 或者 Visual Basic 6 应用程序的道理是类似的。

默认情况下，微软的 C++编译器生成的是非托管模块，即我们已经熟悉的 EXE 或者 DLL 文件(译注：这里指的都是传统的可执行文件或者动态连接库文件，而非.NET 中的程序集文件)。它们的执行不需要 CLR。然而，通过指定一个新的命令行开关，C++编译器也能够产生出需要 CLR 才能执行的托管模块。在上面提到的所有微软的编译器中，C++是比较独特的，它允许开发人员可以同时编写托管代码和非托管代码，并且可以将它们生成到同一个模块中。这是一个很棒的特性，因为它允许开发人员能够先用托管代码来编写应用程序的绝大部分(往往出于类型安全和组件互操作的缘故)，然后再在非托管 C++代码中访问它们。

1.2 将托管模块组合为程序集

CLR 实际上并不和托管模块打交道，它直接打交道的对象是**程序集**(assembly)。程序集是一个抽象的概念，刚开始往往很难理解。首先，程序集是一个或多个托管模块，以及一些资源文件的逻辑组合。其次，程序集是组件复用，以及实施安全策略和版本策略的最小单位。根据我们对编译器和相关工具所做的选择，程序集可以是一个文件或者多个文件。

本书第 2 章将探讨有关程序集的更多细节，所以这里不会对该话题花费过多的时间。目前大家只需要知道有这样一个概念性的东西，它提供了一种方式允许我们将一组文件看作一个单独的实体。

图 1.2 应该能够有助于解释程序集是什么。在该图中，一些托管模块和资源(或者数据)文件被一个工具所处理。该工具产生一个 PE 文件来表示所有文件的逻辑组合。需要注意的是该 PE 文件中包含了一个称作**清单**(manifest)的数据块。所谓的清单仅仅是另外一些元数据表的集合。这些表描述了组成程序集的文件，程序集所有文件中实现的公有导出类型，以及一些和程序集相关的资源文件或数据文件。

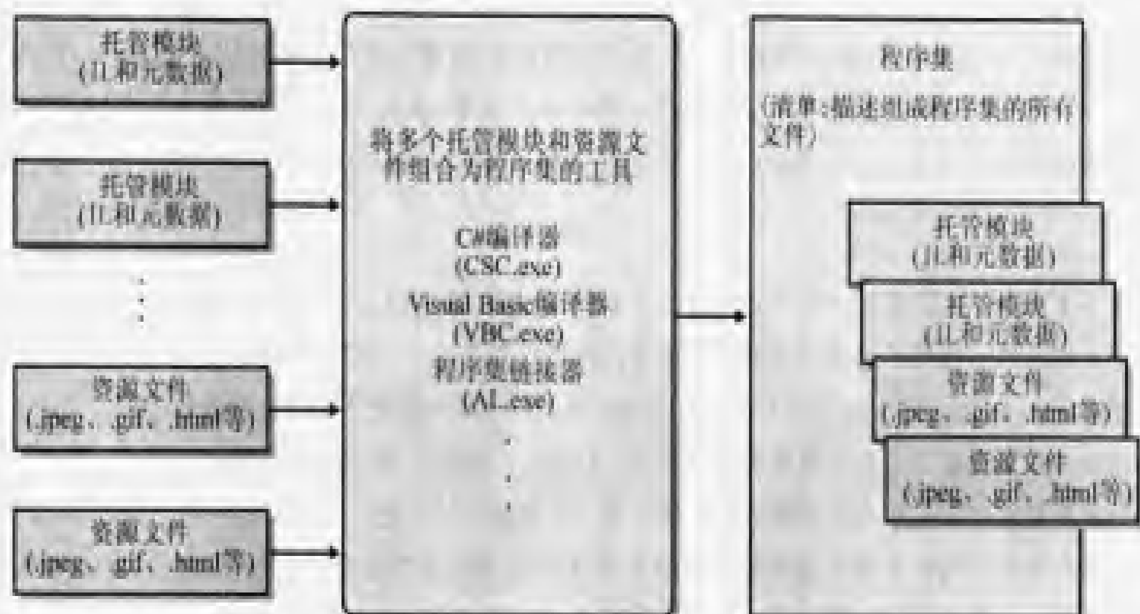


图 1.2　将托管模块组合为程序集

默认情况下，编译器会将产生的托管模块转换为一个程序集。也就是说，C#编译器产生的是一个包含了清单的托管模块，其中的清单表明程序集中仅包含一个文件。对于仅包含一个托管模块、并且没有资源(或者数据)文件的项目来说，程序集就是托管模块，而且在创建过程中，不需要执行任何其他的步骤。如果希望将一组文件合并到一个程序集中，我们需要掌握一些更多的工具(比如程序集链接器，Assembly Linker，即 AL.exe)和它们的命令行选项。本书第 2 章将解释这些工具和选项。

对于一个可重用、可部署、可实施版本管理的组件来说，程序集允许我们分离它的逻辑表示和物理表示。如何将代码和资源划分到不同的文件中完全取决于我们。例如，我们可以将一些很少使用的类型或者资源放在一些单独的程序集文件中，然后根据需要从 Web 上下载它们。如果没有用到这些文件，它们将不会被下载，这样既节省了磁盘空间，也减少了安装时间。程序集允许我们将文件的部署分解开来，同时又将所有的文件视作一个单独的集合。

程序集中的模块还包含它所引用的程序集的一些信息(如版本号信息)。这些信息使得一个程序集得以实现**自描述**(self-describing)。换句话说，CLR 知道执行一个程序集所需要的所有内容，它不需要

再在注册表或者活动目录(Active Directory)中获取额外的信息。因此，程序集的部署要比非托管组件的部署容易得多。

1.3 加载通用语言运行时

一个程序集或者是一个可执行应用程序，或者是一个包含供可执行应用程序使用的一组类型(组件)的 DLL。CLR 负责管理包含在程序集中的代码的执行。这意味着宿主机器必须安装.NET 框架。微软已经创建了一个可以将.NET 框架免费安装到客户机器上的分发包。.NET 框架最终会和将来的 Windows 打包在一起，这样我们就不再需要将它和我们的程序集放在一起来发布了。

我们可以通过在 %windir%\system32 目录下查找 MSCorEE.dll 文件来判断一个机器中是否安装了.NET 框架。如果一个机器中存在该文件，则证明该机器上已经安装了.NET 框架。多个版本的.NET 框架可以安装在同一个机器内。如果想确定一个机器中安装了哪些版本的.NET 框架，可以查看下面的注册表键下的子键：

```
HKEY_LOCAL_MACHINE \ SOFTWARE \ Microsoft \ .NETFramework \ policy
```

当生成一个 EXE 程序集时，编译器/链接器会产生一些特殊的信息，并将它们嵌入到结果程序集的 PE 文件表头及其各个组成文件的.text 部分。当 EXE 文件被调用时，这些特殊的信息将导致 CLR 被加载并初始化。CLR 随后会定位到应用程序的入口点方法，从而以此来启动应用程序。

类似地，如果是一个非托管应用程序通过调用 LoadLibrary 来加载一个托管程序集，那么该托管程序集 DLL 的入口点函数也会知道去加载 CLR 来处理包含在其中的代码。

大多数时候，我们并不需要知道或者理解 CLR 是如何被加载的。对于大多数程序员，这些特殊信息仅仅意味着使得应用程序得以启动，无需更多的考虑。然而为了满足某些人士的好奇，本节的剩余部分将对托管 EXE 或 DLL 如何启动 CLR 做一解释。如果大家对此不感兴趣，完全可以跳过本节。如果大家对创建寄宿(host)CLR 的非托管应用程序感兴趣，可参见本书第 20 章。

图1.3总结了一个托管EXE加载并初始化CLR的过程。

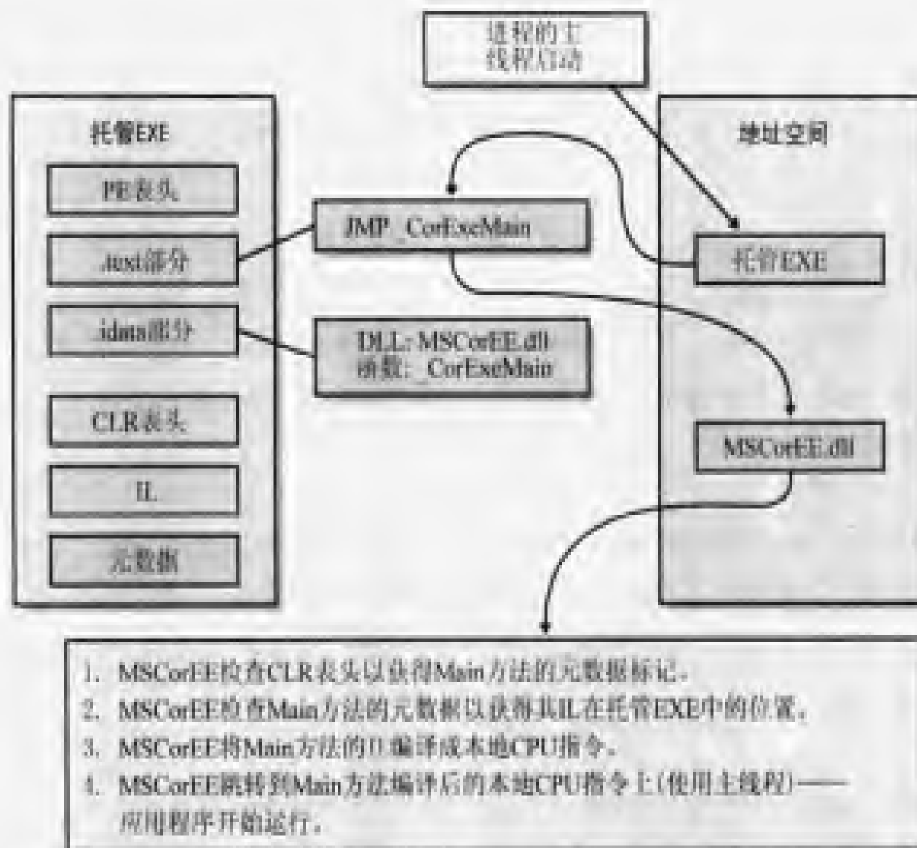


图1.3 加载并初始化CLR

当编译器/链接器创建一个可执行程序集时，下面这个6字节的x86 stub函数将被嵌入到PE文件的.text部分：

```
JMP _CorExeMain
```

由于_CorExeMain函数是从MSCorEE.dll动态链接库中导入的，所以MSCorEE.dll将在程序集文件的导入(.idata)部分被引用。MSCorEE.dll表示**微软组件对象运行时执行引擎**(Microsoft Component Object Runtime Execution Engine)。当托管EXE文件被调用时，Windows将像对待通常(非托管)的EXE文件一样来对待它。Windows加载器首先加载该文件，然后检查其.idata部分发现MSCorEE.dll需要被加载到进程的地址空间，于是加载器获取MSCorEE.dll中_CorExeMain函数的地址，同时修正托管EXE文件中stub函数的JMP指令。

之后，进程的主线程开始执行修正后的x86 stub函数，该stub函数立即跳转到MSCorEE.dll中的_CorExeMain函数上。_CorExeMain函数接着初始化CLR，并查看可执行程序集的CLR表头以确定要执行的托管入口点方法。入口点方法找到以后，其IL代码随之被编译为本地CPU指令。最后，CLR跳转到编译后的本地指令上(使用进程的主线程)。直到这时，托管应用程序才算开始真正运行。

托管 DLL 的情形与此类似。当生成托管 DLL 时，编译器/链接器将会为 DLL 程序集 PE 文件的 .text 部分产生一个类似的 6 字节的 x86 stub 函数：

```
JMP _CorDllMain
```

_CorDllMain 函数也是从 MSCorEE.dll 中导入的，因此托管 DLL 中的 .idata 部分也包含有对 MSCorEE.dll 的引用。当 Windows 加载托管 DLL 时，它将自动加载 MSCorEE.dll(如果它还没有被加载)，然后获取_CorDllMain 函数的地址，并修正托管 DLL 中 x86 stub 函数的 JMP 指令。之后，调用 LoadLibrary 加载托管 DLL 的线程将跳转到该托管 DLL 中的 x86 stub 函数上，该 stub 函数接着又立即跳转到 MSCorEE.dll 中的_CorDllMain 函数上。_CorDllMain 初始化 CLR(如果 CLR 还没有为该进程初始化)后便立即返回，应用程序也返回到正常状态并继续运行。(译注：这里描述的 Windows 加载托管 DLL 的过程特指用非托管代码加载托管 DLL——通常使用 LoadLibrary 函数——的过程。而对于托管代码调用托管 DLL 的情况，它通常首先检测托管 DLL 中的元数据，然后便以即时编译的方式执行其内方法的 IL 代码，stub 函数及其跳转过程将被忽略。另外注意这里是托管 DLL 的加载过程，而非其内方法的 IL 代码的执行过程，所以这里的_CorDllMain 在初始化 CLR 后会立即返回。而其内方法的 IL 代码是在非托管代码明确调用托管 DLL 中的方法后才开始以即时编译的方式执行的。)

由于 Windows 98、Windows98 第 2 版、Windows Me、Windows NT 和 Windows 2000 这些操作系统都是在 CLR 之前发布的，所以在它们之上运行托管程序集都需要上述 6 字节的 x86 stub 函数。注意这里 6 字节的 stub 函数专门用于 x86 机器。如果将 CLR 移植到其他 CPU 体系上，该 stub 函数将无法正常运行。由于 Windows XP 和 Windows .NET 服务器家族同时支持 x86 和 IA64 CPU 体系，所以它们的加载器针对托管程序集进行了专门的改动。

在 Windows XP 和 Windows .NET 服务器家族上，当一个托管程序集被调用时(通常通过 CreateProcess 或者 LoadLibrary)，操作系统的加载器会通过查看 PE 文件表头的第 14 条目录项(参见 WinNT.h 文件中的 IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR)来检测该文件中是否包含托管代码。如果该目录项存在且不为 0，那么加载器将忽略该文件的导入部分(.idata)，而自动将 MSCorEE.dll 加载到进程的地址空间内。一旦加载完成，操作系统的加载器将使该进程的线程直接跳转到 MSCorEE.dll 中相应的函数上。6 字节的 x86 stub 函数在运行 Windows XP 和 Windows .NET 服务器家族的操作系统上将完全被忽略。

最后需要注意的是，托管 PE 文件总是使用 32 位(而非 64 位)的 PE 文件格式。在 64 位的 Windows 系统上，操作系统加载器检测到 32 位的托管 PE 文件后会自动创建 64 位的地址空间。

1.4 执行程序集代码

如前所述，托管模块中包含着元数据和 IL 代码。IL 是由微软在咨询了一些商业和学术上的语言编译器作者之后开发的一种独立于 CPU 的机器语言。IL 要比大多数 CPU 机器语言高级得多，它可以

理解对象类型，并且拥有很多高级的指令，这些指令可以创建和初始化对象，调用对象上的虚方法，以及直接操作数组元素。它甚至还有抛出和捕获异常的指令。我们可以把 IL 视作一种面向对象的机器语言。

通常情况下，开发人员会使用一门高级语言(比如 C#或 Visual Basic)来编写程序。这些高级语言的编译器产生的将是 IL 代码。然而，和其他的机器语言一样，我们也可以直接以汇编语言的方式编写 IL 程序。并且微软也确实为我们提供了一个 IL 汇编器，即 ILAsm.exe。另外，微软还提供了一个反汇编器，即 ILDasm.exe。

IL 与知识产权的保护

有些人担心 IL 不能为他们的算法提供足够的知识产权保护。换句话说，他们担心某些人会使用一些工具，比如 IL 反汇编器，来反编译自己开发的托管模块中的代码。

的确，IL 代码比大多数其他的汇编语言要更高级，对它的反编译也相对比较简单。但是，如果就 XML Web 服务(XML Web Services)或者 Web 窗体(Web Froms)应用程序来讲，由于它们的托管模块都是驻留在公司自己的服务器上，公司以外的任何人都无法获取这些模块，因此也就不可能利用工具来查看其中的 IL 代码——这自然也就不存在知识产权的安全问题。

对于向外发布的托管模块，我们则可以从第三方厂商那里获取一个混淆器(obfuscator)工具。这些工具可以“搅乱”托管模块元数据内所有私有符号的名称。要破译被“搅乱”的名称并且理解每一个方法的意思通常是很困难的。但是我们也要注意这些混淆器工具提供的保护是有限的，因为在某些地方我们还必须保证 IL 代码是可用的，毕竟 CLR 还要处理它们。

如果这类混淆器不能为我们提供期望的知识产权保护，我们还可以考虑在一些非托管模块中实现较为敏感的算法——这些非托管模块包含的将是本地 CPU 指令，而非 IL 代码和元数据。然后我们再利用 CLR 的互操作功能来实现应用程序中托管部分和非托管部分的通信。当然，这里的前提假设是不必担心有人反编译非托管代码中的本地 CPU 指令。

需要注意的是，任何高级语言大多数情况下提供的都只是 CLR 全部功能的一个子集。但 IL 汇编语言却允许开发人员获取 CLR 所有的功能。因此，如果我们选择的编程语言没有提供我们所需要的 CLR 的某些功能，我们则可以选择 IL 汇编语言或者另一个提供了该功能的高级语言来编写这部分程序。

了解 CLR 提供了哪些功能的惟一途径就是阅读有关 CLR 的文档。本书重点讲述 CLR 的一些特性，以及它们在 C#语言中哪些是如何提供的，哪些是没有提供的。我估计很多其他的书籍和文章都会从一门语言的角度去探讨 CLR，并且大多数开发人员也倾向于相信 CLR 仅提供了他选择的语言所展现的功能。如果大家选择的语言能够实现想做的事情，那么这种理解也并非坏事，虽然有些混乱。

重要 我个人认为这种允许在编程语言之间方便切换和高度集成的能力是 CLR 一个很厉害的特性。不幸的是，许多开发人员经常会忽视这个特性。我们知道，C#和 Visual Basic 善于进行 I/O 操作，而 APL 在做高级工程和金融计算时则非常突出。通过 CLR，我们可以用 C#来编写应用程序的 I/O 部分，然后再用 APL 来编写应用程序的工程计算部分。CLR 提供的语言之间的集成能力是前所未有的，这使得混合语言编程值得许多开发项目考虑。

另一个关于 IL 需要记住的要点是它并不束缚于任何特定的 CPU 平台，这意味着一个包含 IL 代码的托管模块有可能运行在任何 CPU 平台上——当然这里的前提是运行在该 CPU 平台上的操作系统中寄宿(host)了某个版本的 CLR。尽管初始发布的 CLR 仅运行在 32 位的 Windows 平台上，但用托管 IL(译注：尽管作者这里使用了“托管 IL”这一术语，但是并没有与之相对应的“非托管 IL”。目前所有的 IL 代码都是托管 IL，其执行都要经由 CLR 的管理。注意“非安全 IL 代码”并不是非托管代码，它是不能验证安全的托管代码。)来编写应用程序仍将使得开发人员更加独立于底层 CPU 体系结构。

.NET 框架的标准化

2000 年 10 月，微软联合英特尔(Intel)和惠普(HP)作为共同倡议者向欧洲计算机制造商协会(European Computer Manufacturer's Association，简称 ECMA)提议了.NET 框架中一个较大的子集以实现标准化。ECMA 接受了该提议，并创立了一个技术委员会(TC39)指导标准化进程。该技术委员会担负着以下职责：

- **技术小组 1** 开发动态脚本语言标准(ECMAScript)。JScript 是微软对该脚本的一个实现。
- **技术小组 2** 开发 C#编程语言标准。
- **技术小组 3** 开发通用语言基础构造(Common Language Infrastructure，简称 CLI)。CLI 是 CLR 和.NET 框架类库的一个子集。具体而言，CLI 定义了文件格式、通用类型系统、可扩展元数据系统、中间语言以及对底层平台的访问(即 P/Invoke)共五项标准。另外，CLI 还将定义一个允许多种编程语言使用的、裁减了的基础类库(主要用于小型硬件设备)。

一旦标准化工作完成，这些标准将被递交给 ISO/IEC JTC1(Information Technology)。同时，ECMA 技术委员会还会进一步调研 CLI、C#和 ECMAScript 未来的发展方向，并接受一些补充或附加的技术提议。关于 ECMA 的更多信息，可参见 *http://www.ECMA.ch* 和 *http://MSDN.Microsoft.com/Net/ECMA*。(译注:C#和 CLI 于 2003 年 4 月已经被批准成为 ISO 标准，其标准号分别为 ISO/IEC 23270 和 ISO/IEC 23271。)

随着 CLI、C#和 ECMAScript 标准化的完成，微软将不再"拥有"任何这些技术。微软仅仅是实现这些技术的众多公司(希望如此)中的一员。当然，微软希望他们的实现在性能和满足用户个性化需求方面能够达到最优。这些产品也将有助于 Windows 的销售，因为微软最好的实现往往只会在 Windows 上运行。然而，其他公司也可以实现这些标准，并同微软进行竞争，而且完全有可能在竞争中取胜。

尽管目前的 CPU 还不能直接执行 IL 指令，但将来的 CPU 也许会具有这种能力。就目前来说，要执行一个方法，它的 IL 代码还必须首先被转换成本地 CPU 指令。这属于 CLR 中即时(just-in-time，简称 JIT)编译器的工作。

图 1.4 展示了一个方法第一次被调用时的情况。

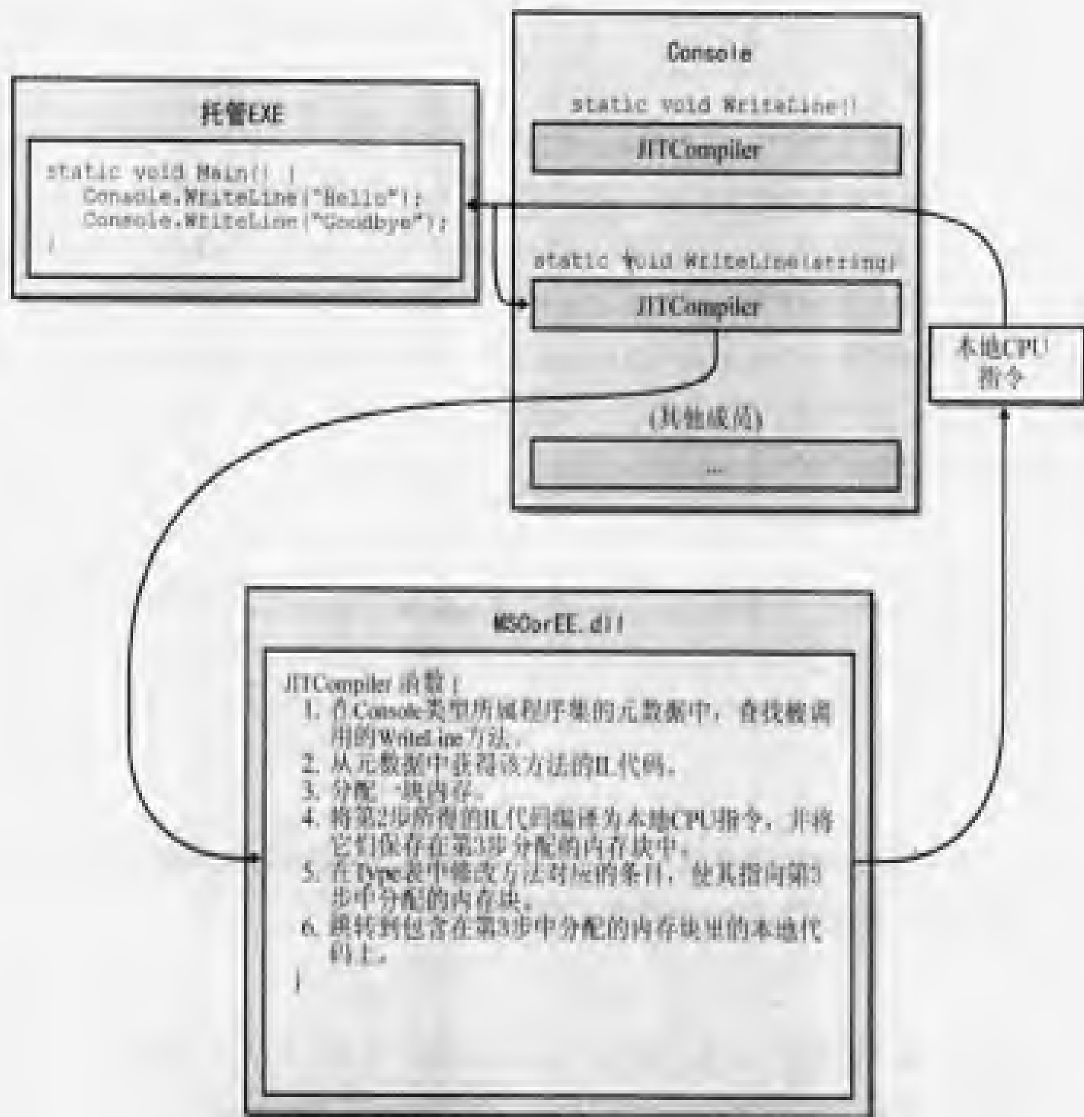


图 1.4 方法的第一次调用

在 Main 方法执行之前，CLR 首先会检测 Main 中代码引用到的所有类型。这会导致 CLR 分配一个内部的数据结构，该数据结构用于管理对所引用到的类型的访问。在图 1.4 中，Main 方法只引用了一个 Console 类型，CLR 将会为此分配一个单独的内部结构。在这个内部的数据结构中，Console 类型中定义的每一个方法都会有一个对应的条目。每个条目中将保存有一个方法实现代码的地址。当该

数据结构被初始化时，CLR 将把每一个条目设置为 CLR 内部的一个没有正式记录的函数，我们暂且称该函数为 JITCompiler。

当 Main 方法第一次调用 WriteLine 时，JITCompiler 函数将被调用，该函数负责将一个方法的 IL 代码编译成本地 CPU 指令。因为 IL 代码是被“即时(just in time)”编译的，所以 CLR 的这一部分通常被称作 JITter 或者**即时编译器**(JIT compiler)。

当 JITCompiler 函数被调用时，它会知道正在调用的是哪个方法，以及该方法是由哪个类型定义的。JITCompiler 函数随后会在被调用方法所定义的程序集中的元数据内搜索其 IL 代码的位置。JITCompiler 接着验证这些 IL 代码并将它们编译成本地 CPU 指令。这些本地 CPU 指令将被保存在一个动态分配的内存块中。然后 JITCompiler 将前面内部数据结构中被调用方法的地址替换为包含本地 CPU 指令的内存块地址。最后 JITCompiler 将跳转到该内存块中的代码上。这里的代码就是 WriteLine 方法(接受一个 String 类型参数的那个版本)的实现代码。当该代码执行完毕，它将返回到 Main 函数中，Main 函数会接着继续执行下面的代码。

现在，Main 函数开始第二次调用 WriteLine。由于 WriteLine 中的 IL 代码已经被验证并且编译过，所以这一次将直接调用内存块中已有的本地代码，完全跳过 JITCompiler 函数的验证和编译过程。在 WriteLine 方法执行之后，同样将返回到 Main 中。图 1.5 展示了第二次调用 WriteLine 方法时的情形。

这样，一个方法只有在被首次调用时才会产生一些性能损失。所有对该方法后续的调用都将以本地代码做全速执行，因为本地代码不再需要验证和编译。

JIT 编译器将本地代码存储于动态内存之中，这意味着当应用程序关闭时，编译生成的本地代码将被丢弃。这样，如果我们以后再次运行该应用程序，或者同时运行该应用程序的两个不同的实例(即位于两个不同的操作系统进程中)，JIT 编译器需要再次将同样的 IL 代码编译成本地指令。

对于大多数应用程序而言，JIT 编译过程引起的性能损失显得微不足道。而且大多数应用程序经常反复调用同一个方法。这样，在应用程序执行时，这些方法引起的也只是一次性能损失。而且通常方法内部执行所花费的时间要远比方法调用本身所花费的时间多得多。

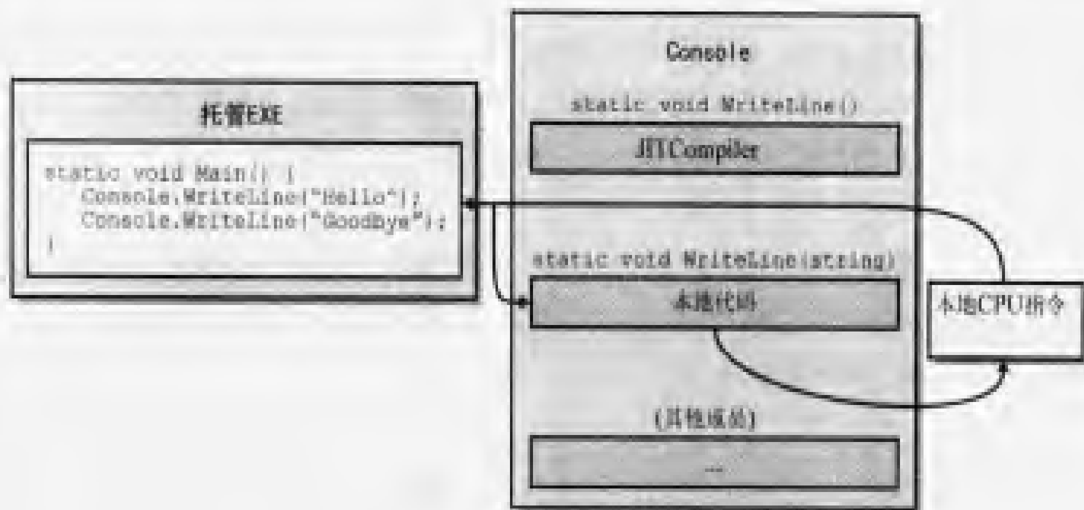


图 1.5 方法的第二次调用

CLR 的 JIT 编译器对本地代码的优化方式类似于非托管 C++编译器后端所做的工作。另外，产生优化代码可能花费更多的时间，但是经过优化后的代码的执行效率显然会更高。

那些有着传统非托管 C/C++经验的开发人员可能会对这里的一些性能差别有所顾虑。毕竟，非托管代码是针对某一特定 CPU 平台所编译的，当它们被调用时，这些代码便会立即执行。而在托管环境中，代码的编译要经过两个阶段才能完成。首先，编译器需要扫描源代码，将其编译为 IL 代码。但是要执行这些 IL 代码，它们还必须在运行时被编译成本地 CPU 指令，这项工作通常需要分配更多的内存，也要花费更多的时间。

由于我自己也来自于C/C++的背景，我对这些额外的开销也非常关心，也有过相当的怀疑。一个基本的事实是，出现在运行时的第二阶段编译不仅会损伤系统性能，还要分配额外的动态内存。但是，微软针对性能已经做了大量的工作，这些额外的开销已经被降到了最低程度。

如果大家也有这样的怀疑，那么就应该自己动手创建一些应用程序，并对它们做一些性能测试。另外，也应该运行一些微软和其他厂商生产的专业化的应用程序，并对它们做一些性能评测。我相信大家将会对得到的良好性能感到惊讶。

实际上，大家很难相信托管应用程序有可能在性能方面超过非托管应用程序，但很多人(包括我)都认为这的确有可能。有许多理由可以支持这一点。例如，当 JIT 编译器在运行时将 IL 代码编译成本地代码时，编译器对于执行环境的了解要多于非托管编译器。下面是托管代码有可能胜过非托管代码的一些地方：

- 如果应用程序在一些新型的 CPU(如奔腾 4 CPU)上运行，JIT 编译器将能够检测到这种情况，并产生利用这些新型 CPU 提供的特殊指令的本地代码。而非托管应用程序通常被编译为面向具有最小通用功能集合的 CPU 平台，一般会避免使用新型 CPU 提供的特殊指令。而这些特殊指令往往会在较新的 CPU 上为应用程序带来很高的性能提升。

- JIT 编译器能检测到正在运行的机器上某些总是返回错误的布尔测试。例如，考虑有如下代码的一个方法：

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

  如果宿主机器只有一个 CPU，那么对于该段代码，JIT 编译器将不会产生任何 CPU 指令。在这种情况下，针对宿主机器的本地代码就会得到更好的调整：代码量将变得更小，执行速度也会更快。

- 在应用程序运行时，CLR 能够分析评估代码的执行情况，并有选择地重新将 IL 代码编译为本地代码。根据观察到的执行模式，被编译的代码可以被重新组织以提高分支预测的成功率。

以上谈的仅仅是托管代码执行效率优于非托管代码的众多原因中的一部分。如前所述，目前大多数托管应用程序的性能已经非常不错了，并且随着时间的推移还有望得到更大的提高。

如果试验显示CLR的JIT编译器不能为应用程序提供期望的性能，我们可以利用NGen.exe工具，它是和.NET框架SDK一起发布的一个工具。该工具可以将一个程序集中所有的IL代码转化为本地代码，并将结果代码保存在磁盘上的一个文件中。在运行时程序集被加载的时候，CLR将自动检查是否有该程序集的预编译版本存在，如果存在，CLR将加载预编译的代码，不再需要额外的运行时编译。注意NGen.exe对于实际执行环境的假设肯定比较保守，因此NGen.exe产生的代码将不如JIT编译器产生的代码那样高度优化。

1.4.1 IL与代码验证

IL是一种基于堆栈的语言，这意味着它的所有指令或者是将操作数推进一个执行堆栈中，或者是从堆栈中弹出结果。因为IL没有提供操作寄存器的指令，所以编译器开发人员可以很容易地产生出IL代码来。他们不必再考虑管理寄存器，并且需要的IL指令也比较少(因为不再有操作寄存器的指令)。

IL指令是无类型的。例如，IL提供了一个add的指令，该指令对推进堆栈中的最后两个操作数做相加操作，没有32位和64位指令的区分。当add指令执行时，它首先判定堆栈上操作数的类型，然后进行适当的操作。

我个人认为，IL的最大好处不是对底层CPU的抽象，而是大大提高了应用程序的健壮性(robustness)。当IL代码被编译为本地CPU指令时，CLR将执行一个称作验证(verification)的过程。验证过程检查高级IL代码，确保它做的每件事情都是“安全”的。下面是验证过程可以检验的一些情形：不能从未初始化的内存中读取数据；每个方法调用都必须传入正确的参数个数，并且各个参数的类型要正确匹配；每个方法的返回值都必须被正确地使用；每个方法都必须有一个返回语句，等等。

托管模块的元数据中包括了所有被验证过程使用的方法和类型信息。如果IL代码被确定是“不安全”的，那么将有一个System.Security.VerificationException异常被抛出，阻止方法继续执行。

我们的代码是否安全？

默认情况下，微软的 C#和 Visual Basic 编译器产生的都是安全代码(safe code)。安全代码是那些可验证安全的代码。但是，通过在 C#或其他一些语言(例如托管扩展 C++和 IL 汇编语言)中使用 unsafe 关键字，我们可以产生出不能验证安全的代码来。这些代码有可能，实际上也应该，是安全的，但是 CLR 并不能通过检查 IL 来证明这一点。

为了确保所有托管模块中的方法包含的都是可验证安全的 IL 代码，我们可以使用和.NET 框架 SDK 一起发布的 PEVerify 工具(PEVerify.exe)。当微软测试他们的 C#和 Visual Basic 编译器时，他们就是用 PEVerify 运行编译得到的模块来确保编译器总是产生可验证安全的代码。如果 PEVerify 检测到非安全代码，微软会修复相应的编译器。

我们可能会考虑在打包和发布自己的模块前对它们运行 PEVerify。如果 PEVerify 检测到问题，那么将表明编译器内有 bug，这时我们应该向微软(或者生产我们所使用的编译器的其他厂商)报告该 bug。如果 PEVerify 没有检测到不可验证的代码，我们就可以确保这些代码在终端用户机器上运行时不会抛出 VerificationException 异常。

需要注意的是验证过程需要访问包含在任何与被验证代码相关的程序集中的元数据。所以当我们使用 PEVerify 来检查一个程序集时，它必须能够定位并加载所有被引用到的程序集。因为 PEVerify 使用 CLR 来定位相关的程序集，所以定位这些程序集所用到的绑定(bind)与探测(probe)规则和执行程序集时的规则相同。(本书第 2 章和第 3 章将讨论绑定与探测规则。)

注意管理员可能会选择关闭验证过程(使用 Microsoft .NET Framework Configuration 管理工具)。如果验证过程被关闭，JIT 编译器将会把不可验证的 IL 代码编译为本地 CPU 指令。这时，管理员对代码的行为负全部责任。

在 Windows 操作系统中，每一个进程都有自己的虚拟地址空间。地址空间的分离是有必要的，因为我们不能信任应用程序的代码。一个应用程序完全有可能(并且不幸的是，也很常见)读写一个无效的内存地址。将每个 Windows 进程放在独立的地址空间提高了应用程序的健壮性，因为这样一个进程就不会干扰另一个进程的运行。

然而，通过验证托管代码，我们可以确保它们不会访问它们不应该访问的内存，因此也就不会干扰另一个应用程序的代码。这意味着我们可以在一个单独的 Windows 虚拟地址空间内运行多个托管应用程序。

因为 Windows 进程需要许多操作系统资源，太多的进程会损伤性能，并限制系统中可用的资源。在一个操作系统进程中运行多个托管应用程序可以减少进程的数量，从而提高系统性能，降低资源需求，而应用程序仍然可以保持良好的健壮性。这是托管代码相较于非托管代码的另一个优点。

实际上，CLR 确实提供了在一个单独的操作系统进程中执行多个托管应用程序的能力。在 CLR 中，一个托管应用程序称作一个**应用程序域**(AppDomain)。默认情况下，一个托管 EXE 仅执行在它自己单独拥有的地址空间中(该地址空间中仅含有一个应用程序域)。但是，CLR 的宿主进程(例如 IIS 服务器和下一个版本的 SQL Server)可以决定在一个操作系统进程中运行多个应用程序域。本书第 20 章将对应用程序域做详细的探讨。

1.5 .NET 框架类库

在.NET 框架中包括有一组.NET 框架类库(Framework Class Library，简称 FCL)程序集，其中含有几千个类型的定义，每一个类型都提供了某种功能。总的来说，CLR 和 FCL 允许开发人员创建以下几种应用程序：

- **XML Web 服务** 即 XML Web Services，又简称为 Web 服务(Web Services)。该服务使得我们可以非常容易地通过互联网来进行方法调用。XML Web 服务是 Microsoft .NET 整个平台创新中最重要的一环。
- **Web 窗体** 即 Web Forms，一种基于 HTML 的应用程序(Web 站点)。Web 窗体应用程序通常会做一些数据库查询和 Web 服务调用，然后对返回的信息进行组合和筛选，最后通过一个基于 HTML 的用户界面将信息表示在浏览器中。Web 窗体为用任何 CLR 语言编写 Web 应用提供了一个类似于 Visual Basic 6 和 Visual InterDev 风格的开发环境。
- **Windows 窗体** 即 Windows Forms，Windows 图形用户界面(GUI)应用程序。和使用 Web 窗体创建应用程序界面相反，Windows 窗体可以使我们利用 Windows 桌面提供的更强大，更高效的功能。Windows 窗体应用程序可以利用控件、菜单、鼠标和键盘事件，并且直接和底层操作系统通信。和 Web 窗体相似，Windows 窗体应用程序也可以做数据库查询，调用 XML Web 服务。Windows 窗体为使用任何一种 CLR 语言编写图形用户界面应用程序提供了一种类似于 Visual Basic 6 风格的开发环境。

- **Windows 控制台应用程序** 对于较简单的用户界面需求，控制台应用程序为我们提供了一个快速、轻便的创建方式。各种编译器、实用程序、工具通常被实现为控制台应用程序。

- **Windows 服务** 利用.NET 框架，我们可以创建出由 Windows 服务控制管理器(Service Control Manager，简称 SCM)控制的服务程序。

- **组件库** .NET 框架允许我们创建单独的组件(类型)，它们可以应用于前面提到的各种类型的应用程序。

FCL 中包含了数以千计的类型，相关的类型放在一个由命名空间(namespace)组织的集合中提供给开发人员。例如，System 命名空间(大家应该对此比较熟悉)中就包含了 Object 基类型，所有其他的类型都直接或者间接由此继承而得。另外 System 命名空间还包括了整数、字符、字符串、异常处理、控制台输入/输出(Input/Output，简称 I/O)，以及许多实用类型，它们可以用来安全地转换数据类型、格式化数据类型、产生随机数以及执行各种数学运算。所有的应用程序都会用到 System 命名空间中的类型。

为了获取.NET 平台的各种特性，大家应该知道自己需要的类型包含在哪个命名空间中。如果希望能够定制某些类型的行为，大家可以从期望的 FCL 类型中继承得到自己的类型。面向对象是.NET 框架为软件开发人员提供的一个一致的编程范式。另外，大家也可以很方便地创建一些命名空间来包含自己的类型。这些命名空间和类型可以与面向对象的编程范式无缝地结合在一起。相较于传统 Win32 编程范式，这种新的方法大大简化了现代软件开发。

FCL 中的大多数命名空间提供的类型都可用于各种应用程序。表 1.2 列出了一些较为通用的命名空间，并且简要地描述了其内类型的用途。

表 1.2 一些通用的 FCL 命名空间

命名空间	描 述
System	其中的类型是为所有应用程序使用的一些基本类型
System.Collections	其中的类型用于管理对象集合。包括常用的集合类型，例如堆栈、队列、散列表等

续表

命名空间	描 述
System.Diagnostics	其中的类型用于帮助诊断和调试应用程序
System.Drawing	其中的类型用于操作二维图形。它们典型地用于 Windows 窗体应用程序，以及创建 Web 窗体页面中显示的图象
System.EnterpriseServices	其中的类型用于管理事务、队列组件、对象池、JIT 激活(译注：这里的 JIT 不同于.NET 框架中所言的 JIT 编译，它特指 COM+组件服务，即.NET 内的企业服务中对象的即时激活技术)、安全以及其他一些提高服务器程序中托管代码效能的特性
System.Globalization	其中的类型用于多国语言支持(National Language Support，简称 NLS)，例如字符串比较、格式化以及日历功能
System.IO	其中的类型用于操作 I/O 流、遍历目录和文件
System.Management	其中的类型通过 Windows 管理设备(Windows Management Instrumentation，简称 WMI)来管理企业中的计算机
System.Net	其中的类型用于网络通信
System.Reflection	其中的类型用于查看元数据以及延迟绑定类型和它们的成员
System.Resources	其中的类型用于操作外部数据资源
System.Runtime.InteropServices	其中的类型允许托管代码访问非托管操作系统平台中的一些功能，如 COM 组件和 Win32 DLL 内的函数
System.Runtime.Remoting	其中的类型用于从远程机器上访问类型
System.Runtime.Serialization	其中的类型用于持久化(persist) 对象实例，以及从一个流(stream)中重新产生对象实例
System.Security	其中的类型用于保护数据和资源
System.Text	其中的类型用于以不同的编码方式(如 ASCII 或者 Unicode)来操作文本
System.Threading	其中的类型用于异步操作，以及同步访问资源
System.Xml	其中的类型用于处理 XML 模式(schema)和数据

本书主要讨论 CLR 以及和 CLR 紧密相关的一些通用类型(位于表 1.2 中列出的大多数命名空间中)。所以无论创建何种应用程序，本书的内容对所有.NET 框架的开发人员都是适用的。

除了一些较为常用的命名空间外，FCL 还提供了一些命名空间，其中的类型用于创建某些特定的应用程序。表 1.3 列出了 FCL 中一些和特定应用程序相关的命名空间。

表 1.3 一些用于特定应用程序的 FCL 命名空间

命名空间	描 述
System.Web.Services	其中中的类型用于创建 XML Web 服务
System.Web.UI	其中的类型用于创建 Web 窗体
System.Windows.Forms	其中的类型用于创建 Windows GUI 应用程序
System.ServiceProcess	其中的类型用于创建由 SCM 控制的 Windows 服务

相信有很多讲解这些特定应用程序(如 Windows 服务、Web 窗体和 Windows 窗体)的好书会陆续出版。它们会为大家提供一个创建应用程序的良好开端。这些着眼于特定应用程序的书籍会帮助大家从高层的角度来学习.NET 应用程序开发，因为它们关注的是应用程序的种类，而非底层开发平台。本书的目标是帮助大家从底层的角度来学习.NET 开发平台。读完本书后，大家可再读一些与特定应用程序相关的书，便可以轻松熟练地创建各种.NET 框架应用程序了。

1.6 通用类型系统

到目前为止，大家应该很清楚 CLR 的所有内容都是围绕着类型展开的。类型为应用程序和组件提供了它们所需的功能。类型也作为一种机制使得一种语言编写的代码可以和另一种语言编写的代码进行无缝地集成。由于类型是 CLR 的基础，微软为此专门制定了一个正式的规范——通用类型系统(Common Type System，简称 CTS)来描述类型的定义和行为。

CTS 规范规定一个类型可以包含 0 个或多个成员。本书的第III部分将对所有这些成员做详细的介绍。目前大家只需要对它们有一个简单的了解即可。

- **字段**　字段是一个属于对象状态部分的数据成员。字段由它们的名称和类型标识。
- **方法**　方法是一个在对象上执行某种操作的函数，通常会改变对象的状态。方法有一个名称、签名和修饰符。方法的签名指定了方法的调用约定、参数个数(以及它们的顺序)、参数类型以及返回值类型。
- **属性**　对于属性的调用者，属性看起来非常类似于字段。但是对于属性的实现者，属性看起来更象一个(或者两个)方法。属性允许实现者在访问数值之前验证输入参数和对象状态的有效性，或者仅在需要的情况下进行求值运算。属性使得类型的用户可以使用简化的语法来表达他们的意图。可以创建只读、只写和可读可写共三种属性。
- **事件**　事件允许在一个对象和其他相关联的对象之间建立一个通知机制。例如，一个按钮可以提供一个事件，当该按钮被按下时，其他对象将得到通知。

CTS 还定义了类型可见性和访问类型成员的一些规则。例如，将一个类型修饰为 public(在 C#中使用 public)，将使得它对于任何程序集都是可见的。另一方面，将一个类型修饰为 assembly(在 C#中使用 internal)，将使得它仅对于其所定义的程序集中的代码可见。CTS 建立了以程序集作为类型可见性边界的规则，而 CLR 实现了这种可见性规则。

无论一个类型对于调用者是否可见，我们都需要指出调用者对于类型的成员有怎样的访问控制。下面的列表展示了几种可用于方法或字段的访问控制选项：

- **Private**　方法只能被同一类型中的其他方法调用。
- **Family**　方法可以被派生类型中的代码调用，而不管它们是否位于同一个程序集中。注意许多编程语言(例如 C++和 C#)都将之称作 protected。
- **Family 与 assembly**　方法只可以被位于同一个程序集中的派生类型中的代码调用。许多编程语言(例如 C#和 Visual Basic)都没有提供这种访问控制。当然，IL 汇编语言可以做到这一点。
- **Assembly**　方法可以被同一个程序集中的任何代码调用。许多编程语言将之称为 internal。

- **Family 或 assembly** 方法可以被任何程序集中的派生类型的代码调用，也可以被同一程序集中的任何类型调用。C#将之称为 protected internal。

- **Public** 方法可以被任何程序集中的任何代码调用。

另外，CTS 还定义了诸多规则来管理类型继承、虚函数、对象生存期等事项。设计这些规则的目的是使它们的语意可以用现代编程语言方便地表达出来。实际上，我们甚至不需要学习 CTS 规则，因为我们选择的语言已经提供了我们所熟悉的语言语法和类型规则，并且在生成托管模块时会将这些特定语言的语法映射为 CLR“语言”。

当我第一次使用 CLR 时，我很快就意识到最好将语言和代码行为视作两个分离的、截然不同的事物。我们可以使用 C++来定义自己的类型和成员，我们也可以使用 C#或者 Visual Basic 来定义同样的类型和成员。虽然定义类型所使用的语法因所选语言的不同而各不相同，但是类型的行为绝对是完全相同、与语言无关的，因为 CLR 中的 CTS 定义了类型的行为。

为了帮助澄清上面的概念，这里给大家一个例子。我们知道，CTS 仅支持单继承。虽然 C++语言支持多继承，但 CTS 却不能接受并操作任何这样的类型。为了帮助开发人员，微软的 Visual C++编译器一旦发现在托管代码中有类型继承自多个基类，它将报告错误。

下面是另一个 CTS 规则：所有的类型都必须(直接或间接)继承自预定义类型 System.Object。很明显，Object 是定义在 System 命名空间中的一个类型。Object 作为所有其他类型的根类保证了每个类型实例都有一组公共行为。具体而言，System.Object 类型允许我们执行以下操作：

- 判断两个实例是否相等
- 获得实例的散列码
- 查询实例的类型
- 执行实例的浅(即按位，bitwise)拷贝(shallow copy)
- 获得实例当前状态的字符串表示

1.7 通用语言规范

COM允许不同语言创建的对象能够进行相互之间的通信。CLR则集成了所有的编程语言，并且允许一种语言创建的对象在另一种不同语言编写的代码中被看作同等的成员。CLR的标准类型集合，自描述类型信息(即元数据)，和通用执行环境使得这种集成成为可能。

语言集成实在是一个近乎幻想的目标，因为各种编程语言之间有着很大的区别。例如，某些语言不区分符号的大小写，或者不提供无符号整数，或者不支持操作符重载，或者方法不支持可变数目的参数。

如果希望创建的类型可以被其他编程语言方便地访问，只能使用编程语言中那些对其他语言来说也可用的特性。为了解决这一问题，微软定义了一个**通用语言规范**(Common Language Specification，简称CLS)，该规范为编译器厂商详细描述了面向CLR的编译器必须支持的一个最小特性集合。

CLR/CTS支持的特性要比CLS定义的子集丰富得多。如果不关心语言之间的互操作，我们则可以开发出非常丰富的类型，这些类型仅受限于我们所选择语言的特性集合。具体而言，CLS定义了可以被任何与CLS兼容的编程语言访问到的，外部可见的类型和方法所必须遵循的规则。注意CLS规则不适用于仅在所定义程序集中可访问的代码。图1.6概括了本段表达的含义。

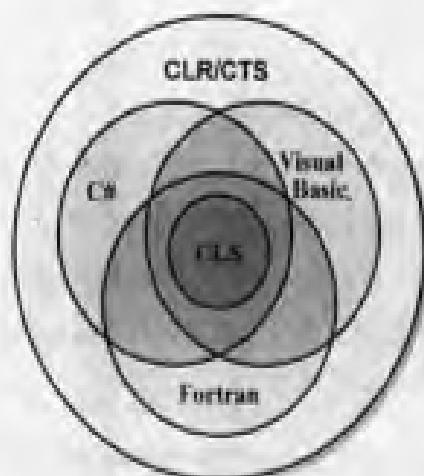


图1.6 各种语言提供了一个CLR/CTS的子集和一个CLS的超集(不必为同一超集)

如图1.6所示，CLR/CTS提供了一组特性。一些语言会提供这些特性中的一个较大的子集。例如，使用IL汇编语言的开发人员将能够利用CLR/CTS提供的所有特性。大多数其他语言，例如C#、Visual Basic和Fortran，提供给开发人员的将是CLR/CTS特性的一个子集。而CLS定义了所有

语言必须支持的一个最小特性集合。

如果打算采用某种语言设计一个类型，并且希望该类型能被其他语言使用，那么就不应该利用任何 CLS 之外的特性。那样做将意味着使用其他编程语言的开发人员可能访问不到该类型的成员。

下面的代码使用 C#来定义一个与 CLS 兼容的类型，其中一些与 CLS 不兼容的构造会引起 C#编译器报错。

```
using System;

// 告知编译器检查 CLS 兼容性
[assembly:CLSCompliant(true)]

// 因为该类是公有的，有关 CLS 不兼容的错误将会被显示
public class App {

    // 错误：App.Abc()的返回值类型与 CLS 不兼容
    public UInt32 Abc() { return 0; }

    // 错误：仅有大小写差别的标识符 App.abc()与 CLS 不兼容
    public void abc() { }

    // 不会报错，因为该方法是私有的
    private UInt32 ABC() { return 0; }
}
```

在上面的代码中，我们在当前的程序集上应用了一个[assembly:CLSCompliant(true)]特性(attribute)。该特性告知编译器必须确保在所有的公有导出类型中，不能有任何阻止其他编程语言访问类型的构造。当上面的代码被编译时，C#编译器将产生两个错误。第一个错误是由于方法 Abc 返回了一个无符号整数；而 Visual Basic 和其他一些语言不能操作无符号整数值。第二个错误是由于该类型提供了两个仅存在名字大小写和返回值类型差别的公有方法：Abc 和 abc。而 Visual Basic 和其他一些语言不能调用这两个方法。

有趣的是，如果将 class App 前面的 public 删除并重新编译，前面的两个错误都将消失。这是因为 App 类型将默认为 internal，因此不再用于程序集之外。关于 CLS 规则的完整列表，可参见.NET 框架 SDK 文档中的“跨语言互操作”部分。

下面是对 CLS 规则的一个更加简化且易于理解的表述。在 CLR 中，类型的每一个成员或者是一个字段(数据)，或者是一个方法(行为)。这意味着每一种编程语言都必须能够访问字段和调用方法。这些字段和方法以或者普通或者特殊的应用方式被展现出来。为了简化程序设计，一些编程语言通常都会提供某些额外的抽象使得这些通用的编程模式编写起来更加容易。例如，一些语言提供了枚举、

数组、属性、索引器、委托、事件、构造器、析构器、重载操作符、转换操作符等诸如此类的概念。当编译器在源代码中遇到这些构造时，必须将它们翻译成字段或方法，这样 CLR 和其他的编程语言才能够访问它们。

我们来看下面的类型定义，其中包含了构造器、析构器、重载操作符、属性、索引器和事件。注意这里显示的代码仅仅是为了能够通过编译，并不是实现一个类型的正确方法。

```
using System;

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

    // 析构器
    ~Test() {}

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

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

    // 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;
}
```

这些代码经过编译后，得到的结果将是一个定义了许多字段和方法的类型。我们可以利用.NET框架 SDK 中提供的 IL 反汇编器工具(ILDasm.exe)来查看得到的托管模块，如图 1.7 所示。

图 1.7　用 ILDasm 显示 Test 类型的字段和方法(由元数据获得)

表 1.4 向大家展示了上面例子中编程语言的各种构造与等效的 CLR 字段和方法之间的映射关系。

表 1.4　Test 类型的字段和方法(由元数据获得)

类型成员	成员的种类	等效的编程语言构造
AnEvent	字段	事件；该字段名称为 AnEvent，类型为 System.EventHandler
.ctor	方法	构造器
Finalize	方法	析构器
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		索引器的 set 访问器方法

Test 类型下面的另外一些节点没有在表 1.4 中提及，即.class、.custom、AnEvent、AProperty 和 Item，它们标识了类型中一些额外的元数据。这些节点没有映射到字段或者方法上，仅仅是提供了一些关于类型的附加信息，它们可以被 CLR、编程语言以及一些工具访问到。例如，利用某种工具我们可以发现 Test 类型提供了一个名为 AnEvent 的事件，该事件通过两个方法(add_AnEvent 和 remove_AnEvent)提供给外界。

1.8 与非托管代码互操作

.NET 框架提供了许多较其他开发平台更为优越的特性。然而，很少有公司能够负担得起重新设计并实现所有现存代码。微软意识到了这一点，并为此对 CLR 做了特殊的设计，从使其允许应用程序同时包含托管和非托管部分。具体来说，CLR 支持三种互操作情形：

- **托管代码调用 DLL 中的非托管函数** 托管代码可以很容易地使用一种称作 P/Invoke(即 Platform Invoke，平台调用)的机制来调用 DLL(译注：这里的 DLL 指的是传统动态链接库文件)中的函数。实际上，许多 FCL 中定义的类型内部都调用了从 Kernel32.dll，User32.dll 等动态链接库中导出的函数。许多编程语言都提供了这样的机制，使得从托管代码中调用 DLL 中的非托管函数变得非常容易。例如，C#或者 Visual Basic 应用程序就可以调用从 Kernel32.dll 中导出的 CreateSemaphore 函数。

- **托管代码使用现存的 COM 组件(非托管组件作为 COM 服务器)** 许多公司已经实现了大量的 COM 组件。利用这些 COM 组件的类型库，我们可以创建出描述它们的托管程序集。托管代码可以象访问其他托管代码一样访问这些托管程序集中的类型。更多信息请参见与.NET 框架 SDK 一起发布的 TlbImp.exe 工具。有时候，我们可能没有类型库，或者希望能对 TlbImp.exe 产生的内容有更多的控制。这时，我们可以在源代码中手动创建自己的类型，CLR 可以用这些类型来进行正确的互操作。例如，我们可以在 C#或者 Visual Basic 应用程序中使用 DirectX COM 组件。

- **非托管代码使用托管类型(托管类型作为 COM 服务器)** 许多现存的非托管代码都需要提供 COM 组件才能正常运行。但是用托管代码来实现这些组件会更容易，因为这样可以避免处理引用计数和 COM 接口。例如我们可以用 C#或者 Visual Basic 来创建 ActiveX 控件或者 shell 扩展程序。更多信息请参见与.NET 框架 SDK 一起发布的 TlbExp.exe 和 RegAsm 工具。

除了以上三种情形，微软的 Visual C++编译器(版本 13)支持一种新的/clr 命令行开关。该开关告知编译器产生 IL 代码，而不是本地 x86 指令。如果我们有许多现存的 C++代码，则可以使用这个新的命令行开关重新编译它们。这些新的代码需要 CLR 才能执行，而且我们可以不断地修正代码以利用 CLR 所独有的一些特性。

但是对于下面的方法，命令行开关/clr 目前还不能将它们编译成 IL 代码：包含内联汇编语言(通过__asm 关键字实现)的方法；接受可变数目参数的方法；调用 setjmp 的方法；包含一些内部程序(intrinsic routine)(例如__enable，__disable，__ReturnAddress，和__AddressOfReturnAddress)的方法。关于不能被编译为 IL 代码的情形的完整列表，请参阅 Visual C++编译器文档。当编译器不能将某些方法编译为 IL 代码时，它可以将它们编译为 x86 代码，这样应用程序仍然可以运行。

注意虽然编译产生的 IL 代码是托管的，但数据却不是。也就是说数据对象并不是在托管堆上分配的，并且它们也不会被垃圾收集器回收。实际上编译器并没有为这些数据类型产生相应的元数据，并且这些类型的方法名仍然需要经过 C++签名编码转换(mangle)。

下面的 C 语言代码调用了标准 C 运行时库中的 printf 函数，以及 System.Console.WriteLine 方法。其中 System.Console 类型定义于 FCL 中。可以看到，托管类型能够象通常的 C/C++程序库一样被 C/C++代码使用。

```
#include <stdio.h>          // 使用 printf

#using <mscorlib.dll>       // 使用定义在该程序集中的托管类型

using namespace System;     // 允许更便捷地使用 System 命名空间中的类型

// 实现通常的 C/C++ main 函数
void main() {

    // 调用 C 运行时库中的 printf 函数
    printf("Displayed by printf.\r\n");

    //调用 FCL 内 System.Console 类型的 WriteLine 方法
    Console::WriteLine("Displayed by Console::WriteLine.");
}
```

编译上面的代码非常容易。假设该代码位于 MgdCApp.cpp 文件中，我们可以在命令行提示符中执行下面的命令进行编译：

```
cl /clr MgdCApp.cpp
```

编译得到的结果将是一个 MgdCApp.exe 程序集文件。运行 MgdCApp.exe 将会看到以下输出：

```
C:\>MgdCApp
Displayed by printf.
Displayed by Console::WriteLine.
```

如果用 ILDasm.exe 查看该文件，将会看到如图 1.8 所示的输出。

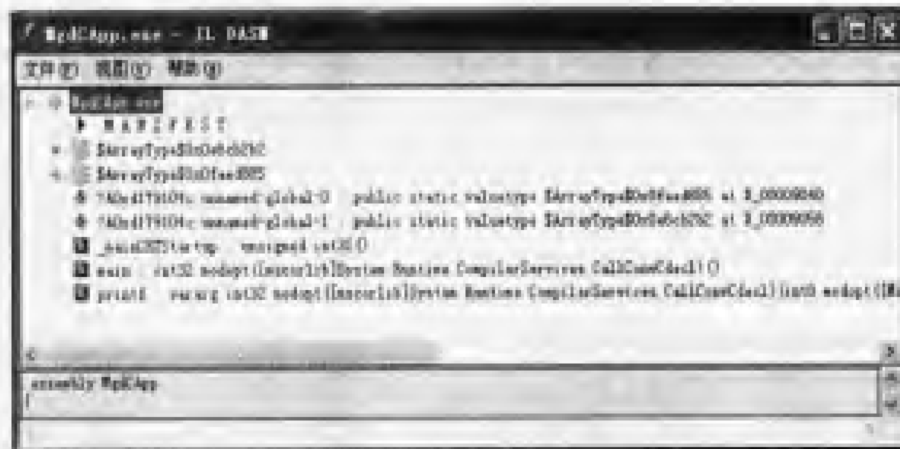


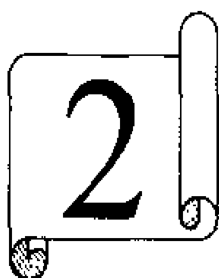
图 1.8 用 ILDasm 显示 MgdCApp.exe 程序集中的元数据

在图 1.8 中，大家可以看到 ILDasm 显示了所有定义在该程序集中的全局函数和全局字段。很明显，编译器自动产生了许多内容。如果双击 Main 方法，ILDasm 将显示以下 IL 代码：

```
.method public static int32
        modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
        main() cil managed
{
   .vtentry 1 : 1
   // Code size    28 (0x1c)
   .maxstack 1
   IL_0000:  ldsflda     valuetype
                         $ArrayType$0x0faed885 '?A0x44d29f64.unnamed-global-0'
```

```
    IL_0005:  call         vararg int32
       modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
       printf(int8
       modopt([Microsoft.VisualC]Microsoft.VisualC.NoSignSpecifiedModifier)
       modopt([Microsoft.VisualC]Microsoft.VisualC.IsConstModifier)*)
    IL_000a:  pop
    IL_000b:  ldsflda      valuetype
                           $ArrayType$0x0e6cb2b2 '?A0x44d29f64.unnamed-global-1'
    IL_0010:  newobj       instance void [mscorlib]System.String::.ctor(int8*)
    IL_0015:  call         void [mscorlib]System.Console::WriteLine(string)
    IL_001a:  ldc.i4.0
    IL_001b:  ret
  } // end of method 'Global Functions'::main
```

这些代码看起来并不漂亮，因为为了使它们能够正常工作，编译器产生了许多特殊的代码。然而，从产生的 IL 代码中，我们仍然可以看到 printf 和 Console.WriteLine 都得到了调用。



生成、打包、部署及管理应用程序与类型

在学习开发 Microsoft .NET 框架应用程序之前，我们首先来探讨一下生成(build)、打包和部署应用程序及其类型的各个环节。本章主要讲述如何创建那些供应用程序单独使用的组件。接下来的第 3 章将会探讨一些更为高级的概念，包括怎样生成和使用一些被多个应用程序所共享的程序集。另外，这两章还会讨论作为一个管理员怎样利用某些信息来影响应用程序及其类型的执行。

当今的应用程序一般都包含着好几个类型。在.NET 框架中，类型(type)又被称作组件(component)，但本书将使用类型这个术语，而避免使用组件。通常情况下，应用程序既包括我们自己创建的类型，也包括微软和其他一些组织创建的类型。如果这些类型都采用支持 CLR 的语言开发，那么它们将可以无缝地在一起协作执行。我们甚至可以使用一种语言开发一个基类，然后再使用另一种语言开发它的子类。

在讲述生成、打包和部署类型之前，我们首先来对.NET 框架所解决的一些问题做一个简短的回顾。

2.1 .NET 框架部署目标

多年以来，Windows 一直背负着一个复杂和不稳定的坏名声。不管这种不良名声是否应得，它们的产生却是来源了许多不同的因素。首先，所有的应用程序都会使用到一些来自微软和其他厂商的动态链接库(DLL)。因为应用程序执行的是来自于不同厂商的代码，而任何一段代码的开发者都不可能确定其他人将怎样使用这些代码，这就会引起各种各样潜在的问题。但从实践来看，这类因交互引起的问题还不算严重，因为它们在部署之前往往会经过严格的评测与调试。

然而，当一个公司决定升级它的代码，并把新文件发布给用户时，往往就会出现问题。这些新文件理应和先前的文件保持向后兼容，但是谁又能保证呢？实际上当一个公司升级它的代码时，要重新评测和调试所有已经发布出去的应用程序通常是不可能的。

我相信我们很多人都遇到过类似这样的问题：当安装新的应用程序时，却发现它损坏了一个现有的应用程序。这种情况被称为“DLL hell”(DLL 地狱)。这种困境甚至给一些计算机用户带来了某种恐惧，结果是用户每次在他们的机器上安装新软件时都要做很慎重的考虑。就我个人而言，因为担心一些现存的应用程序受影响，我一般不会去尝试安装某些软件。

造成 Windows 不良名声的第 2 个原因是其复杂的安装过程。目前大多数应用程序的安装都会影响到系统的各个部分。例如，安装一个应用程序会导致文件被复制到许多目录中，注册表设置要被更新，快捷链接也要被安装到桌面、开始菜单和快速启动工具栏上。这样带来的问题是应用程序不能被隔离为一个单独的实体。备份一个应用程序就显得特别困难，因为必须复制应用程序所有的文件和相关的注册表内容。另外，将应用程序从一个机器移到另一个机器上也很困难。必须重新运行安装程序才能使所有的文件和注册表条目都得到正确的配置。最后，卸载或者删除应用程序也很麻烦，因为总免不了担心应用程序的某些部分仍然遗留在机器中。

第 3 个原因与安全有关。当应用程序被安装到用户的机器中时，它们往往会带来各种各样的文件，其中许多都是来自不同公司之手。另外，“Web 应用程序”通常还会通过网络下载代码，而用户甚至根本意识不到一些代码正在被安装到他们的机器中。这些被安装到用户计算机中的代码可以执行任何操作，包括删除文件、发送电子邮件。由于它们可能引起潜在的危害，用户自然会对安装新的应用程

序感到担心。为了使用户彻底放心，系统必须建立一种安全机制，来使得用户显式地允许或者禁止各个公司开发的代码去访问系统资源。

从本章和第3章中我们将会看到，.NET框架在很大程度上解决了“DLL hell”问题。在解决应用程序状态遍布用户硬盘的问题方面，它也有长足的进步。例如，组件不再需要像COM那样在注册表中进行注册。但还有一点比较遗憾，那就是目前的应用程序仍然需要快捷链接，但以后的Windows可能会解决这一问题。至于安全问题，.NET框架包含了一个称作**代码访问安全**(code access security)的新型安全模型。我们知道，Windows安全基于用户的身份，而代码访问安全则基于程序集的标识。利用代码访问安全模型，我们可以自己决定程序集的安全许可，比如信任微软发布的所有程序集，或者不信任任何从互联网上下载的程序集。相对于Windows操作系统而言，.NET框架使得我们对自己机器中的安装内容和运行内容有了更多的控制权。

2.2 将类型生成为模块

本节向大家展示怎样将包含各种类型的源代码文件变成一个可部署的文件。先来看一个简单的应用程序：

```
public class App {
    static public void Main(System.String[] args) {
        System.Console.WriteLine("Hi");
    }
}
```

上面的应用程序定义了一个名为App的类型。该类型有一个静态公有方法Main。Main里面又引用了另外一个类型System.Console。System.Console是微软实现的一个类型，该类型所有方法的IL代码都位于MSCorLib.dll文件中。可以看到，这个应用程序既定义了自己的类型，也使用了其他公司的类型。

要生成上面的示例应用程序，首先将前面的代码放在一个源代码文件中，比如App.cs，然后执行下面的命令：

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

该命令告诉C#编译器产生一个可执行文件App.exe(/out:App.exe)，且产生的文件类型为Win32控制台应用程序(/t[arget]:exe)。

当C#编译器处理上面的源文件时，它会发现代码中引用了System.Console类型的WriteLine方法。

这时，编译器需要确定 System.Console 类型存在，且它有一个 WriteLine 方法，同时还要确定该方法期望参数的类型和代码中提供的相匹配。为了使 C#编译器顺利编译这段代码，我们需要给它指定一组可以用来辨析外部类型引用的程序集。(后面会对程序集下定义，目前可以暂时把程序集看作是一组 DLL 文件。)在上面的命令行中，/r[eference]:MSCorLib.dll 命令行开关告诉编译器到 MSCorLib.dll 文件标识的程序集中查找外部类型。

MSCorLib.dll 是一个特殊的文件，它包含了.NET 框架中所有的核心类型，例如字节、整数、字符、字符串、等等。由于这些类型的使用非常频繁，C#编译器会自动引用该程序集。换句话说，下面的命令(/r 命令行开关已经被去掉)和前面的命令是等效的：

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

另外，因为/out:App.exe 和/t:exe 命令行开关也正好是 C#编译器的默认设置，所以下面的命令也和前面的命令等效：

```
csc.exe App.cs
```

如果由于某种原因，我们不想让 C#编译器引用 MSCorLib.dll 程序集，我们可以使用/nostdlib 命令行开关。例如，下面的命令在编译 App.cs 文件时将会出现错误，这是因为 System.Console 类型定义在 MSCorLib.dll 文件中。

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

现在让我们来更进一步看一看 C#编译器输出的文件 App.exe。这个文件究竟是什么呢？对于初学者而言，它是一个标准 PE 文件。这意味着一台运行 32 位或者 64 位 Windows 的机器应该能够加载并使用该文件。Windows 支持两种类型的应用程序，即控制台用户界面(console user interface，简称 CUI)应用程序和图形用户界面(graphical user interface，简称 GUI)应用程序。因为前面我们指定的是/t:exe 命令行开关，所以 C#编译器产生的将是一个 CUI 应用程序。我们也可以用/t:winexe 命令行开关来使 C#编译器产生一个 GUI 应用程序。

现在我们已经知道了我们创建的是哪一种 PE 文件。但是在 App.exe 文件中究竟是什么呢？一个托管 PE 文件包含 4 部分：PE 表头、CLR 表头、元数据和 IL 代码。PE 表头是 Windows 操作系统要求的标准信息。CLR 表头专门用于那些需要 CLR 才能运行的模块(托管模块)。CLR 表头中包含和模块一起创建的元数据的主版本号和次版本号，一些标记，如果模块是 CUI 或者 GUI 可执行文件还有一个标识入口点方法的 MethodDef 标记(后面会有讲述)，以及一个可选的强命名数字签名(将在第 3 章讨论)。最后该表头中还包括模块内某些元数据表的大小和偏移量。大家可以通过查看 CorHdr.h 头文件中定义的 IMAGE_COR20_HEADER 来得到 CLR 表头的准确格式。

元数据实际上就是一块二进制数据，其中包含着一些表。我们可以将元数据表划分为 3 大类：定义表、引用表和清单表。表 2.1 描述了一个托管模块的元数据块中包含的一些比较常用的定义表。

表 2.1 常用的元数据定义表

元数据定义表名称	描 述
ModuleDef	ModuleDef 表中总是包含一个标识托管模块的条目。该条目包括模块的文件名和扩展名(不含路径)，以及一个模块版本 ID(由编译器创建的 GUID 形式)。这使得文件在重命名时，可以保留它原来的名称记录。然而，强烈建议大家不要重命名 PE 文件，因为这将阻止 CLR 在运行时定位程序集
TypeDef	托管模块中定义的每一个类型在 TypeDef 表中都有一个对应的条目。每个条目包括类型的名称及其基类型，一些标记(如 public、private 等)，以及一些指针(这些指针指向 MethodDef 表中属于该类型的方法、FieldDef 表中属于该类型的字段、PropertyDef 表中属于该类型的属性以及 EventDef 表中属于该类型的事件)
MethodDef	托管模块中定义的每一个方法在 MethodDef 表中都有一个对应的条目。每一个条目包括方法名称，一些标记(如 private、public、virtual、abstract、static、final 等)，方法签名，以及该方法的 IL 代码在模块中的偏移量。另外，每一个条目还包含一个指向 ParamDef 表中对应条目(其中包含着方法的参数信息)的指针
FieldDef	托管模块中定义的每一个字段在 FieldDef 表中都有一个对应的条目。每一个条目包括一个字段名称，一些标记(如 private、public 等)，以及字段类型
ParamDef	托管模块中定义的每一个参数在 ParamDef 表中都有一个对应的条目。每一个条目包括一个参数名称和一些标记(如 in、out、retval 等)
PropertyDef	托管模块中定义的每一个属性在 PropertyDef 表中都有一个对应的条目。每一个条目包括属性名称，一些标记，属性类型，以及属性对应的后端字段(可能为 null)
EventDef	托管模块中定义的每一个事件在 EventDef 表中都有一个对应的条目。每一个条目包括一个事件名称和一些标记

当编译器编译源代码时，代码中定义的任何内容都会导致一个条目在表 2.1 所描述的相应表中被创建。另外，编译器还会检测源代码中引用到的类型、字段、方法、属性和事件。元数据用一组引用表来记录这些内容。表 2.2 展示了一些比较常用的元数据引用表。

表 2.2 常用的元数据引用表

元数据引用表名称	描 述
AssemblyRef	托管模块中引用的每一个程序集在 AssemblyRef 表中都有一个对应的条目。每一个条目包括绑定程序集所必需的信息：程序集名称(不含路径和扩展名)、版本号、语言文化(culture)以及一个公有密钥标记(通常为一个小的散列值，由发布者给出的公有密钥产生，标识被引用程序集的发布者)。另外，每一个条目还包括一些标记和一个散列值。该散列值是被引用程序集中的位的一个校验和。CLR 完全忽略该散列值，并且在将来也可能继续如此
ModuleRef	托管模块中有时会引用到实现在同一程序集中其他不同模块内的一些类型，这些模块每一个都在 ModuleRef 表中有一个对应的条目。每一个条目包括模块的文件名和扩展名(不含路径)。该表用来绑定那些实现在相同程序集中不同模块内的类型
TypeRef	托管模块中引用的每一个类型在 TypeRef 表中都有一个对应的条目。每一个条目包括类型的名称和一个指向类型所在位置的指针。如果该类型在另一个类型内实现，那么这个指针指向一个 TypeRef 条目。如果该类型在同一个模块中实现，那么这个指针指向一个 ModuleDef 条目。如果该类型在同一程序集的其他模块中实现，那么这个指针指向一个 ModuleRef 条目。如果该类型在其他不同的程序集中实现，那么这个指针指向一个 AssemblyRef 条目
MemberRef	托管模块中引用的每一个成员(字段、方法以及属性方法和事件方法)(译注：这里的属性方法指的是属性编译为 IL 代码后产生的 get 访问器方法和/或 set 访问器方法，事件方法指的是事件编译为 IL 代码后产生的 add 访问器方法和/或 remove 访问器方法。当托管模块引用一个属性或者事件时，它并不会引用属性或者事件本身的元数据，而是直接引用相应的访问器方法元数据)在 MemberRef 表中都有一个对应的条目。每一个条目包括成员的名称，成员的签名，以及一个指向 TypeRef 表中对应条目(其所标识的类型中定义了该成员)的指针

还有许多表未列入表2.1和表2.2，这里仅仅是希望能让大家了解一下编译器产生的各种元数据信息。前面还提到过清单元数据表，它将放在本章稍后讨论。

有许多工具可以用来查看托管PE文件中的元数据。我个人喜欢用ILDasm.exe，即IL反汇编器。可以执行下面的命令行来查看元数据表：

```
ILDasm /Adv App.exe
```

上面的命令将导致ILDasm.exe运行，并加载App.exe程序集。其中/Adv命令行开关告诉ILDasm启用一些"高级"的菜单项。这些高级菜单项可以在【视图】菜单中找到。如果想以一种漂亮、可读的形式来查看元数据，可以选择【视图.元信息.显示!】菜单项(或者按下Ctrl+M键)，它会显示如下信息：

```
ScopeName : App.exe
MVID      : {ED543DFC-44DD-4D14-9849-F7EC1B840BD0}
===========================================================
Global functions
-------------------------------------------------------

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

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

TypeDef #1
-------------------------------------------------------
    TypDefName    : App  (02000002)
    Flags         : [Public] [AutoLayout] [Class] [AnsiClass]  (00100001)
    Extends       : 01000001 [TypeRef] System.Object
    Method #1 [ENTRYPOINT]
    -------------------------------------------------------
        MethodName    : Main (06000001)
        Flags         : [Public] [Static] [HideBySig] [ReuseSlot]  (00000096)
        RVA           : 0x00002050
        ImplFlags     : [IL] [Managed]  (00000000)
        CallCnvntn    : [DEFAULT]
        ReturnType    : Void
        1 Arguments
            Argument #1:  SZArray String
        1 Parameters
            (1) ParamToken : (08000001) Name : args flags: [none] (00000000)
```

```
   Method #2
   -------------------------------------------------------------
      MethodName     : .ctor (06000002)
      Flags          : [Public] [HideBySig] [ReuseSlot] [SpecialName]
         [RTSpecialName] [.ctor]  (00001886)
      RVA            : 0x00002068
      ImplFlags      : [IL] [Managed]  (00000000)
      CallCnvntn     : [DEFAULT]
      hasThis
      ReturnType: Void
      No arguments.

TypeRef #1 (01000001)
-------------------------------------------------------------
Token:                 0x01000001
ResolutionScope:       0x23000001
TypeRefName:           System.Object
   MemberRef #1
   -------------------------------------------------------------
      Member: (0a000003) .ctor:
      CallCnvntn: [DEFAULT]
      hasThis
      ReturnType: Void
      No arguments.

TypeRef #2 (01000002)
-------------------------------------------------------------
Token:                 0x01000002
ResolutionScope:       0x23000001
TypeRefName:           System.Diagnostics.DebuggableAttribute
   MemberRef #1
   -------------------------------------------------------------
      Member: (0a000001) .ctor:
      CallCnvntn     : [DEFAULT]
      hasThis
      ReturnType     : Void
      2 Arguments
         Argument #1:  Boolean
         Argument #2:  Boolean

TypeRef #3 (01000003)
-------------------------------------------------------------
Token:                 0x01000003
ResolutionScope:       0x23000001
TypeRefName:           System.Console
```

```
    MemberRef #1
    -------------------------------------------------------
        Member: (0a000002) WriteLine:
        CallCnvntn      : [DEFAULT]
        ReturnType      : Void
        1 Arguments
            Argument #1:  String

Assembly
-------------------------------------------------------
    Token: 0x20000001
    Name : App
    Public Key    :
    Hash Algorithm : 0x00008004
    Major Version: 0x00000000
    Minor Version: 0x00000000
    Build Number: 0x00000000
    Revision Number: 0x00000000
    Locale: <null>
    Flags : [SideBySideCompatible]  (00000000)
    CustomAttribute #1 (0c000001)
    -------------------------------------------------------
        CustomAttribute Type: 0a000001
        CustomAttributeName: System.Diagnostics.DebuggableAttribute ::
            instance void .ctor(bool,bool)
        Length: 6
        Value : 01 00 00 01 00 00                    >               <
        ctor args: ( <can not decode> )

AssemblyRef #1
-------------------------------------------------------
    Token: 0x23000001
    Public Key or Token: b7 7a 5c 56 19 34 e0 89
    Name: mscorlib
    Major Version: 0x00000001
    Minor Version: 0x00000000
    Build Number: 0x00000c1e
    Revision Number: 0x00000000
    Locale: <null>
    HashValue Blob:
    Flags: [none] (00000000)

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

所幸 ILDasm 对元数据表做了一些处理，并且在适当的地方合并了信息，这样我们就不必自己去分析原始表中的信息了。例如，在上面所显示的信息中，我们可以看到 ILDasm 显示了一个 TypeDef 条目，并且紧接着在第一个 TypeRef 条目之前显示了其内相关的成员定义信息。

目前大家还没有必要完全理解这些信息，只需要知道 App.exe 包含了一个名为 App 的 TypeDef 即可。该类型标识的是一个从 System.Object(一个引用自另一个程序集中的类型)继承而来的公有类。App 类型还定义了两个方法：Main 和.ctor(一个构造器)。

Main 是一个静态的公有方法，它的实现代码为 IL 指令(相对于本地 CPU 指令而言，比如 x86)。另外，它还有一个 void 返回值类型，并且接受一个 String 数组参数 args。构造器方法(总是以.ctor 的名字显示)也是一个公有方法，并且代码也是 IL 指令。构造器方法也有一个 void 返回值类型，并且除了 this 指针以外，没有任何其他参数。其中 this 指针指向构造器方法被调用时正在被构造的对象内存。

强烈建议大家经常使用 ILDasm。它可以显示很多丰富的信息。对 ILDasm 显示的内容理解得越多，大家对 CLR 及其能力的掌握也就会越好。大家将会看到，本书很多地方都使用了 ILDasm。

让我们来看一下有关 App.exe 程序集的一些统计结果。选择 ILDasm 中的【视图.统计】菜单项，我们将会看到以下信息：

```
File size                 : 3072
   PE header size         : 512 (496 used)     (16.67%)
   PE additional info     : 923                (30.05%)
   Num.of PE sections     : 3
   CLR header size        : 72                 ( 2.34%)
   CLR meta-data size     : 520                (16.93%)
   CLR additional info    : 0                  ( 0.00%)
   CLR method headers     : 24                 ( 0.78%)
   Managed code           : 18                 ( 0.59%)
   Data                   : 828                (26.95%)
   Unaccounted            : 175                ( 5.70%)

Num.of PE sections        : 3
   .text    - 1024
   .rsrc    - 1024
   .reloc   - 512

CLR meta-data size        : 520
   Module            -    1 (10 bytes)
   TypeDef           -    2 (28 bytes)     0 interfaces, 0 explicit layout
   TypeRef           -    3 (18 bytes)
```

```
    MethodDef          -    2 (28 bytes)     0 abstract, 0 native, 2 bodies
    MemberRef          -    3 (18 bytes)
    ParamDef           -    1 (6 bytes)
    CustomAttribute    -    1 (6 bytes)
    Assembly           -    1 (22 bytes)
    AssemblyRef        -    1 (20 bytes)
    Strings            -  128 bytes
    Blobs              -   40 bytes
    UserStrings        -    8 bytes
    Guids              -   16 bytes
    Uncategorized      -  172 bytes

CLR method headers : 24
    Num.of method bodies   - 2
    Num.of fat headers     - 2
    Num.of tiny headers    - 0

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

其中我们可以看到文件的大小(字节数)和组成文件的各个部分的大小(字节数和百分比)。对于这个很小的 App.exe 应用程序，PE 表头和元数据占据了文件的大部分内容。实际上 IL 代码仅仅占用了 18 个字节。当然，随着应用程序的增长，它会重用大多数自己的类型和对其他类型与程序集的引用，这将会极大地降低元数据和表头信息在整个文件中所占的比重。

2.3　将模块组合为程序集

上一节讨论的 App.exe 文件不仅仅是一个含有元数据的 PE 文件，它还是一个**程序集**(assebmly)。程序集是包含一个或多个类型定义文件和资源文件的集合。在程序集包含的所有文件中，有一个文件用于保存**清单**(manifest)。清单是另外一组元数据表的集合，其中主要包含了程序集中一部分文件的名称。另外，清单还描述了程序集的版本、语言文化、发布者、公有导出类型，以及组成该程序集的所有文件。

程序集是 CLR 操作的对象。也就是说，CLR 总是先加载包含清单元数据表的文件，然后利用该清单来获取程序集中的其他文件。下面是应该牢记的一些程序集的特性：

- 程序集定义了可重用的类型。
- 程序集标识有一个版本号。
- 程序集可以包含与之相关的安全信息。

除了包含清单元数据表的文件外，程序集中的其他各个文件没有上述这些特性。

要对类型打包、进行版本控制、实施安全策略，并使用它们，我们必须将类型放在程序集的模块部分。大多数情况下，一个程序集仅包含一个文件，就像前面的App.exe例子那样。但是，一个程序集也可以包含多个文件，例如一些带有元数据的PE文件和一些.gif和.jpg之类的资源文件。把程序集视为一个逻辑上的EXE或者DLL将有助于我们对它的理解。

相信大家读到这里，很多人都会奇怪为什么微软会引入程序集这个概念。理由是程序集允许我们分离可重用类型的逻辑表示和物理表示。例如，一个程序集可以包含好几个类型。我们可能会把常用的类型放在一个文件中，而把很少使用的类型放在另一个文件中。假设我们的程序集是通过互联网下载的方式来部署，而客户端又从来不去访问那些很少使用的类型，那么包含它们的文件就不必被下载到客户的机器上。例如，致力于UI(用户界面)控件的ISV(独立软件开发商)可能会选择在一个单独的模块中实现活动辅助(Active Accessibility)类型(这也是满足微软的徽标要求)。这样只有那些需要额外的活动辅助特性的用户才会下载该模块。

我们可以通过在应用程序的配置文件中指定一个codeBase元素(第3章将予以讨论)来配置应用程序需要下载的程序集文件。codeBase元素标识了一个可以找到程序集所有文件的URL地址。当试图加载一个程序集的文件时，CLR首先获得codeBase元素的URL地址，然后检查本地机器的下载缓存看是否已经存在该文件。如果存在，该文件将被加载。否则，CLR将从codeBase元素的URL地址中将该文件下载到缓存中。如果找不到该文件，CLR将在运行时抛出FileNotFoundException异常。

使用多文件程序集通常有以下三个原因：

- 可以将类型分别实现在不同的文件中，从而允许文件在互联网环境中进行增量下载。将类型划分到不同的文件中还允许软件在套装零售(shrink-wrapped)的情形下进行分段地打包和部署。

- 可以按需要向程序集中添加资源或数据文件。例如，我们可能有一个计算某些保险信息的类型。该类型需要访问某些保险统计表来进行计算。这时我们就可以借助一个工具(如程序集链接器，即AL.exe，后面予以讨论)，使得在维持保险统计表和程序代码分离的同时，仍然将它视为程序集的一部分，而不必非要把这些保险统计表嵌入到源代码中。顺便提一句，这些

数据文件可以是任何格式：文本文件、Excel 电子表格、Word 表格、或者任何我们喜欢的格式——只要我们的应用程序知道如何分析这些文件的内容。

- 可以使我们创建的程序集包含一些用不同编程语言实现的类型。当编译 C#源代码时，编译器产生一个模块。当编译 Visual Basic 源代码时，编译器产生另外一个模块。我们可以用 C#实现一些类型，用 Visual Basic 实现一些类型，或用其他语言实现另外一些类型。然后借助一个工具，我们便可以将这些模块组合为一个单独的程序集。对于使用该程序集的开发人员来说，它只是包含了许多类型，他们甚至不知道它是用不同的编程语言实现的。顺便提一句，我们也可以对每一个模块运行 ILDasm.exe 来获取一个 IL 源代码文件。然后运行 ILAsm.exe，传入所有的 IL 源代码文件。ILAsm.exe 将产生一个包含所有类型的文件。这个技巧要求源代码编译器产生的只是 IL 代码，所以对于 Visual C++之类的编译器并不适用。

重要　总而言之，程序集是一个可重用、可实施版本策略和安全策略的单元。它允许我们将类型和资源划分到不同的文件中，这样程序集的使用者便可以决定将哪些文件打包在一起部署。一旦 CLR 加载了程序集中包含清单的那个文件，它就可以确定程序集的其他文件中哪些包含了程序正在引用的类型和资源。任何程序集的使用者仅需要知道包含清单的文件名称。文件的划分对使用者是透明的，并且可以在将来改变，同时又不会破坏现有应用程序的行为。

要生成一个程序集，我们必须为其选择一个 PE 文件作为清单的保存者。我们也可以创建一个单独的 PE 文件，让它除了清单外不包括任何其他内容。表 2.3 显示了一些清单元数据表，正是有了它们一个托管模块才得以成为一个程序集。

表 2.3 清单元数据表

清单元数据表名	描 述
AssemblyDef	如果模块标识为一个程序集，那么它将在 AssemblyDef 表中有一个对应的条目。该条目包括程序集的名称(不含路径和扩展名)，版本号(主版本号、次版本号、生成版本号和修订版本号)，语言文化，一些标记，散列算法，以及发布者的公有密钥(可能为 null)
FileDef	除了清单所在的 PE 文件外，程序集包含的其他 PE 文件和资源文件都在 FileDef 表中含有一个对应的条目。该条目包含文件名与扩展名(不包括路径)，散列值，和一些标记。如果程序集仅包括清单所在的 PE 文件，FileDef 表中将不包含任何条目
ManifestResourceDef	程序集中包含的每一个资源都在 MainfestResourceDef 表中有一个对应的条目。该条目包括资源名称，一些标记(public、private)，和一个该资源所在的资源文件或流在 FileDef 表中的索引。如果该资源不是一个单独的文件(例如.jpeg 或者.gif)，那么它就是嵌入在 PE 文件中的一个流。对于嵌入的资源，该条目还将包括一个表示资源流在 PE 文件中的起始偏移
ExportedTypesDef	程序集中所有的 PE 模块导出的每一个公有类型都在 ExportedTypesDef 表中有一个对应的条目。该条目包括类型名称，一个 FileDef 表中的索引(表示程序集中实现该类型的那个文件)，和一个 TypeDef 表中的索引。注意：为了节省文件空间，清单 PE 文件中导出的类型不会在该表中重复出现，因为其类型信息可以通过清单 PE 文件内的 TypeDef 元数据表获得

清单为程序集的使用者和其各个部分之间提供了一层间接关联，也使得程序集得以实现自描述。另外需要注意的是，虽然包含清单的文件知道程序集中的其他文件，但其他文件本身却并不清楚它们是一个程序集的一部分。

注意　包含清单的程序集文件中还有一个AssemblyRef表。一个程序集中所有文件引用的其他程序集都在该表中有一个对应的条目。这允许一些工具只打开程序集的清单便可以了解它引用到的所有程序集，而无需再打开程序集中的其他文件。AssemblyRef表中的条目也是程序集得以实现自描述的一个原因。

当使用命令行开关/t[arget]:exe、/t[arget]:winexe、或/t[arget]:library时，C#编译器的输出结果将是一个程序集，即一个包含着清单元数据表的PE文件。这三个命令行开关得到的文件分别为CUI可执行文件、GUI可执行文件、或DLL文件。

除了这些命令行开关以外，C#编译器还支持/t[arget]:module命令行开关。该命令行开关告诉编译器产生一个不包括清单元数据表的PE文件，而且得到的PE文件总是一个DLL文件，在其中的类型被外界访问之前，它必须首先被添加到一个程序集中。当使用/t:module命令行开关时，C#编译器默认输出扩展名为.netmodule的文件。

重要　很不幸，Visual Studio .NET集成开发环境（integrated development environment，简称IDE）本身不支持创建多文件程序集。如果需要创建多文件程序集，必须求助于命令行工具。

有许多方式可以将模块添加到一个程序集中。如果使用C#编译器来生成含有清单的PE文件，那么我们可以使用/addmodule命令行开关。为了帮助大家理解怎样生成一个多文件程序集，下面演示一个例子。假设我们有两个源代码文件：

- RUT.cs，其中包含着很少使用的类型
- FUT.cs，其中包含着经常使用的类型

首先将很少使用的类型编译为一个模块，这样如果用户从不访问这些类型，程序集将不需要部署该模块。

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

上面的命令行将使C#编译器创建一个RUT.netmodule文件。该文件是一个标准DLL PE文件，但它本身不能被CLR加载。

下面再将经常使用的类型编译为一个模块。因为它们很常用，所以我们将该模块作为程序集清单的保存者。实际上，因为该模块现在将代表整个程序集，我们可以把输出文件的名字改为JeffTypes.dll，而不是采用FUT.dll：

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

上面的命令告诉C#编译器将FUT.cs文件编译为JeffTypes.dll文件。因为指定了/t:library，所以JeffTypes.dll就是包含清单元数据表的DLL PE文件。其中命令行开关/addmodule:RUT.netmodule告诉编译器RUT.netmodule文件应该被作为程序集的一部分来对待。具体而言，/addmodule命令行开关将告诉编译器把该文件加到FileDef清单元数据表中，并把RUT.netmodule文件中的公有导出类型加入到ExportedTypesDef清单元数据表中。

一旦编译器完成了所有的工作，将有如图2.1所示的两个文件被创建。其中右面的模块包含着清单元数据表。

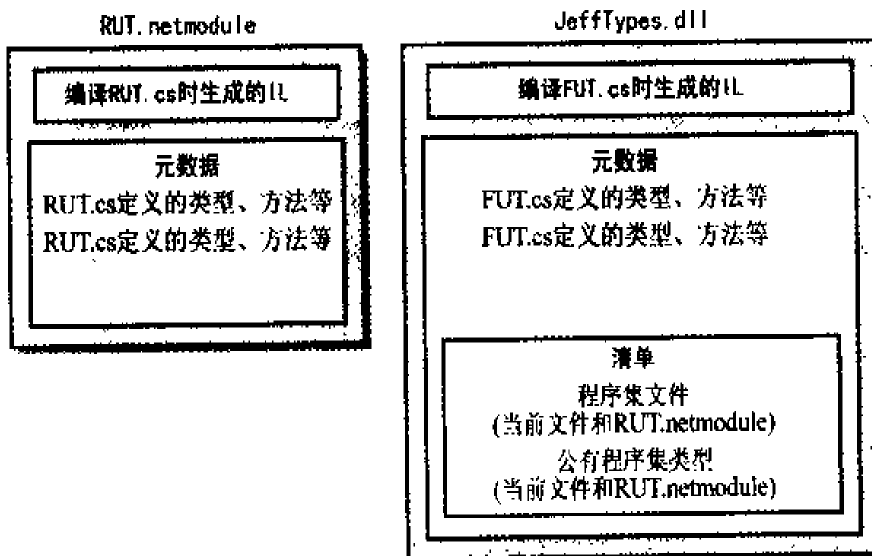


图2.1 包含两个托管模块的多文件程序集，其中右面的模块包含着清单元数据表

RUT.netmodule文件包含着编译RUT.cs产生的IL代码，一些定义元数据表(其中描述了在RUT.cs中定义的类型、方法、字段、属性、事件等内容)，以及一些引用元数据表(其中描述了RUT.cs文件中引用到的类型、方法等内容)。JeffTypes.dll是一个单独的文件。和RUT.netmodule文件类似，该文件

也包括编译 FUT.cs 产生的 IL 代码，以及一些定义元数据表和一些引用元数据表。除此之外，JeffTypes.dll 还包括有额外的清单元数据表，这使得它成为一个程序集。清单元数据表描述了组成程序集的所有文件(JeffTypes.dll 文件本身和 RUT.netmodule 文件)，以及所有从 JeffTypes.dll 文件和 RUT.netmodule 文件中导出的公有类型。

> **注意**　实际上，清单元数据表并不包括清单本身所在 PE 文件中的导出类型。这种优化的目的是减少 PE 文件中清单信息所需的字节数。所以前面所说的"清单元数据表包括了所有从 JeffTypes.dll 文件和 RUT.netmodule 文件中导出的公有类型"并非百分之百正确。然而，这种说法还是准确地反映了清单在逻辑上的导出内容。

一旦生成了 JeffTypes.dll 程序集，我们便可以使用 ILDasm.exe 来查看元数据中的清单表来验证该程序集文件是否确实引用了 RUT.netmodule 文件中的类型，我们还将看到其中的 FileDef 表和 ExportedTypesDef 表。下面是一些表的示例：

```
File #1
-------------------------------------------------------
    Token: 0x26000001
    Name : RUT.netmodule
    HashValue Blob :  03 d4 09 ef 2d ac d3 4b 64 75 d7 81 cc 8e 88 7d 51
                      67 e2 5b
    Flags : [ContainsMetaData]  (00000000)


ExportedType #1
-------------------------------------------------------
    Token: 0x27000001
    Name: ARarelyUsedType
    Implementation token: 0x26000001
    TypeDef token: 0x02000002
    Flags : [Public] [AutoLayout] [Class] [AnsiClass]  (00100001)
```

从上面的示例中可以看出，RUT.netmodule 文件被当成了程序集的一部分。在 ExportedType 表中，我们可以看到有一个公有的导出类型 ARarelyUsedType。该类型的实现标记为 0x26000001，它表示实现类型的 IL 代码位于 RUT.netmodule 文件中。

注意 元数据标记是一个 4 字节的数值，其高位字节表示标记的类型(0x01=TypeRef，0x02=TypeDef，0x26=FileRef，0x27=ExportedType)(译注：第三项中的 FileRef 应该为 FileDef，元数据标记中的文件没有"引用"的概念，只有"定义"的概念。实际上 CorHdr.h 文件中使用的是"File"标记)。有关元数据标记更完整的列表，可参见.NET 框架 SDK 内 CorHdr.h 文件中的 CorTokenType 枚举类型。标记中低位的 3 个字节标识相关元数据表中的索引。例如上面的实现标记 0x26000001 指的是 FileRef(译注：同样应为 FileDef)表中的第一行(行从 1，而不是 0 开始计数)。

任何使用 JeffTypes.dll 程序集中类型的代码必须使用/r[eference]:JeffTypes.dll 命令行开关来编译。该命令行开关告诉编译器加载 JeffTypes.dll 程序集和所有在其 FileDef 表中列出的文件。编译器要求程序集中所有的文件都存在，并且有访问权限。如果将 RUT.netmodule 文件删除，C#编译器将产生以下错误信息：fatal error CS0009: 未能打开元数据文件"C:\JeffTypes.dll"——"导入程序集'C:\JeffTypes.dll'的模块'RUT.netmodule'时出错"——系统找不到指定的文件。这意味着要生成一个程序集，其引用的所有文件都必须存在。

代码的执行过程便是调用方法的过程。当一个方法被第一次调用时，CLR 将检测出该方法引用了哪些类型。CLR 然后尝试加载被引用程序集中包含清单的那个文件。如果方法引用的类型恰好在此文件中，CLR 将执行一些内部的簿记工作，然后便开始使用该类型。如果清单文件显示方法引用的类型在其他文件中，CLR 将试图加载这些必要的文件，并执行一些内部的簿记工作，然后再开始使用该类型。只有当引用一个类型的方法被调用时，CLR 才会加载该类型所在的程序集文件。这意味着一个应用程序的运行并不需要其引用的程序集中所有的文件都存在。

2.3.1 使用 Visual Studio .NET IDE 为项目添加程序集引用

如果使用 Visual Studio .NET IDE 来创建项目，我们就必须为项目添加其引用到的所有程序集。要做到这一点，首先打开【解决方案资源管理器】窗口，右击希望添加引用的项目，然后选择【添加引用】菜单项。这时会弹出【添加引用】对话框，如图 2.2 所示。

图 2.2 Visual Studio .NET 中的【添加引用】对话框

我们可以从列表中选择希望引用的托管程序集。如果想要的程序集不在列表中，可以选择【浏览】按钮来查找期望的程序集(包含清单的文件)并添加。其中【添加引用】对话框上的 COM 选项卡允许托管源代码访问非托管的 COM 服务器。【项目】选项卡允许目前的项目引用同一个解决方案中其他项目创建的程序集。

为了使我们创建的程序集出现在.NET 选项卡的列表中，可以将下面的子键添加到注册表中：

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\
AssemblyFolders\MyLibName
```

其中 MyLibName 是我们创建的一个惟一的名字——Visual Studio .NET 不会显示该名字。在创建了上面的子键之后，还应该修改其默认的字符串值使其指向一个包含程序集文件的目录(例如"C:\Program Files\MyLibPath")。

2.3.2 使用程序集链接器

有时候我们可能会使用程序集链接器(Assembly Linker，即 AL.exe)，而不是 C#编译器，来生成程序集。如果我们要创建的程序集包含来自不同编译器生成的模块，而使用的编译器又不支持类似于 C#中/addmodule 那样的命令行开关，或者在生成模块时还不知道程序集的打包需求，这时程序集链接器就显得非常有用。我们也可以使用 AL.exe 来生成仅含资源的程序集(又称**卫星程序集**，satellite

assembly，本章稍后会有详述)，其典型地用于解决本地化问题。

AL.exe 可以产生一个除了清单外不包括任何其他内容的 EXE 或者 DLL PE 文件，其中的清单只用于描述其他模块中的类型。为了理解 AL.exe 如何工作，下面我们来改变一下 JeffTypes.dll 程序集的生成方式：

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

图 2.3 显示了执行上述命令后的结果。

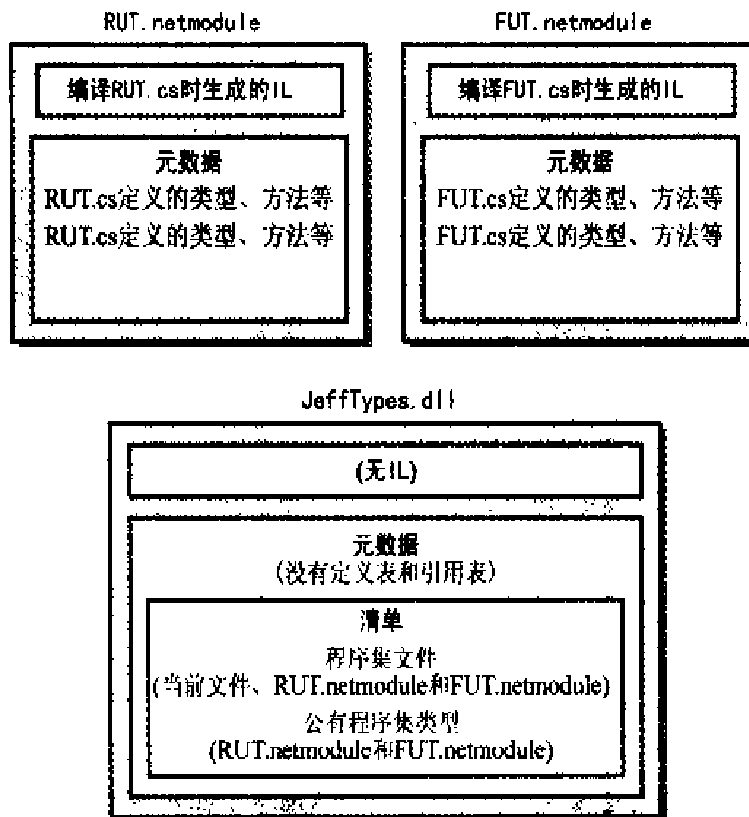


图 2.3 包含三个托管模块的多文件程序集，其中一个仅含有清单

在上面的例子中，我们首先创建了两个单独的模块 RUT.netmodule 和 FUT.netmodule，它们本身并不是程序集(因为它们没有包含清单元数据表)。然后又生成了第三个文件 JeffTypes.dll，它是一个很小的 DLL PE 文件(因为使用了/t[arget]:library 命令行开关)，其中除了清单元数据表外不包含任何 IL 代码，清单元数据表中的信息表明 RUT.netmodule 和 FUT.netmodule 是程序集的一部分。最后我们得到的程序集包含三个文件：JeffTypes.dll、RUT.netmodule 和 FUT.netmodule。程序集链接器不能将多个文件组合成一个单独的文件。

AL.exe 还能够产生 CUI 或 GUI PE 文件(使用/t[arget]:exe 或 /t[arget]:winexe 命令行开关)，但这很少使用，因为这意味着在我们产生的 EXE PE 文件中，必须含有足够的 IL 代码来调用另一个模块中的方法。当使用/main 命令行开关时，AL.exe 将会为我们产生这样的 IL 代码。

```
csc /t:module /r:JeffTypes.dll App.cs
al /out:App.exe /t:exe /main:App.Main app.netmodule
```

这里第一行将 App.cs 文件编译为一个模块，第二行将产生一个包含清单元数据表的 App.exe PE 文件。另外由于使用了/main:App.main 命令行开关，AL.exe 还产生了一个很小的全局函数，即__EntryPoint 函数，它包含以下 IL 代码：

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

该段代码仅仅是调用了 App.netmodule 文件中 App 类型的 Main 方法。

AL.exe 的/main 命令行开关并不是十分有用，因为一般情况下，我们不会为应用程序创建这样一个程序集(其清单元数据表所在的 PE 文件不包含应用程序的入口点)。这里仅仅是让大家了解存在这样一种选择。

2.3.3　在程序集中包含资源文件

当使用 AL.exe 来创建一个程序集时，我们可以用命令行开关/embed[resource]将一些资源文件(非 PE 文件)添加到程序集中。该命令行开关接受一个文件(任何类型的文件)，然后将其中的内容嵌入到产生的 PE 文件中。同时，清单中的 ManifestResourceDef 表将被更新以反映该资源的存在。

AL.exe 还支持/link[resource]命令行开关，它也接受一个包含资源的文件。但是/link[resource]命令行开关将只更新清单中的 ManifestResourceDef 表和 FileDef 表，以反映资源的存在，并标识出程序集的哪个文件包含着资源文件。资源文件本身不会被嵌入到程序集 PE 文件中，它仍然保持独立，并且必须和其他程序集文件一起打包、部署。

和 AL.exe 类似，CSC.exe 也允许我们将资源包含到 C#编译器产生的程序集中。C#编译器的/resource 命令行开关会把指定的资源文件嵌入到产生的程序集 PE 文件中，并更新 ManifestResourceDef 表中的内容。而/linkresource 命令行开关则会向 ManifestResourceDef 和 FileDef 清单表中添加一个条目，使其指向一个单独的资源文件。

关于资源需要注意的最后一点是，我们也可以将标准 Win32 资源嵌入到一个程序集中。这可以通过为 AL.exe 或 CSC.exe 添加/win32res 命令行开关，并指定一个.res 文件路径来实现。另外还可以为 AL.exe 或 CSC.exe 添加/win32icon 命令行开关，并指定一个.ico 文件路径，将一个标准 Win32 图标资源嵌入到程序集文件中。利用这项技术，我们可以在 Windows 的【资源管理器】中将一个托管 EXE 文件显示为一个指定的图标。

2.4 程序集版本资源信息

当 AL.exe 或 CSC.exe 产生 PE 文件时，它们还会将一个标准 Win32 版本资源嵌入到 PE 文件中。用户可通过查看文件属性看到该资源。图 2.4 显示了【JeffTypes.dll 属性】对话框中的【版本】选项卡。

图 2.4 【JeffTypes.dll 属性】对话框的【版本】选项卡

另外，我们还可以使用 Visual Studio .NET 中的资源编辑器(如图 2.5 所示)来查看或者修改版本资源字段。

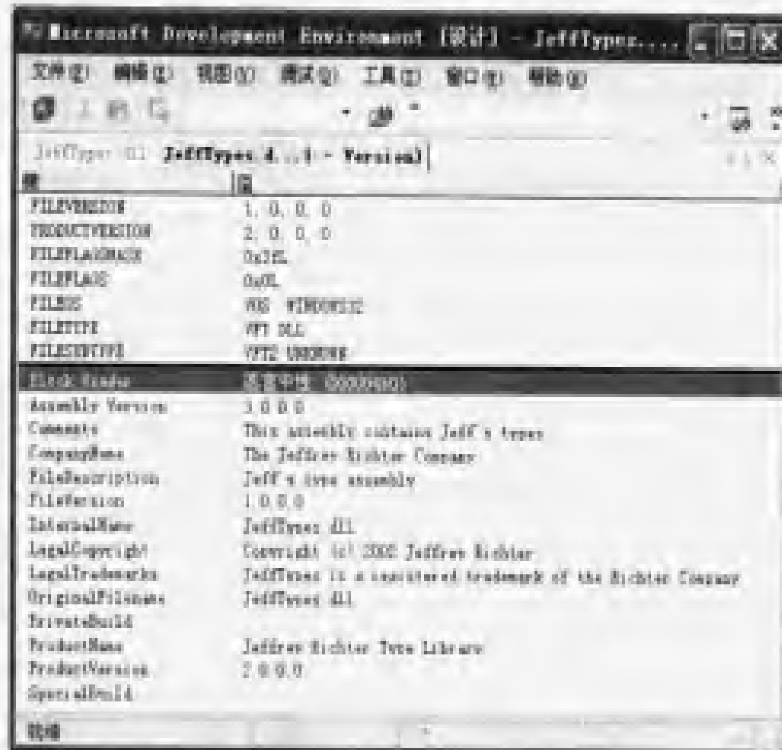


图2.5 Visual Studio .NET中的资源编辑器

当生成一个程序集时，我们应该使用一些定制特性来设置其版本资源字段。注意，这些定制特性应该被应用在源代码中的程序集层次上。下面的代码将产生图2.5中所示的版本信息：

```
using System.Reflection;

// 设置 CompanyName、LegalCopyright 和 LegalTrademarks 版本字段
[assembly:AssemblyCompany("The Jeffrey Richter Company")]
[assembly:AssemblyCopyright("Copyright (c) 2002 Jeffrey Richter")]
[assembly:AssemblyTrademark(
   "JeffTypes is a registered trademark of the Richter Company")]

// 设置 ProductName 和 ProductVersion 版本字段
[assembly:AssemblyProduct("Jeffrey Richter Type Library")]
[assembly:AssemblyInformationalVersion("2.0.0.0")]

// 设置 FileVersion、AssemblyVersion、
// FileDescription 和 Comments 版本字段
[assembly:AssemblyFileVersion("1.0.0.0")]
[assembly:AssemblyVersion("3.0.0.0")]
[assembly:AssemblyTitle("Jeff's type assembly")]
[assembly:AssemblyDescription("This assembly contains Jeff's types")]

// 设置语言文化(参见本章后面"2.5 语言文化"一节)
[assembly:AssemblyCulture("")]
```

表 2.4 显示了版本资源字段以及和它们对应的一些定制特性。如果使用 AL.exe 来生成程序集，我们可以利用命令行开关来代替这些定制特性设置版本信息。表 2.4 的第二列显示了对应于每一个版本资源字段的 AL.exe 命令行开关。注意 C#编译器没有提供这些命令行开关。因此，一般来讲，使用定制特性来设置版本信息应该是一种首选的方式。

重要 当使用 Visual Studio .NET 创建 C#项目时，它会自动为我们产生一个 AssemblyInfo.cs 文件，该文件包含了本节描述的几乎所有的程序集特性，另外几个特性会在第 3 章中讨论。我们可以直接打开 AssemblyInfo.cs 文件修改其中和程序集相关的信息。然而，Visual Studio .NET 为我们创建的这个文件有一些问题，本章后面将会予以解释。在一个实际的产品项目中，我们必须修改该文件的内容。

表 2.4 版本资源字段和它们对应的 AL.exe 命令行开关以及定制特性

版本资源	命令行开关	定制特性/注解
FILEVERSION	/fileversion	System.Reflection.AssemblyFileVersionAttribute
PRODUCTVERSION	/productversion	System.Reflection.AssemblyInformationalVerSionAttribute
FILEFLAGSMASK	(无)	总是设置为 VS_FFI_FILEFLAGSMASK（在 WinVer.h 中定义为 0x0000003F)
FILEFLAGS	(无)	总为 0
FILEOS	(无)	目前总为 VOS__WINDOWS32
FILETYPE	/target	如果指定了 /target:exe 或 /target:winexe，将设置为 VFT_APP；如果指定了 /target:library，将设置为 VFT_DLL
FILESUBTYPE	(无)	总是设置为 VFT2_UNKNOWN(该字段对于 VFT_APP 和 VFT_DLL 没有意义)
AssemblyVersion	/version	System.Reflection.AssemblyVersionAttribute

续表

版本资源	命令行开关	定制特性/注解
Comments	/description	System.Reflection.AssemblyDescriptionAttribute
CompanyName	/company	System.Reflection.AssemblyCompanyAttribute
FileDescription	/title	System.Reflection.AssemblyTitleAttribute
FileVersion	/version	System.Reflection.AssemblyVersionAttribute
InternalName	/out	设置为指定的输出文件名称(不含扩展名)
LegalCopyright	/copyright	System.Reflection.AssemblyCopyrightAttribute
LegalTrademarks	/trademark	System.Reflection.AssemblyTrademarkAttribute
OriginalFilename	/out	设置为输出文件的名称(不含路径)
PrivateBuild	(无)	总为空
ProductName	/product	System.Reflection.AssemblyProductAttribute
ProductVersion	/productversion	System.Reflection.AssemblyInformationalVersionAttribute
SpecialBuild	(无)	总为空

2.4.1 版本号

在前面一节中，我们看到一个程序集上可以应用多个版本号。所有这些版本号都有着相同的格式：每一个都包括四个由点号分开的部分，如表2.5所示。

表2.5 版本号格式

版本资源	主版本号	次版本号	生成版本号	修订版本号
示例	2	5	719	2

表2.5显示了一个版本号的例子：2.5.719.2。前两个版本号组成了“面向公众”的版本部分。公众将认为该例子中程序集的版本为2.5。第三个版本号719，表示程序集的生成版本。如果大家的公

司每天都生成一次程序集，那么该程序集的生成版本号也应该随之递增。最后一个版本号 2，表示对生成版本的修订版。如果因为某种原因(比如可能是解决了一个影响其他工作的严重 bug)，必须在一天内对一个程序集进行多次生成，那么修订版本号就应该递增。

微软使用上述的版本标号模式，当然这仅仅是一个建议。如果愿意，我们也可以设计自己的版本标号模式。CLR 的惟一假设就是较大的版本号表示更新的版本。

大家可能注意到一个程序集有三个相关的版本号，这是令人感到比较遗憾的，因为这会导致很多混乱。下面对每一种版本号的意图及其使用方式做一解释：

- **AssemblyFileVersion** 该版本号存储在 Win32 版本资源中。它仅仅是一个辅助性的信息，CLR 既不查看、也不关心该版本号。典型地，我们可以设置其主版本号和次版本号来表示希望被公众看到的版本。然后每次执行一次生成，便递增其生成版本号以及修订版本号。理想情况下，微软的工具(如 CSC.exe 或者 AL.exe)应该为我们自动更新生成版本号和修订版本号(根据生成时的数据/时间)，但很不幸，事实并非如此。该版本号可以在 Windows 中的【资源管理器】看到，并且可以通过它来判断一个程序集文件是何时生成的。

- **AssemblyInformationalVersionAttribute** 该版本号也存储在 Win32 版本资源中，也仅作为辅助性的信息之用，CLR 也不查看、关心该版本号。该版本号表示包含程序集的产品的版本。例如，MyProduct 的 2.0 版本可能包含好几个程序集；其中的一个程序集因为没有和 MyProduct 的 1.0 版本一起发布，它将被标识为 1.0 版本。典型地，我们可以设置该版本号的主要部分和次要部分，来表示产品的公共版本。然后，每次将所有的程序集打包为一个完整的产品时，便递增其生成版本号和修订版本号部分。

- **AssemblyVersion** 该版本号存储在 AssemblyDef 清单元数据表中。当绑定强命名程序集(强命名程序集将在第 3 章讨论)时，CLR 会用到该版本号。这个版本号非常重要，它用来惟一地标识一个程序集。当开始开发一个程序集时，我们就应该设置它的主版本号、次版本号、生成版本号和修订版本号，并且如果没有开始开发程序集的下一个部署版本，便不应该改变它们。当生成一个程序集时，其引用的所有程序集的 AssemblyVersion 会被嵌入到程序集的 AssemblyRef 表内相应的条目中。这意味着一个程序集将会和它所引用的程序集的特定版本紧紧绑定在一起。

重要 CSC.exe 和 AL.exe 工具支持程序集版本号(AssemblyVersion)随着每一次生成进行递增的能力。然而，这项功能却是一个 bug，应该避免使用，因为改变程序集版本号将会损害任何引用到它的其他程序集。当创建一个新项目时，Visual Studio .NET 为我们自动产生的 AssemblyInfo.cs 文件也有错误：它会设置 AssemblyVersion 特性以使它的主版本号和次版本号部分为 1.0，而生成版本号和修订版本号部分会被编译器自动更新。我们应该修改该文件，手工指定程序集版本号的四个部分。

2.5 语言文化

和版本号类似，语言文化(culture)也是程序集标识的一部分。例如，我们可能有一个专门应用于德语的程序集，一个专门应用于瑞士德语的程序集，以及一个专门应用于美国英语的程序集，等等。语言文化通过一个包含主标记和次标记的字符串(在 RFC1766 中有描述)来标识。表 2.6 显示了一些例子：

表 2.6 程序集语言文化标记示例

主标记	次标记	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

一般情况下，如果我们创建的是包含代码的程序集，就不宜再为其指派特定的语言文化。这是因为代码实际上不会预设任何特定的语言文化。没有指定语言文化的程序集被称为是**语言文化中性**(culture neutral)的程序集。

如果设计的是一个有着特定语言文化资源的应用程序，微软强烈建议我们首先创建一个包含代码和应用程序默认(或后备)资源的程序集。当生成程序集时，暂不指派特定的语言文化。该程序集将被其他的程序集引用来创建和操作类型。

然后，我们再创建一个或多个包含特定语言文化资源的单独的程序集——这些程序集中将不包含任何代码。这些标识着特定语言文化的程序集又称**卫星程序集**(satellite assembly)。卫星程序集中指定了能够正确反映其中资源的语言文化。我们应该为每一种希望支持的语言文化创建一个卫星程序集。

通常使用 AL.exe 工具来创建卫星程序集。因为卫星程序集中不包含任何代码，所以不会用到编译器。当使用 AL.exe 时，我们用/c[ulture]:text 命令行开关来指定所需的语言文化，其中 text 是一个字符串，例如表示美国英语的“en-US”。当部署卫星程序集时，我们应该将它放在一个子目录下，其名称要和前面指定语言文化时的 text 字符串相匹配。例如，如果应用程序的基目录为 C:\MyApp，那么代表美国英语的卫星程序集就应该放在 C:\MyApp\en-US 子目录下。在运行时，我们可以使用 System.Resources.ResourceManager 类来访问卫星程序集中的资源。

注意 虽然不提倡创建包含代码的卫星程序集，但还是有可能做到的。如果我们愿意，仍然可以用 System.Reflection.AssemblyCultureAttribute 定制特性代替 AL.exe 的/culture 命令行开关来指定语言文化。示例如下：

```
// 将程序集的语言文化设置为瑞士德语
[assembly:AssemblyCulture("de-CH")]
```

通常情况下，我们创建的程序集不应该引用卫星程序集。换句话说，一个程序集的 AssemblyRef 条目指向的都应该是语言文化中性的程序集。如果想访问一个卫星程序集中的类型或成员，我们应该使用第 20 章中的反射技巧。

2.6　简单应用程序部署(私有部署程序集)

前面我们详细讨论了怎样生成模块，以及怎样将它们组合到一个程序集中。本节向大家介绍怎样打包和部署程序集，只有这样用户才能运行我们的应用程序。

程序集并未规定或要求任何特别的打包方式。打包一组程序集最容易的方式就是直接复制这些文件。例如，我们可以把所有的程序集文件放在 CD-ROM 光盘内，然后连同一个批处理文件安装程序一起发布给用户，其中的安装程序只需要把光盘中的文件复制到用户硬盘的目录下即可。因为程序集中包括了所有其依赖的引用程序集和类型，所以用户可以立即运行应用程序，而 CLR 就在应用程序的目录中寻找所引用的程序集。注册表和活动目录(Active Directory)没有必要为应用程序的运行做任何改变。如果要卸载应用程序，简单地删除所有的文件就可以了。

当然，我们也可以用其他的机制，如.cab 文件(典型地用于互联网下载情形来压缩文件以节省下载时间)，来打包和安装程序集文件。我们也可以将程序集文件打包为一个 MSI 文件，来供 Windows 安装器(Windows Installer)服务(MSIExec.exe)使用。MSI 文件允许程序集在 CLR 第一次加载它时才被安装。这并非是 MSI 的一个新功能，它也可以为非托管 EXE 和 DLL 文件提供同样的按需加载功能。

注意　使用批处理文件或者其他简单的“安装软件”可以将应用程序装进用户的机器内；但是我们还需要更强大的安装软件来在用户桌面上创建快捷链接、开始按钮、以及快速启动工具条。另外，虽然我们可以很容易地备份和恢复应用程序，或者将它从一个机器移到另一个机器上，但是各种各样的快捷链接还需要一些特别的处理。将来的 Windows 可能会在这方面有所改进。

和应用程序部署在同一目录下的程序集称作**私有部署程序集**(privately deployed assembly)，因为这些程序集文件不会为任何其他的应用程序所共享(除非该应用程序也部署在同样的目录下)。私有部署程序集为开发人员、用户和管理员带来了很大的便利，因为只要将它们简单地复制到一个应用程序的基目录下，CLR 便能够加载并执行它们。另外，应用程序还可以通过简单地删除目录下的程序集来完成卸载。这也使得备份和恢复工作变得非常简单。这种简单的安装、移动和卸载之所以可能，是因为

每一个程序集都包含着描述自己的一些元数据，这些元数据指出了程序集在运行时需要加载哪些其他的程序集，不再需要设置注册表或活动目录。

另外，引用程序集还准确地界定(scope)了每一个类型。这意味着应用程序总是能够准确地绑定到它生成和测试时的类型。CLR 不可能加载另一个仅仅提供了同名类型的程序集。这和 COM 有所不同，COM 中的类型被记录在注册表中，从而使得它们可以被用于机器内运行的任何应用程序。

第 3 章将讨论如何部署可被多个应用程序访问的共享程序集。

2.7 简单管理控制(配置)

终端用户或管理员可以在很大程度上决定一个应用程序的执行。例如，一个管理员可以决定移动用户硬盘上的程序集文件，或者覆盖(override)包含在程序集清单中的某些信息。另外还存在一些和版本控制与远程传送(remoting)相关的情形，其中的一些内容将会在第 3 章中予以探讨。

为了对一个应用程序进行管理控制，我们可以在应用程序的目录下安放一个配置文件。应用程序的发布者可以创建、打包该文件。安装程序则应该将该配置文件安装在应用程序的基目录下。另外，管理员和终端用户应该能够创建或者修改该文件。CLR 通过解析该文件的内容来改变其定位和加载程序集时的策略。

这些配置文件中包含的都是 XML 数据，可以和一个应用程序、或者一台机器相关联。使用一个单独的文件(而不是注册表设置)使得很容易备份该文件，并且允许管理员将应用程序复制到另外的机器上：只需复制必要的文件，管理策略也会随之被复制。

第 3 章将会深入探讨这个配置文件。但是这里先让我们对它有一个初步的了解。假设应用程序的发布者希望在部署他的应用程序时，能把应用程序中的 JeffTypes 程序集文件部署在和其他程序集文件不同的目录下。比如，他希望的目录结构如下：

```
AppDir 目录（包含着应用程序的程序集文件）
    App.exe
    App.exe.config（下面讨论）

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

因为 JeffTypes 文件已不再位于应用程序基目录下，CLR 将不能定位并加载这些文件。运行应用程序将会抛出 System.IO.FileNotFoundException 异常。要修复这个问题，发布者可以创建一个 XML 配置文件，并把它部署到应用程序的基目录下。该文件的名称必须是应用程序的名称再加一个.config 扩展名，例如 App.exe.config。该配置文件的内容看起来应该像下面这样：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <runtime>
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <probing privatePath="AuxFiles" />
        </assemblyBinding>
    </runtime>
</configuration>
```

当 CLR 试图定位一个程序集文件时，它总是先查看应用程序的基目录，如果在那里找不到该文件，它将会去查看 AuxFiles 子目录。我们可以在 probing 元素的 privatePath 属性(译注：这里指 XML 格式中的属性，即 attribute，而非.NET 框架中的特性)中指定多个以分号隔开的路径。每一个路径都被认为是应用程序基目录的相对路径。我们不能用一个绝对路径或者相对路径来指定一个在应用程序基目录之外的目录。这种约束的含义如下：一个应用程序可以控制它的基目录和子目录，但不能控制其他目录。

另外，我们也可以编写代码来打开并分析包含在配置文件中的信息。这允许我们的应用程序定义一些额外的设置，而管理员和终端用户则可以像对待应用程序中的其他设置一样将它们创建和保存在同样的配置文件中。我们可以使用 System.Configuration 命名空间中的类在运行时操作配置文件。

对于不同的应用程序类型，XML 配置文件的名称和位置也有所不同。

程序集文件的定位

当 CLR 需要定位一个程序集时，它将扫描应用程序的几个子目录。下面是 CLR 扫描一个语言文化中性的程序集时的顺序：

```
AppBase\AsmName.dll
AppBase\AsmName\AsmName.dll
AppBase\privatePath1\AsmName.dll
AppBase\privatePath1\AsmName\AsmName.dll
AppBase\privatePath2\AsmName.dll
AppBase\privatePath2\AsmName\AsmName.dll
……
```

在上面的例子中，如果 JeffTypes 程序集文件被部署在 JeffTypes 子目录下，那么将不需要任何的配置文件，因为 CLR 将自动扫描与正在查找的程序集名称相同的子目录。

如果在前面所有的子目录下都找不到该程序集，CLR 将以.exe 扩展名代替.dll 扩展名来搜索。如果仍然找不到该程序集，将有一个 FileNotFoundException 异常被抛出。

对于卫星程序集来说，除了期望被放在应用程序基目录下、并且名字和其语言文化相匹配的子目录中以外，其他的规则仍然适用。例如，假设 AsmName.dll 应用有一个“en-US”语言文化，CLR 将会扫描以下目录：

```
AppBase\en-US\AsmName.dll
AppBase\en-US\AsmName\AsmName.dll
AppBase\en-US\privatePath1\AsmName.dll
AppBase\en-US\privatePath1\AsmName\AsmName.dll
AppBase\en-US\privatePath2\AsmName.dll
AppBase\en-US\privatePath2\AsmName\AsmName.dll
……
```

同样，如果该程序集不能在上面列出的子目录中找到，CLR 将在相同的程序集集合中以.exe 扩展名代替.dll 扩展名查找该程序集。

- 对于可执行应用程序(EXE)，配置文件必须位于应用程序的某目录中，并且它的名字必须是EXE文件名再加.config扩展名。

- 对于ASP.NET Web窗体和XML Web服务应用程序，配置文件必须位于Web应用程序的虚拟根目录下，并且名称总是Web.config。另外，子目录也可以包含它们自己的Web.config文件，并继承上一目录的配置设置。例如，一个位于*http://www.Wintellect.com/Training*下的Web应用程序将同时使用包含在应用程序虚拟根目录和Training子目录下的Web.config文件中的设置。

- 对于包含客户方控件、以微软的IE浏览器为宿主的程序集，HTML页面必须包含一个链接标记，并将其中的rel属性设置为“Configuration”，而将href属性设置为配置文件的URL，配置文件的名字可以任意。下面是一个例子：<LINK REL=Configuration HREF=http://www.Wintellect.com/Controls.config>。关于这方面的更多信息，可参见.NET框架文档。

如本节开始所提到的，配置设置可以应用于一个特定的应用程序，也可以应用于整个机器。当我们安装.NET框架时，它将为我们创建一个Machine.config文件。机器内安装的每一个版本的CLR都会有一个Machine.config文件。因为在不远的将来，我们很有可能在同一台机器中同时安装多个版本的.NET框架。

Machine.config文件位于下面的目录中：

```
C:\WINDOWS\Microsoft.NET\Framework\version\CONFIG
```

当然，C:\WINDOWS表示我们的Windows目录，而version则是一个版本号，它标识着一个特定版本的.NET框架。

Machine.config文件中的设置会覆盖特定应用程序的配置文件中相应的设置。(译注：这里所说的是不正确的，实际上是特定应用程序的配置文件中的设置会覆盖Machine.config文件中相应的设置。)管理员只需修改一个单独的文件就可以建立起应用于整个机器范围的策略。通常情况下，管理员和用户应该避免直接修改Machine.config文件，因为该文件包含着和许多内容相关的设置，这使得它非常难以浏览。再者，我们还希望能够备份和恢复应用程序的设置，将设置保存在特定应用程序的配置文件中可以实现这一目的。

因为编辑 XML 配置文件有些麻烦，微软的.NET 框架小组开发了一个辅助的 GUI 工具。该 GUI 工具被实现为一个微软管理控制台(Microsoft Management Console，即 MMC)单元，这意味着在运行 Windows 98、Windows 98 第 2 版，或者 Windows Me 的机器上将不能使用该工具。打开【控制面板】，选择【管理工具】，然后选择 Microsoft .NET Framework Configuration 即可找到该工具。在出现的窗口中，我们可以在左边的树形面板上找到【应用程序】节点，如图 2.6 所示。

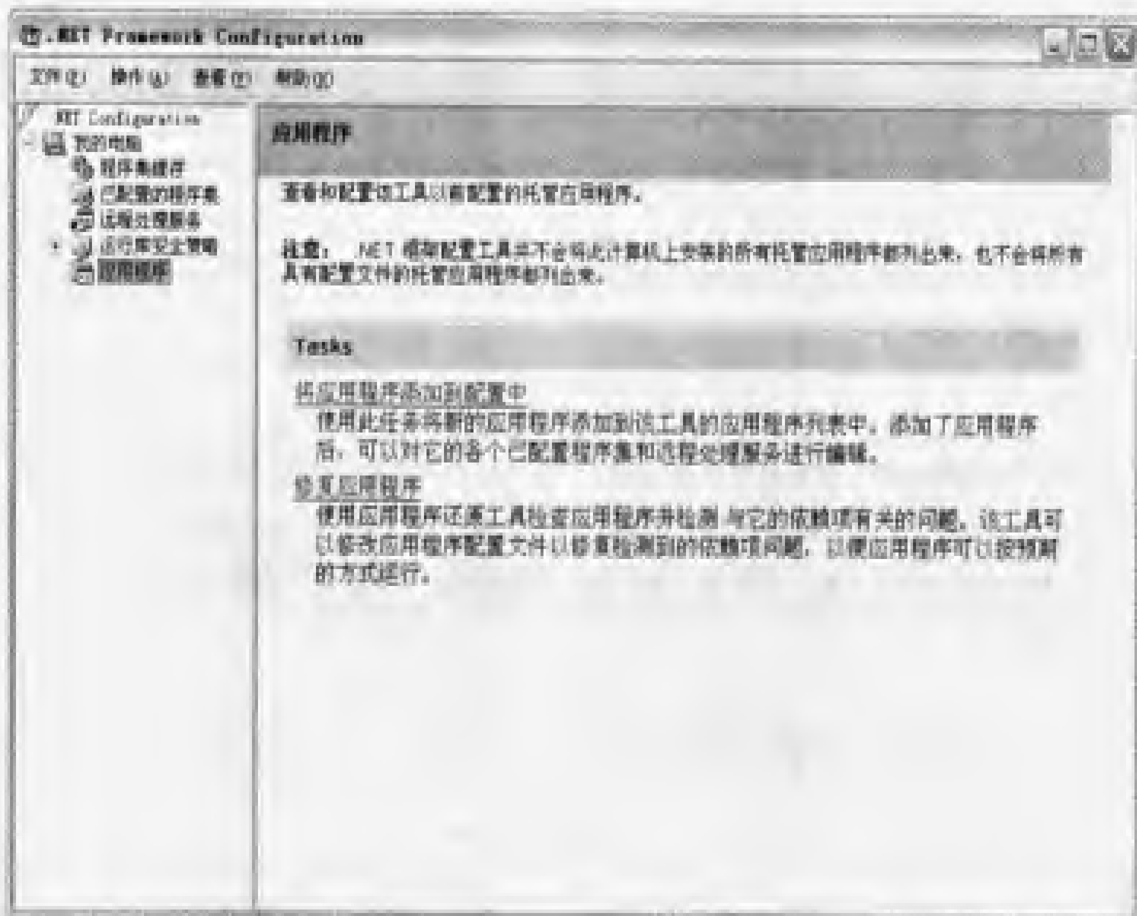


图 2.6　Microsoft .NET Framework Configuration 工具中的【应用程序】节点

由【应用程序】节点，我们可以选择右侧面板上的【将应用程序添加到配置中】链接，这将会调用一个向导程序，提示我们提供希望创建 XML 配置文件的可执行文件路径名。在添加了一个应用程序之后，我们便可以使用该工具来修改它的配置文件。图 2.7 显示了可以在一个应用程序上执行的操作。

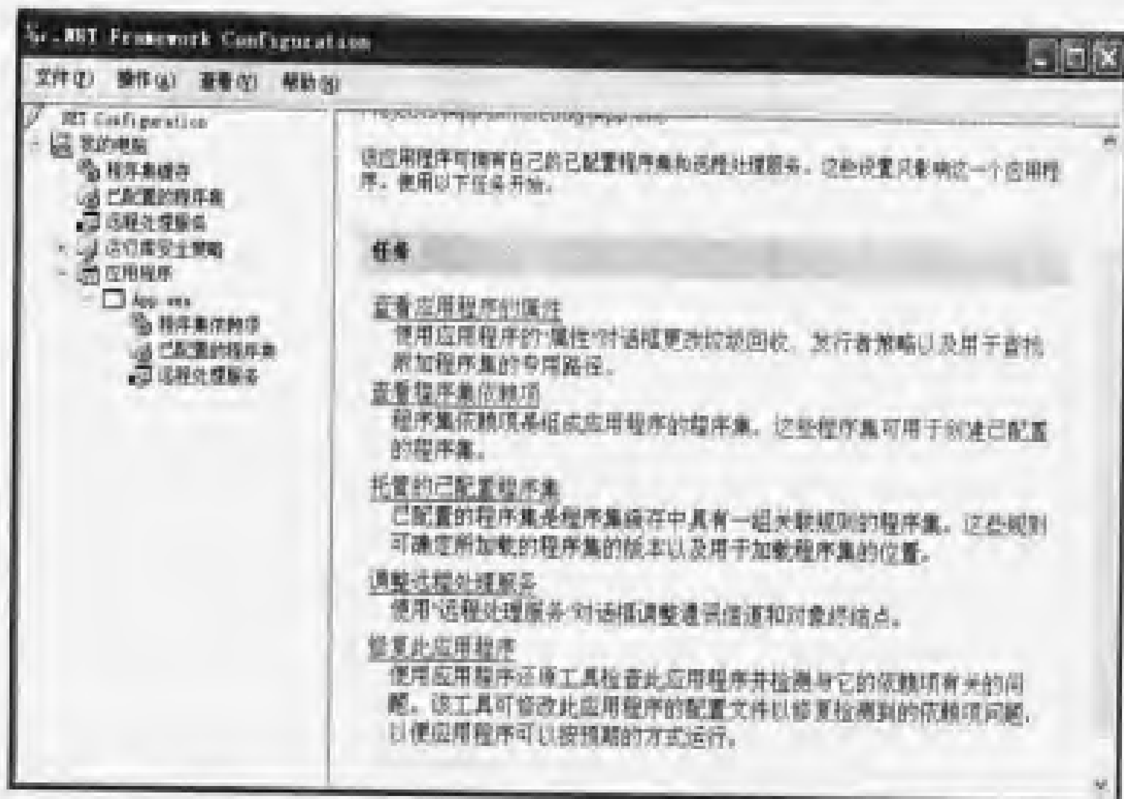


图 2.7 使用 Microsoft .NET Framework Configuration 工具配置应用程序

第3章将讨论配置文件的更多细节。

3

共享程序集

本书第 2 章讨论了生成、打包和部署一个程序集的一些必要步骤，同时我们还对程序集的**私有部署方式**(private deployment，即将程序集放在应用程序的某目录及其子目录中，为应用程序所单独使用的部署方式)做了详细的讨论。私有部署方式使得我们可以方便地控制程序集的命名、版本和行为。

本章我们将探讨创建可以被多个应用程序共同访问的程序集，即**全局部署程序集**(globally deployed assembly)。和微软的.NET 框架一起发布的程序集就是一个典型的例子，因为几乎所有的托管应用程序都使用了微软在.NET 框架类库(FCL)中定义的类型。

如本书第 2 章所述，Windows 有着一个不稳定的坏名声。这个坏名声的主要原因就是使用了其他人编写的代码来生成和测试应用程序。实际上，当我们编写 Windows 应用程序时，我们的应用程序肯定要调用到微软公司的开发人员编写的代码。另外，许多公司都在创建第三方控件，这些控件往往被应用程序开发人员集成到自己的应用程序中。事实上，.NET 框架鼓励这么做，而且随着时间的推移会有越来越多的控件厂商出现。

但是，随着时间的推移，微软和其他一些控件厂商的开发人员会不断地修改他们的代码：修复 bug，增加特性，等等诸如此类的事情。最终，新代码会被安装到用户的硬盘中，用户先前已经安装并且运行正常的应用程序使用的将不再是当初生成和测试时的代码。结果是应用程序的行为变得不可预期，正是这些原因造成了 Windows 的不稳定。

文件版本是一个很难解决的问题。实际上，如果仅仅在一个文件中将其某一位从0改变到1、或者从1改变到0，我们便不能绝对保证使用原来文件的代码和它使用新版文件时的行为一样。这是因为许多应用程序都会有意或者无意地引入bug。如果一个文件的后续版本修复了一个bug，应用程序便不再如预期那样运行。

这就存在一个问题：怎样在修复bug和增加特性的同时，还能保证不会损坏现有的应用程序？我曾经对这个问题思考了很久，并且得出了一个结论——那就是这是不可能的。很明显，这样的回答解决不了问题。应用程序文件总是会携带bug，开发人员也总是想增加新的特性。在对应用程序的正常运行抱有良好预期的同时，还必须采用另一种分发新文件的方式来保证一旦应用程序不能正常运行，我们仍然可以将它们轻易地恢复到最近一次的正常状态。

本章将解释.NET 框架处理这些版本问题的基础构造。首先提醒一下大家，这里描述的一些东西有些复杂。其中会讨论很多内建于CLR的算法、规则以及策略。还会提到一些应用程序开发人员必须使用的许多工具和实用程序。这些内容比较复杂的原因在于版本问题本身就很难处理和解决。

3.1 两种程序集、两种部署方式

.NET 框架支持两种程序集：弱命名程序集(weakly named assembly)和强命名程序集(strongly named assembly)。

> **重要** 顺便说一句，在.NET 框架的任何文档中都找不到“弱命名程序集”这个术语。为什么呢？因为这是我个人命名的。这里决定发明这个术语的目的是能使我们在谈论程序集时，不至于造成某种混淆。

弱命名程序集和强命名程序集在结构上是相同的，也就是说，它们使用同样的PE文件格式、PE表头、CLR表头、元数据，以及清单表。并且我们可以使用同样的工具(例如C#编译器和AL.exe)来生成两种程序集。二者之间的真正区别在于，强命名程序集有一个发布者的公钥/私钥对签名，其中的公钥/私钥对惟一地标识了程序集的发布者。利用公钥/私钥对，我们可以对程序集进行惟一的标识、实施安全策略和版本策略，从而可以将其部署在用户硬盘的任何地方、甚至互联网上。这种惟一标识程序集的能力使得应用程序在试图绑定一个强命名程序集时，CLR能够实施某些“已确知安全”的策略。本章将详细解释强命名程序集的概念，以及CLR对它们实施的策略。

一个程序集有两种部署方式：即私有方式和全局方式。私有部署方式将程序集部署在应用程序的基目录及其子目录下。弱命名程序集只能够进行私有部署。第2章已经介绍了私有部署程序集。全局部署方式将程序集部署在一些CLR确知的地方，当CLR搜索程序集时，它会知道到这些地方去查找。强命名程序集既可以进行私有部署，也可以进行全局部署。本章将解释怎样创建和部署强命名程序集。表3.1总结了程序集的种类，以及它们可用的部署方式。

表3.1 弱命名与强命名程序集的部署方式

程序集的种类	是否可以进行私有部署	是否可以进行全局部署
弱命名程序集	是	否
强命名程序集	是	是

3.2 强命名程序集

如果一个程序集要被多个应用程序访问，那么这个程序集必须被放在一个CLR确知的目录下。当CLR检测到该程序集被引用时，它会自动到这个目录下查找该程序集。

然而，这里有一个问题：两个(或多个)不同的公司可能会生产出有相同名称的程序集来。如果这些同名程序集都被复制到相同的目录下，最后一个安装的程序集将会替代前面的程序集，引用那些程序集的所有应用程序将不能再如期运行。(这实际上就是目前 Windows 中出现 DLL hell 的原因。)

很明显，简单地用文件名来区分程序集是不够的。CLR 需要支持某种机制来惟一地标识一个程序集。这就是所谓的强命名程序集。一个强命名程序集包含四个惟一标识程序集的特性：文件名(没有扩展名)、版本号、语言文化标识、和一个公有密钥标记(由公有密钥产生的一个值)。下面的字符串分别标识了四个不同的程序集文件：

```
"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"
```

其中第一个字符串标识了一个名为 MyTypes.dll 的程序集。它的版本号为 1.0.8123.0，并且没有设定任何特殊的语言文化，因为 Culture 被设为 neutral。当然，任何公司都可能生产一个名为 MyTypes.dll、版本号为 1.0.8123.0，以及语言文化中性的程序集。

所以，必须有一种方式能够将一个公司的程序集和另一个公司的程序集区别开来。由于某些原因，微软选择了标准的公钥/私钥对加密技术(其他标识惟一性的技术有 GUID、URL、URN 等)。这种加密技术使得我们在程序集被安装到用户硬盘上的时候，能够校验其内比特位的完整性，并且还可以根据发布者的身份来授权某些许可。本章稍后将探讨这些技巧。

这样，如果一个公司想惟一地标识它的程序集，那么它必须首先获取一个公钥/私钥对。然后将公有密钥和程序集相关联。不存在两个公司拥有同样的公钥/私钥对的情况，这种区分使得我们能够创建有着相同名称、版本、语言文化的程序集，而不引起任何冲突。

注意 利用 System.Reflection.AssemblyName 类，我们可以很容易地创建一个程序集名称，并获取一个程序集名称的各个部分。该类提供了几个公有实例属性，例如 CultureInfo、FullName、KeyPair、Name，以及 Version。该类还提供了几个公有实例方法，例如 GetPublicKey、GetPublicKeyToken、SetPublicKey，以及 SetPublicKeyToken。

本书第2章向大家介绍了怎样命名一个程序集，以及如何为一个程序集指定版本号和语言文化。一个弱命名程序集可以在其清单元数据中嵌入版本号和语言文化特性，但 CLR 总会忽略版本号，并且只有在搜寻子目录查找卫星程序集的时候才会使用其中的语言文化信息。因为弱命名程序集总是以私有方式部署的，所以 CLR 在程序集的基目录或任何子目录(在 XML 配置文件定位元素的 privatePath 属性中指定)中搜索程序集时，它只需利用该程序集的名称(加上扩展名.dll 或者 exe)就可以了。

一个强命名程序集包含有一个文件名、一个版本号以及一个语言文化信息。另外，强命名程序集还有一个发布者的私有密钥签名。

创建一个强命名程序集首先需要获取一个用强命名实用工具(Strong Name Utility，即 SN.exe，和.NET 框架 SDK，以及 Visual Studio .NET 一起发布的一个工具)产生的密钥。该实用工具提供了一整套的功能，我们可以根据命令行开关来指定它们。注意 SN.exe 的所有命令行开关都是区分大小写的。要产生一个公钥/私钥对，我们可以象下面这样运行 SN.exe 实用工具：

```
SN -k MyCompany.keys
```

该命令告诉 SN.exe 创建一个名为 MyCompany.keys 的文件。MyCompany.keys 文件将包含一对以二进制格式存储的公有密钥和私有密钥。

公有密钥非常大。我们可以执行下面的命令来查看它们：

```
SN -tp MyCompany.keys
```

(译注：上述查看公有密钥的方法是错误的，正确的做法如下：首先，以-p 命令行开关调用 SN.exe 创建一个只包含公有密钥的文件：

```
SN -p MyCompany.keys MyCompany.PublicKey
```

然后以-tp 命令行开关调用 SN.exe，并将前面得到的包含公有密钥的文件传递给它：

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

另外，下文的输出也是按正确的做法得到的结果。)

执行上面的命令将得到以下输出：

```
Microsoft (R) .NET 框架强名称实用工具版本 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.


公钥为

0024000004800000940000000602000000240000525341310004000001000100c7dec4f75afed2
6acfe15647b971934la5efd36d3971c1f95262e9dc6af1f1751e3f71451aeb0615adf8109167a6
d95c773b32fec01634d5c1c5df66d498a97407fb743a8da28d6e12a68e8e3b2225977628d26d78
dbc6889eeb6e9cf9e5f6a2b9e26ff378e01d4716af6af995731d7d5d86dc2ce36385445b8d9fdf
79f086a7

公钥标记为 fc2f0bde961055da
```

公有密钥的尺寸使得它们很难使用。为了方便开发人员(以及终端用户)，**公有密钥标记**(public key token)应运而生。公有密钥标记是一个 64 位的公有密钥散列值。SN.exe 的-tp 命令行开关在输出的末尾显示了和完整的公有密钥相对应的公有密钥标记。

既然已经知道了怎样创建一个公钥/私钥对，创建强命名程序集就变得很容易了。只需把 System.Reflection.AssemblyKeyFileAttribute 特性的一个实例应用到我们的源代码中就可以了：

```
[assembly:AssemblyKeyFile("MyCompany.keys")]
```

当编译器在源代码中遇到该特性时，编译器将打开其中指定的文件(MyCompany.keys)，用私有密钥对程序集进行签名，并将公有密钥嵌入到清单中。注意只能对包含清单的那个程序集文件进行签名，程序集的其他文件不能被显式地签名。

下面是对程序集文件签名过程的一个详细解释：当生成一个强命名程序集时，该程序集的 FileDef 清单元数据表将包含组成该程序集的所有文件的一个列表。当每个文件的名称被加入到清单中时，该文件的内容也被转换成一个散列值，该散列值将和文件名一起存入 FileDef 表中。

我们可以用 AL.exe 的/algid 命令行开关，或者应用于程序集上的 System.Reflection.AssemblyAlgorithmIdAttribute 定制特性来改变默认的散列算法。默认的算法为 SHA-1 算法(译注：SHA 即 Secure Hash Algorithm，安全散列算法)，这对绝大多数应用程序已经足够了。

在生成包含清单的 PE 文件之后，该 PE 文件的整个内容都被转换为一个散列值，如图 3.1 所示。这里使用的散列算法总是 SHA-1，而且不能被改变。该散列值——尺寸一般在 100 和 200 个字节之间——经由发布者的私有密钥签名，生成的 RSA 数字签名被存储在 PE 文件的一个保留区域(不包括在散列值的计算中)。最后，PE 文件的 CLR 表头被更新以反映数字签名在文件中的嵌入位置。

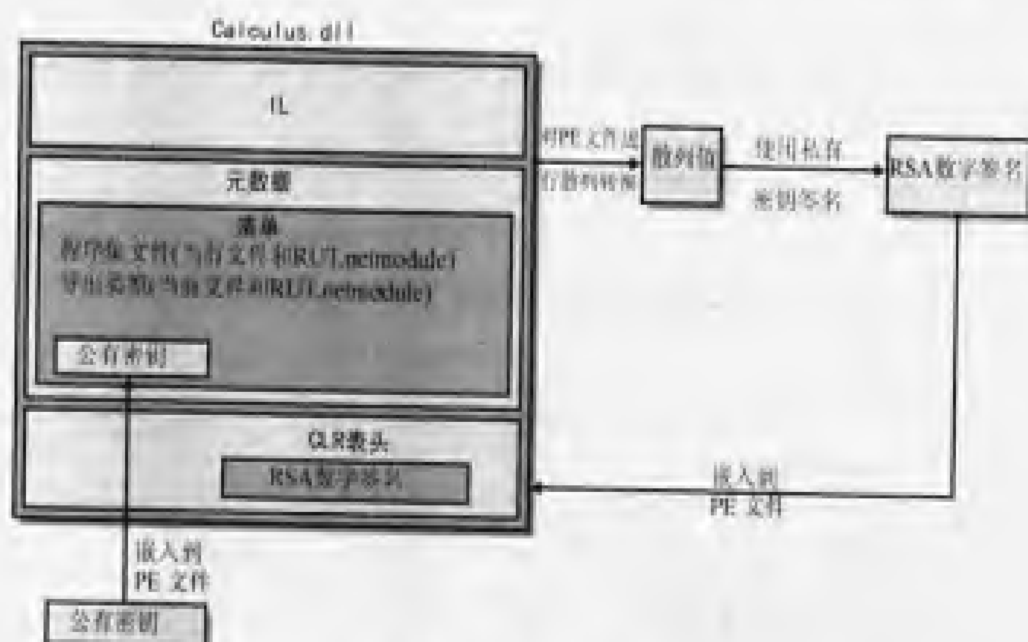


图 3.1　签名一个程序集

另外，发布者的公有密钥也被嵌入到 PE 文件的 AssemblyDef 清单元数据表中。文件名、版本号、语言文化、以及公有密钥四者合起来组成了一个程序集的强命名，这保证了程序集的惟一性。因为两个公司不可能产生有着同样公有密钥的"Calculus"程序集(这里的前提假设是两个公司不会彼此分享他们的密钥对)。

到这里，程序集及其所有的文件就可以被打包并发布了。

如第 2 章所述，当我们编译源代码时，编译器会检测代码中引用的类型和成员，我们必须为编译器指定所引用的程序集。对于 C#编译器来说，可以使用/reference 命令行开关来实现这一点。

编译器的部分工作就是在生成的托管模块中嵌入一个 AssemblyRef 元数据表。AssemblyRef 元数据表中的每一个条目都标识着一个被引用程序集的名称(不含路径和扩展名)、版本号、语言文化以及公有密钥信息。

> **重要**　因为公有密钥是如此之大，所以如果一个程序集引用了许多其他程序集，那么它的最终文件的很大一部分将被公有密钥信息占去。为了节省存储空间，微软对公有密钥首先进行散列转换，然后只取得到的散列值的最末尾的 8 个字节。这个被截取的值在统计学上被认为是惟一的，因此可以被安全地传给系统。存储在 AssemblyRef 表中的正是这些被截取后的公有密钥值(又称公有密钥标记)。一般而言，开发人员和终端用户看到的通常是公有密钥标记，而非完整的公有密钥值。

下面是第 2 章中讨论过的 JeffTypes.dll 文件中的 AssemblyRef 元数据信息：

```
AssemblyRef #1
-------------------------------------------------------
    Token: 0x23000001
    Public Key or Token: b7 7a 5c 56 19 34 e0 89
    Name: mscorlib
    Major Version: 0x00000001
    Minor Version: 0x00000000
    Build Number: 0x00000ce4
    Revision Number: 0x00000000
    Locale: <null>
    HashValue Blob: 3e 10 f3 95e3 73 0b 33 1a 4a 84 a7 81 76 eb 32 4b
        36 4d a5
    Flags: [none] (00000000)
```

其中我们可以看到 JeffTypes.dll 引用的一个类型包含在有着下面特性的程序集中：

```
"MSCorLib,Version=1.0.3300.0,Culture=neutral,PublicKeyToken=b77a5c561934e089"
```

不幸的是，ILDasm.exe 在本该使用 Culture 的地方使用了术语 Locale。微软许诺他们将在该工具的下一个版本中修正这个术语。

我们再来看一下 JeffTypes.dll 中 AssemblyDef 元数据表的内容：

```
Assembly
-------------------------------------------------------
    Token: 0x20000001
    Name : JeffTypes
    Public Key :
    Hash Algorithm : 0x00008004
    Major Version: 0x00000001
    Minor Version: 0x00000000
    Build Number: 0x00000253
    Revision Number: 0x00005361
    Locale: <null>
    Flags : [SideBySideCompatible] (00000000)
```

这和下面的信息是等同的：

```
"JeffTypes,Version=1.0.595.21345,Culture=neutral,PublicKeyToken=null"
```

大家可以看到其中没有指定公有密钥标记，这是因为第 2 章中的 JeffTypes.dll 没有公有密钥签名，这使它成为一个弱命名程序集。

如果我们使用 SN.exe 创建一个密钥文件，并将 AssemblyKeyFileAttribute 特性添加到源代码中，然后重新编译，得到的程序集将是经过签名后的程序集。如果使用 AL.exe 来生成程序集，我们可以指定/keyfile 命令行开关来代替使用 AssemblyKeyFileAttribute 特性。如果再使用 ILDasm.exe 来查看新生成的程序集元数据，我们可以看到 AssemblyDef 条目中的“Public Key”后面将有一些字节，这表示程序集为强命名程序集。顺便说一句，AssemblyDef 条目总是存储着整个公有密钥，而不是公有密钥标记。完整的公有密钥用来确保程序集文件不会被篡改。本章后面将解释强命名程序集的防篡改特性。

3.3 全局程序集缓存

了解了怎样创建一个强命名程序集后，我们现在来学习如何部署这样的程序集，以及 CLR 怎样使用一些信息来定位和加载这样的程序集。

如果一个程序集被多个应用程序所访问，那么该程序集必须被放在一个 CLR 已确知的目录下，并且 CLR 在探测到有对该程序集的引用时，它必须能够自动到该目录下寻找这个程序集。这个已确知的目录称作**全局程序集缓存**(Global Assembly Cache，即 GAC)，它通常位于下面的目录中：

```
C:\Windows\Assembly\GAC
```

GAC是一个结构化的目录，它包含许多子目录，这些子目录的名称是用一种算法来产生的。我们不应该手动将程序集文件拷贝到GAC中，相反，我们应该使用工具来完成这项工作。因为这些工具知道GAC的内部结构，以及怎样产生正确的子目录名称。

在做开发和测试时，向GAC中安装一个强命名程序集最常用的工具就是GACUtil.exe。不带任何命令行开关运行该工具将产生下面的使用方法提示：

```
Microsoft (R) .NET Global Assembly Cache Utility.  Version 1.0.3415.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Usage: Gacutil <option> [<parameters>]
 Options:
  /i
    Installs an assembly to the global assembly cache.  Include the
    name of the file containing the manifest as a parameter.
    Example:  /i myDll.dll

  /if
    Installs an assembly to the global assembly cache and forces
    overwrite if assembly already exists in cache.  Include the
    name of the file containing the manifest as a parameter.
    Example:  /if myDll.dll

  /ir
    Installs an assembly to the global assembly cache with traced
    reference. Include the name of file containing manifest,
    reference scheme, ID and description as parameters
    Example:  /ir myDll.dll FILEPATH c:\apps\myapp.exe MyApp

  /u[ngen]
    Uninstalls an assembly. Include the name of the assembly to
    remove as a parameter. If ngen is specified, the assembly is
    removed from the cache of ngen'd files, otherwise the assembly
    is removed from the global assembly cache
    Examples:.
      /ungen myDll
      /u myDll,Version=1.1.0.0,Culture=en,PublicKeyToken=874e23ab874e23ab
```

```
/ur
  Uninstalls an assembly reference. Include the name of the
  assembly, type of reference, ID and data as parameters.
  Example:  /ur myDll,Version=1.1.0.0,Culture=en,
                PublicKeyToken=874e23ab874e23ab
                FILEPATH c:\apps\myapp.exe MyApp

/uf
  Forces uninstall of an assembly by removing all install references
  Include the full name of the assembly to remove as a parameter..
  Assembly will be removed unless referenced by Windows Installer.
  Example:  /uf myDll,Version=1.1.0.0,Culture=en,
                PublicKeyToken=874e23ab874e23ab

/l
  Lists the contents of the global assembly cache. Allows optional
  assembly name parameter to list matching assemblies only

/lr
  Lists the contents of the global assembly cache with traced
  reference information. Allows optional assembly name parameter
  to list matching assemblies only

/cdl
  Deletes the contents of the download cache

/ldl
  Lists the contents of the downloaded files cache

/nologo
  Suppresses display of the logo banner

/silent
  Suppresses display of all output
```

我们可以调用 GACUtil.exe 并指定/i 命令行开关来将一个程序集安装到 GAC 中。也可以用 GACUtil.exe 的/u 命令行开关来将一个程序集从 GAC 中卸载掉。注意不能将一个弱命名程序集安装到 GAC 中。如果向 GACUtil.exe 传递的是一个弱命名程序集，它将显示如下错误信息：

"Failure adding assembly to the cache: Attempt to install an assembly without a strong name."（程序集添加失败：试图安装一个非强命名的程序集。）

注意 默认情况下，GAC只能被属于Windows管理员组的用户来操作。如果调用GACUtil.exe的用户不是该组的成员，操作将会失败。

GACUtil.exe 的/i 命令行开关非常便于开发人员在测试环境中使用。但是如果是在实际的生产环境中，推荐使用/ir 命令行开关来部署程序集，/ur 命令行开关来卸载程序集。命令行开关/ir 中集成了Windows 的安装和卸载引擎，它告诉系统哪个应用程序需要安装程序集，并将该应用程序和程序集关联在一起。

注意 如果一个强命名程序集以.cab文件的形式打包或者经过了某种方式的压缩，那么在使用GACUtil.exe来将该程序集文件安装到GAC中时，必须首先把它们解压为临时文件，待完成这些程序集文件的安装之后，就可以把临时文件删去。

GACUtil.exe 工具没有和面向终端用户的.NET 框架分发包一起发布。如果应用程序包括一些需要部署到 GAC 中的程序集，那么必须使用 2.0 版本以上的 Windows 安装器(MSI)，因为它能够将程序集安装到 GAC 中，并且终端用户的机器上总是存在该工具。(可以通过运行 MSIExec.exe 来确定Windows 安装器的版本。)

重要 将程序集文件部署到GAC中仍是一种注册程序集的形式，虽然实际上Windows注册表并没有受到任何影响。将程序集安装到GAC中破坏了简化应用程序安装、备份、恢复、移动和卸载的目标。只有当我们放弃全局部署而采用私有部署时，我们才能获得这些简化的目标。

那么在GAC中"注册"一个程序集的意图是什么呢？很简单，假设两个公司都生产了一个Calculus程序集，其中仅包含一个文件Calculus.dll。显然，这两个文件不可能放在同一个目录下，因为最后一个被安装的会覆盖前面的一个，这必定会破坏一些应用程序。当我们使用一个工具将程序集安装到GAC中时，该工具会在C:\Windows\Assembly\GAC目录下创建子目录，并将程序集文件复制到该子目录中。

通常情况下，不会有人去检查GAC的子目录，所以GAC的结构应该和我们没有多少关系。只要相关的工具和CLR知道它的结构就可以了。处于娱乐的目的，下一节会对GAC的内部结构做一番描述。

当安装.NET框架时，安装程序会同时安装一个【资源管理器】shell扩展程序(shFusion.dll)。该shell扩展程序知道GAC的结构，并且能够以一种漂亮、友好的方式来显示GAC中的内容。打开【资源管理器】到C:\Windows\Assembly目录下，我们将会看到如图3.2显示的安装到GAC中的程序集。其中每一行显示了程序集的名称、类型、版本号、语言文化(如果有的话)，以及公有密钥标记。

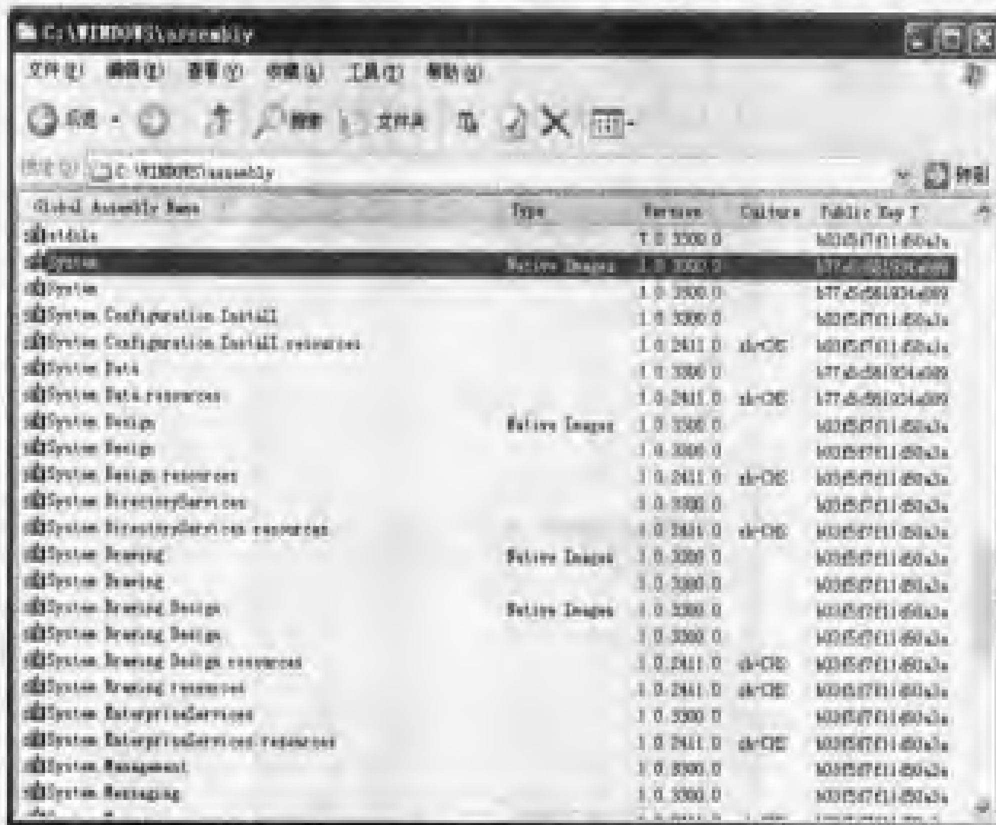


图3.2 使用【资源管理器】的shell扩展查看安装在GAC中的程序集

我们可以选择其中一个条目，并点击鼠标右键得到一个上下文菜单。该上下文菜单包含了【删除】和【属性】两个菜单项。显然，【删除】菜单项将从GAC中删除被选择的程序集文件，并且会修复GAC的内部结构。而【属性】菜单项将显示一个如图3.3所示的属性对话框。其中【上次修改时间】显示了程序集被加到GAC中的时间。选择【版本】选项卡将得到如图3.4所示的结果。

图 3.3 【System 属性】对话框的【常规】选项卡

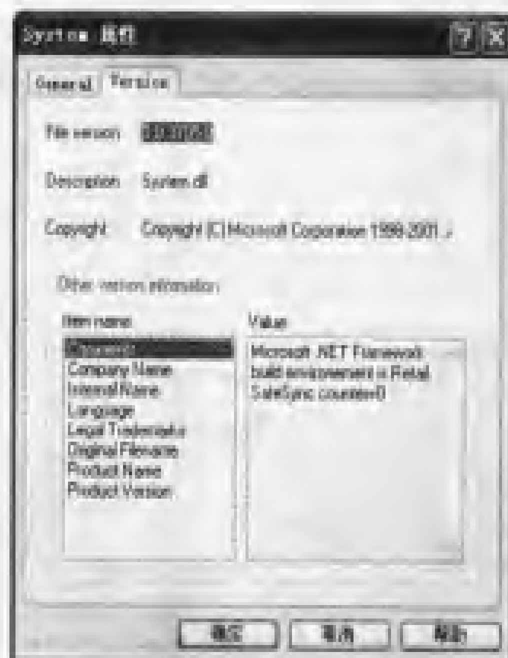


图 3.4 【System 属性】对话框的【版本】选项卡

另外，我们还可以将包含清单的程序集文件拖放到【资源管理器】窗口中，这时，shell 扩展程序会将该程序集的所有文件安装到 GAC 中。对于某些开发人员来说，这是替代 GACUtil.exe 工具将程序集安装到 GAC 中进行测试的一个更为方便的办法。

3.3.1 GAC的内部结构

简而言之，GAC 的目的就是为在强命名程序集和子目录之间维持一个关系。CLR 有一个内部函数，它接受程序集的名称、版本、语言文化以及公有密钥标记四个参数，返回指定程序集文件所在子目录的路径。

如果使用命令行提示符将当前路径改变到 C:\Windows\Assembly\GAC 目录下，我们将会看到几个子目录，每个子目录对应于安装在 GAC 中的一个程序集。下面是 GAC 目录的一个示例(其中为了节省篇幅删除了某些目录):

```
Volume in drive C has no label.
 Volume Serial Number is 94FA-5DE7

 Directory of C:\WINDOWS\assembly\GAC

07/15/2002   05:07 PM    <DIR>         .
07/15/2002   05:07 PM    <DIR>         ..
07/15/2002   03:09 PM    <DIR>         Accessibility
07/15/2002   05:06 PM    <DIR>         ADODB
07/03/2002   04:54 PM    <DIR>         CRVsPackageLib
07/15/2002   05:06 PM    <DIR>         Microsoft.ComCtl2
07/15/2002   05:06 PM    <DIR>         Microsoft.ComctlLib
07/15/2002   05:05 PM    <DIR>         Microsoft.JScript
07/15/2002   05:07 PM    <DIR>         Microsoft.mshtml
07/15/2002   05:06 PM    <DIR>         Microsoft.MSMAPI
07/15/2002   05:06 PM    <DIR>         Microsoft.MSMask
07/15/2002   05:07 PM    <DIR>         Microsoft.MSRDC
07/15/2002   05:07 PM    <DIR>         Microsoft.MSWinsockLib
07/15/2002   05:07 PM    <DIR>         Microsoft.MSWLess
07/15/2002   05:07 PM    <DIR>         Microsoft.PicClip
07/15/2002   05:07 PM    <DIR>         Microsoft.RichTextLib
07/15/2002   05:06 PM    <DIR>         Microsoft.StdFormat
07/15/2002   05:07 PM    <DIR>         Microsoft.SysInfoLib
07/15/2002   05:07 PM    <DIR>         Microsoft.TabDlg
07/15/2002   05:05 PM    <DIR>         Microsoft.VisualBasic
07/15/2002   03:09 PM    <DIR>         System
07/15/2002   03:09 PM    <DIR>         System.Configuration.Install
07/15/2002   03:09 PM    <DIR>         System.Data
07/15/2002   03:09 PM    <DIR>         System.Design
07/15/2002   03:09 PM    <DIR>         System.DirectoryServices
07/15/2002   03:09 PM    <DIR>         System.Drawing
07/15/2002   03:09 PM    <DIR>         System.Drawing.Design
07/15/2002   03:08 PM    <DIR>         System.EnterpriseServices
```

```
07/15/2002    03:09 PM     <DIR>          System.Management
07/15/2002    03:09 PM     <DIR>          System.Messaging
07/15/2002    03:09 PM     <DIR>          System.Runtime.Remoting
07/15/2002    03:08 PM     <DIR>          System.Security
07/15/2002    03:09 PM     <DIR>          System.ServiceProcess
07/15/2002    03:09 PM     <DIR>          System.Web
07/15/2002    03:09 PM     <DIR>          System.Web.RegularExpressions
07/15/2002    03:09 PM     <DIR>          System.Web.Services
07/15/2002    03:09 PM     <DIR>          System.Windows.Forms
07/15/2002    03:09 PM     <DIR>          System.Xml
                  0 File(s)              0 bytes
                 95 Dir(s)  14,798,938,112 bytes free
```

如果进入其中的一个目录，我们会发现其中又有一个或多个子目录。下面是进入 System 子目录的示例：

```
Volume in drive C has no label.
 Volume Serial Number is 94FA-5DE7

 Directory of C:\WINDOWS\assembly\GAC\System

07/15/2002    03:09 PM     <DIR>          .
07/15/2002    03:09 PM     <DIR>          ..
07/15/2002    03:09 PM     <DIR>          1.0.3300.0__b77a5c561934e089
                 0 File(s)              0 bytes
                 3 Dir(s) 14,798,929,920 bytes free
```

机器内每一个安装到 GAC 的 System.dll 程序集在 System 目录中都有一个子目录。上面的示例中只有一个版本的 System.dll 程序集：

```
"System,Version=1.0.3300.0,Culture=neutral,PublicKeyToken=b77a5c561934e089"
```

这些特性以下划线分割的形式呈现为“(Version)_(Culture)_(PublicKeyToken)”。示例中没有语言文化信息，表示 System 程序集的语言文化为中性。在子目录内部是构成强命名 System 程序集的所有文件(即 System.dll)。

重要 显然，GAC的整个目的就是存放一个程序集的多个版本。例如，GAC可以同时包含Calculus.dll的1.0.0.0版本和2.0.0.0版本。如果一个应用程序在生成和测试的时候用的是版本为1.0.0.0的Calculus.dll，那么CLR在执行该应用程序的时候加载的将是1.0.0.0版本的Calculus.dll，即使GAC中装有该程序集的后续版本。这是CLR在加载多版本程序集时的默认策略，这种策略的好处是安装新版的程序集不会影响已经安装的应用程序。但是我们自己可以通过几种方式来改变这种策略，这将在本章后面予以介绍。

3.4　引用强命名程序集

无论我们何时创建一个程序集，它总会引用到其他的强命名程序集，这是因为 System.Object 就定义在 MSCorLib.dll 程序集中。我们创建的程序集也可能会引用到其他强命名程序集中的类型，这些程序集可能是由微软、第三方厂商或者我们自己发布的。

本书第 2 章向大家展示了怎样用 CSC.exe 的/reference 命令行开关来指定期望引用的程序集文件名。如果该文件名为一个完整的路径，CSC.exe 将会加载指定的文件并利用其中的元数据信息来生成程序集。如果指定的是一个不带路径的文件名，CSC.exe 将在以下目录中查找程序集(按序进行)：

1. 当前工作目录。
2. 编译器目前使用的CLR所在的目录。MSCorLib.dll总是包含在该目录中。该目录一般为类似于下面的路径：

```
C:\WINDOWS\Microsoft.NET\Framework\v1.0.3427
```

3. 任何用CSC.exe的/lib命令行开关指定的目录。
4. 任何LIB环境变量中指定的目录。

所以如果我们正在创建一个引用 System.Drawing.dll 的程序集，我们就可以在使用 CSC.exe 的时候为其指定/reference:System.Drawing.dll 命令行开关。编译器将检查前面列出的目录，并且将会在包含目前编译器使用的 CLR 所在的目录中找到 System.Drawing.dll 文件。注意，虽然该目录是编译时发现程序集的地方，但它并不是运行时加载程序集的位置。

我们可以看到在安装.NET 框架时，会有两份微软的程序集文件拷贝被安装。其中一份被装在 CLR 所在的目录中，另一份被装在 GAC 目录中。CLR 所在目录中的拷贝使得我们能够方便地生成自己的程序集。而 GAC 中的拷贝用于运行时加载这些程序集文件。

CSC.exe 不在 GAC 中查找所引用的程序集的原因是我们必须为程序集文件指定一个很长而且很难看的路径，例如 C:\WINDOWS\Assembly\GAC\System.Drawing\ 1.0.3300.0__b03f5f7f11d50a3a\System.Drawing.dll。当然我们可以想到采取另外一种替换的方案，使得 CSC.exe 允许我们指定一个虽然很长、但看起来舒服一些的字符串来引用 GAC 中的程序集，例如 “System.Drawing, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a”。但最后的结果是，这两种方案都被认为要比在用户的硬盘上安装两份程序集文件来得更糟。

在结束本节之前，让我们来讨论一下**响应文件**(response file)。响应文件是一个包含一组编译器命令行开关的文本文件。当执行 CSC.exe 命令时，编译器会打开响应文件，并且就像使用通过命令行传递的开关一样使用响应文件中指定的命令行开关。我们可以在命令行中指定以符号@开头、后面紧跟响应文件名称的形式来告诉编译器使用响应文件。例如，我们可能有一个包含下列文本的响应文件 MyProject.rsp：

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

为了使 CSC.exe 利用这些设置，我们可以象下面这样来调用它：

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

这将告诉 C#编译器输出文件的名称和要创建的目标文件的类型。如我们所见，响应文件非常方便，因为我们不必再每次编译项目时都用手动来表达期望的命令行参数。

C#编译器支持多个响应文件。除了我们显式传递给 CSC.exe 的响应文件外，编译器还会自动搜索名为 CSC.rsp 的响应文件。当我们运行 CSC.exe 时，它会在当前目录中搜索一个本地的 CSC.rsp 文件，我们应该将专用于当前项目的设置放在该文件中。另外，编译器还会在 CSC.exe 所在的目录中搜索一个全局的 CSC.rsp 文件，我们应该将应用于所有项目的设置放在该文件中。编译器最后会使用所有响应文件中指定的设置。如果本地响应文件和全局响应文件中的设置有冲突，本地响应文件中的设置将覆盖全局响应文件中的设置。类似地，如果存在冲突，显式传递给 CSC.exe 的设置也将覆盖本地响应文件中的设置。

当我们安装.NET 框架时，它会同时安装一个默认的全局 CSC.rsp 文件。该文件包括以下命令行开关：

```
# 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.Vsa.dll
/r:System.Configuration.Install.dll
/r:System.Data.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.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceProcess.dll
/r:System.Web.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.Dll
/r:System.XML.dll
```

当编译一个项目时，编译器假设我们希望引用以上列出的所有程序集文件。不过不必担心，只有当我们的源代码中明确引用了一个程序集中的类型或成员时，编译器才会在生成的模块中创建一个和该程序集对应的 AssemblyRef 条目。响应文件对于开发人员来说非常方便，因为它允许我们使用微软发布的各种程序集中的类型和命名空间，同时在编译时又不必指定任何的/reference 命令行开关。当然，如果需要，我们也可以向全局 CSC.rsp 文件中添加自己的命令行开关。

注意 我们可以通过指定/noconfig 命令行开关来告诉编译器忽略所有的本地和全局 CSC.rsp 文件。

3.5 强命名程序集的防篡改特性

用私有密钥为程序集签名可以确保该程序集的生产者为对应公有密钥的持有者。当程序集被安装到 GAC 目录内时，系统将会对包含清单的文件内容进行散列转换，并用得到的散列值来和嵌入在 PE 文件中的 RSA 数字签名进行比较(在用公有密钥“反签名”之后)。如果两个值相同，则证明程序集文件的内容没有被篡改，并且还可以知道我们拥有着和发布者的私有密钥相对应的公有密钥。另外，系统还会对程序集中其他文件的内容进行散列转换，然后将得到的散列值和清单文件中 FileDef 表内存储的散列值进行比较。如何发现有任何不匹配的情况，则证明至少有一个程序集的文件被篡改了，程序集向 GAC 中的安装将告失败。

重要 这种机制只可以确保文件的内容不会被篡改。除非我们确知所持有的公有密钥属于特定的发布者，并且相信该发布者的私有密钥从未泄漏，否则这种强命名机制就无法用于确认程序集发布者的身份。如果发布者希望能够将自己的身份和程序集联系起来，还必须另外采用一种微软的认证码(Authenticode)技术。

当应用程序需要绑定一个程序集时，CLR 将使用所引用的程序集的一些属性(名称、版本、语言文化、以及公有密钥)来在 GAC 中定位程序集。如果找到了被引用的程序集，它所在的子目录将被返回，保存清单的那个文件将会被加载。以这种方式来查找程序集确保了运行时加载的程序集和编译时生成的程序集总是来自于同一发布者，这是因为引用程序集的 AssemblyRef 表中的公有密钥标记和被引用程序集的 AssemblyDef 表中的公有密钥标记是一致的。如果被引用的程序集不在 GAC 中，CLR 将在应用程序的基目录中查找。如果还找不到，CLR 将会到应用程序的配置文件中标识的私有路径中查找。如果应用程序是使用 MSI 来安装的，CLR 还会要求 MSI 来定位程序集。如果在以上所有位置中都找不到程序集，绑定将告失败，系统将会抛出一个 System.IO.FileNotFoundException 异常。

当强命名程序集文件是从一个非 GAC 的地方(比如配置文件中 codeBase 元素指示的位置)加载时，CLR 会在程序集被加载的时候比较散列值。换句话说，每次执行应用程序都会计算一次程序集文件的散列值。为了确保程序集文件的内容不会被篡改，这点性能损失是必要的。当 CLR 在运行时检测到不匹配的散列值时，将会抛出 System.IO.FileLoadException 异常。

3.6 延迟签名

本章早先曾经讨论过使用 SN.exe 工具产生公钥/私钥对的方法。该工具通过调用 Windows 提供的 Crypto API 来产生密钥。这些密钥可以被保存在文件、或者其他一些存储设备中。例如，一些大的组织(比如微软)可能会在某些被锁定安全的硬件设备中保存返回的私有密钥，并且只允许公司中很少的人拥有对这些私有密钥的访问权。这种防范措施可以防止破坏私有密钥，并且也保证了密钥的完整性。而公有密钥则是可以公开并允许自由发布的。

当我们准备打包强命名程序集时，我们必须使用安全的私有密钥来为之签名。然而在开发和测试程序集时，允许访问安全的私有密钥可能会导致私钥泄漏。因此，.NET 框架支持一种称作**延迟签名**(delayed signing)的技术，有时候也叫**局部签名**(partial signing)。

延迟签名允许我们只使用公司的公有密钥就可以生成程序集，不再需要私有密钥。如果某些程序集引用了我们的程序集，使用公有密钥允许它们将正确的公有密钥嵌入到 AssemblyRef 元数据条目中。除此之外，用公有密钥生成的程序集还被允许放在 GAC 的内部结构中。如果没有用私有密钥对这些文件签名，它们将失去篡改保护功能，因为程序集文件没有经过散列转换，也没有在文件中嵌入数字签名。但是，失去这种保护不应该成为一个问题，因为我们是在开发程序集的时候，而不是在打包和部署的时候采用这种延迟签名的。

要实施延迟签名，我们通常应将公司的公有密钥值存放在一个文件中，然后将该文件名传递给程序集的生成工具。(可以使用 SN.exe 的-p 命令行开关来从包含公钥/私钥对的文件中提取公有密钥。) 我们还必须告诉生成工具希望对程序集进行延迟签名，也就是说我们不再提供私有密钥。这可以通过在源代码中应用 AssemblyKeyFileAttribute 和 DelaySignAttribute(译注：这里的 DelaySignAttribute 应该为 AssemblyDelaySignAttribute)两个特性来完成。如果使用 AL.exe 工具，还可以在命令行中指定/keyf[ile]和/delay[sign]开关来实现这一点。

当编译器或者 AL.exe 检测到我们正在对程序集进行延迟签名时，它会为该程序集产生一个 AssemblyDef 清单条目并将程序集的公有密钥包含在其中。另外，公有密钥的存在也允许我们将程序集放在 GAC 中。我们还可以生成引用这些程序集的其他程序集，引用程序集将在其 AssemblyRef 元数据表的条目中包含正确的公有密钥。当生成最终的程序集时，得到的 PE 文件中将会为 RSA 数字签名留出一定的空间(一些实用工具可以根据公有密钥的大小中判断出需要的空间)。注意程序集文件的内容现在还没有经过散列转换。

现在，生成的程序集还没有一个有效的签名。这时候试图直接向 GAC 中安装程序集将会失败，因为文件内容还没有经过散列转换——文件看上去好像遭到了篡改。为了将程序集安装到 GAC 中，我们必须阻止系统对程序集进行的完整性验证。这可以通过为 SN.exe 工具指定-Vr 命令行开关来实现。该命令行开关还会告诉 CLR 在运行时加载程序集的时候，跳过对其内任何文件的散列值的检查。

一旦完成了开发和测试，我们就需要正式地为程序集进行签名以实施打包和部署。为了完成这一点，我们还需要再次使用SN.exe实用工具，这一次使用的是命令行开关-R和包含实际私有密钥的文件名(译注：SN.exe没有提供获取一个公钥/私钥对中私有密钥的方式，这里应该为包含公钥/私钥对的文件名)。命令行开关-R会使SN.exe对文件的内容进行散列转换，并以私有密钥进行签名，然后将RSA数字签名嵌入到文件中原来预留出的空间。完成上面的步骤后，我们就可以部署经过完全签名的程序集了。我们还可以使用SN.exe的-Vu或-Vx命令行开关来恢复对程序集的验证过程。

下面总结了本节对使用延迟签名技巧来开发程序集所做的讨论：

1. 当开发程序集时，首先取得仅包含公司公有密钥的文件，并将下面两个特性加到源代码中：

```
[assembly:AssemblyKeyFile("MyCompanyPublicKey.keys")]
//(译注：下面的DelaySign应该为AssemblyDelaySign)
[assembly:DelaySign(true)]
```

2. 在生成程序集后，执行下面的命令以便后面可以将该程序集安装到GAC中，或者生成引用该程序集的其他程序集，以及测试该程序集。注意该操作只能执行一次，没有必要每次生成程序集时都执行一遍该命令。

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

3. 当准备打包和部署程序集时，取得公司的私有密钥(译注：这里的“私有密钥”应该为“公钥/私钥对”)，然后执行下面的命令：

```
// (译注：下面命令中最后一个参数应为“MyCompany.keys”)
SN.exe -R MyAssembly.dll MyCompanyPrivateKey.keys
```

4. 执行下面的命令，恢复验证过程以进行测试：

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

本节开始我们提到了一些组织可以将它们的密钥对保存在一个硬件设备(如一块智能卡中)中，为了保证这些密钥的安全，我们必须确保密钥值不会被保存到磁盘文件中。一些加密服务商(Cryptographic service provider，简称CSP)为抽象这些密钥的位置提供了某些“容器”。例如，微软就使用了采用容器的CSP服务，当这些容器被访问时，它会从一块智能卡中取得私有密钥。

如果我们的公钥/私钥对保存在一个CSP容器中，那么就不要再使用AssemblyKeyFileAttribute特性或者AL.exe的/keyf[ile]命令行开关。相反，我们应该使用System.Reflection.Assembly-KeyNameAttribute特性或者AL.exe的/keyn[ame]命令行开关。而当使用SN.exe将私有密钥添加到延迟签名的程序集上时，我们应该用-Rc命令行开关来代替-R命令行开关。此外，SN.exe还提供了一些命令行开关帮助我们利用CSP来执行某些操作。

重要 在部署程序集前不管我们何时对其进行操作，延迟签名都是非常有用的。例如，因为程序集仅仅是一个普通Windows PE文件，我们可能希望能够改变该文件被加载时在内存中的基地址，为了实现这一点，我们可以使用和微软的Win32平台SDK一起发布的Rebase.exe工具，但是如果程序集经过完全的签名之后，我们就不能再进行这样的操作了，因为散列值会出现错误。所以如果我们希望调整一个程序集文件的基地址，或者进行任何其他的生成之后的操作，我们都应该使用延迟签名，然后再进行生成之后的操作。并且最后要以-R或者-Rc命令行开关运行SN.exe，执行所有的散列转换，从而完成程序集的签名。

下面是我为自己所有的项目创建的 AssemInfo.cs 文件：

```
/******************************************************************************
Module: AssemInfo.cs
Notices: Copyright (c) 2002 Jeffrey Richter
******************************************************************************/

using System.Reflection;
//////////////////////////////////////////////////////////////////////////////

// 设置版本中的 CompanyName，LegalCopyright，以及 LegalTrademarks 字段
[assembly:AssemblyCompany("The Jeffrey Richter Company")]
[assembly:AssemblyCopyright("Copyright (c) 2002 Jeffrey Richter")]
[assembly:AssemblyTrademark(
   "JeffTypes is a registered trademark of the Richter Company")]

//////////////////////////////////////////////////////////////////////////////

// 设置版本中的 ProductName 和 ProductVersion 字段
[assembly:AssemblyProduct("Jeffrey Richter Type Library")]
[assembly:AssemblyInformationalVersion("2.0.0.0")]

//////////////////////////////////////////////////////////////////////////////

// 设置版本中的 FileVersion，AssemblyVersion，
// FileDescription，以及 Comments 字段
[assembly:AssemblyFileVersion("1.0.0.0")]
[assembly:AssemblyVersion("3.0.0.0")]
[assembly:AssemblyTitle("Jeff's type assembly")]
[assembly:AssemblyDescription("This assembly contains Jeff's types")]

//////////////////////////////////////////////////////////////////////////////
```

```
// 设置程序集的语言文化(其中双引号""表示语言文化为中性)
[assembly:AssemblyCulture("")]

////////////////////////////////////////////////////////////////////////////////

#if !StronglyNamedAssembly

// 弱命名程序集永远不会被签名
[assembly:AssemblyDelaySign(false)]

#else

// 强命名程序集通常在生成阶段采用延迟签名,最后
// 再用 SN.exe 的-R 或-Rc 命令行开关进行完整的签名
[assembly:AssemblyDelaySign(true)]

    #if !SignedUsingACryptoServiceProvider

    // 给出包含公钥/私钥对的文件名。
    // 如果进行的是延迟签名,则只使用公有密钥
    [assembly:AssemblyKeyFile("MyCompany.keys")]

    // 注意:如果同时指定了 AssemblyKeyFile 和 AssemblyKeyName,
    // 系统将会进行以下操作:
    // 1) 如果存在容器,密钥文件将被忽略
    // 2) 如果不存在容器,文件中的密钥将被复制到容器中,
    // 然后再对程序集进行签名

    #else

    // 给出包含公钥/私钥对的加密服务提供商(CSP)容器。
    // 如果进行的是延迟签名,则只使用公有密钥
    [assembly:AssemblyKeyName("")]

    #endif

#endif

/////////////////////////////////// 文件结束 ///////////////////////////////////
```

当我们用 Visual Studio .NET 创建一个新项目时，它会自动产生一个新的 AssemblyInfo.cs 文件，该文件中的特性和上面我个人创建的文件中的特性非常相似。但我更喜欢自己创建的文件，注释中描述了它们的用途，以及这些特性是怎样映射到程序集的版本资源信息的。另外，Visual Studio .NET 添加的 AssemblyInfo.cs 文件将 AssemblyVersion 特性错误地设为“1.0.*”，这会导致 CSC.exe 每次生成程序集的时候都自动地产生一个生成版本号和修订版本号。如果先前创建的程序集引用了该程序集的旧有版本，当 CLR 加载 CSC.exe 产生的新版程序集时将告失败。

最后，Visual Studio .NET 添加的 AssemblyInfo.cs 文件还包括一个 System.Reflection.AssemblyConfiguration 特性，但是 CLR 却从来不使用该特性。我认为应该将其从.NET 框架中完全删除掉，将其放入 AssemblyInfo.cs 文件中完全是浪费空间。

3.7 强命名程序集的私有部署

将程序集安装到 GAC 中有几个好处。首先，GAC 使得很多应用程序可以共享程序集，这从整体上减少了使用的物理内存。其次，我们很容易将一个新版的程序集部署到 GAC 中，并允许所有的应用程序通过一种发布者策略(本章稍后将予以讨论)来使用这个新的版本。最后，GAC 还提供了对不同版本程序集的并存(side-by-side)管理方式。然而，GAC 的安全策略通常只允许管理员来安装程序集。同时，向 GAC 中安装程序集也破坏了.NET 框架简单拷贝部署的许诺。

虽然强命名程序集可以被安装到 GAC 中，但并非必须这样做。实际上，仅当程序集被多个应用程序共享时，才可考虑将它们部署到 GAC 中。如果程序集不太可能被共享，我们应该以私有方式来部署它。私有部署实现了“简单”拷贝安装部署的目标，也为应用程序和程序集提供了一个更好的隔离方式。另外对于普通文件，我们不应该将 GAC 当作类似于 C:\Windows\System32 那样的新的倾倒场所，这是因为新版的程序集不会覆盖旧版的程序集，它们会以并存的方式安装，非常耗费磁盘空间。

除了以 GAC 或者私有的方式部署强命名程序集之外，我们还可以将强命名程序集部署在仅为一小部分应用程序知道的某个任意的目录中。例如，我们可能会创建三个应用程序，并希望它们共享同一个强命名程序集。

这样在安装的时候，我们就可以为每个应用程序创建一个目录，以及额外一个放置共享程序集的目录。当我们将每个应用程序安装到各自的目录中时，我们还需要为它们安装一个 XML 配置文件，并将共享程序集的 codeBase 元素指向其所在的实际路径。这样，在运行时，CLR 将知道在强命名程序集的目录中查找共享程序集。然而，事实上这种技术很少使用，并且从某种角度来看也是不被鼓励的，因为这样的话，将没有哪个应用程序能够控制共享程序集文件何时应该被卸载。

> **注意** 配置文件的codeBase元素实际上标识着一个URL。该URL可以指向用户硬盘上的任何目录或者是一个Web地址。如果是Web地址，CLR将自动下载该文件，并将其存储在用户的下载缓存中(C:\Documents and Settings\UserName\Local Settings\Application Data\Assembly\DL下的一个子目录)。如果以后再引用到这样的程序集，CLR将会从该目录中而不是URL中加载它。本章后面有一个包含codeBase元素的配置文件的例子。

> **注意** 当一个强命名程序集被安装到GAC中时，系统会确保包含清单的文件没有被篡改这种检查只在安装时出现一次。另一方面，当从一个非GAC的目录中加载强命名程序集时，CLR会验证程序集的清单文件，以确保该文件的内容没有被篡改。每次加载程序集时都会出现这种额外的性能损失。(译注：实际上，不管是以GAC方式部署的安装时，还是以非GAC方式部署的运行时，系统不仅会验证程序集中包含清单的文件，还会验证组成程序集的其他文件。)

3.8 并存执行

这里展示的情形是一个强命名程序集的例子，首先有一个 App.exe 程序集，它绑定在一个版本为 2.0.0.0 的 Calculus.dll 程序集和一个版本为 3.0.0.0 的 AdvMath.dll 程序集。而 AdvMath.dll 程序集同时又绑定着一个版本为 1.0.0.0 的 Calculus.dll 程序集。图 3.5 显示了其中的情形：

图3.5 一个需要不同版本的Calculus.dll程序集才能运行的应用程序

CLR能够将名称相同但路径不同的多个文件加载到同一个地址空间，这在.NET框架中称为并存(sidy-by-side)执行，它是解决Windows中“DLL hell”问题的一项关键技术。

重要 DLL的并存执行能力是一个很酷的特性，因为它允许我们创建的新版的程序集不必维持向后兼容，这可以减少一个产品的编码和测试时间，从而允许我们将产品更快地推向市场。

开发人员必须清楚并存执行机制以避免出现一些诡秘的 bug。例如应用程序中的某个程序集可能会创建一个命名 Win32 文件映射内核对象，并使用该对象提供的存储区域。而应用程序随后可能会加载该程序集的另一个版本，并创建有着相同名称的文件映射内核对象。然而第二个程序集并不会得到新的存储区域，相反它会直接访问第一个程序集分配的同一个存储区。如果编码的时候不仔细，两个程序集很可能会扰乱彼此的数据，应用程序的行为也就变得不可预期。

3.9 CLR 如何解析类型引用

在本书第 2 章开始，我们看到了以下代码：

```
public class App {
    static public void Main(System.String[] args) {
        System.Console.WriteLine("Hi");
    }
}
```

该段代码被编译并生成为一个程序集 App.exe。当我们运行该应用程序时，CLR 将加载并初始化它。然后 CLR 会读取该程序集的 CLR 表头来寻找标识应用程序入口点方法(Main)的 MethodDefToken。根据 MethodDef 元数据表，CLR 会定位到文件中该方法的 IL 代码所处的偏移，然后将其以 JIT 的方式编译为本地代码，同时完成代码的类型安全验证过程，最后执行编译后的本地代码。下面是 Main 方法对应的 IL 代码。要获得该项输出，我们可以运行 ILDasm.exe，并选择【视图】菜单中【显示字节】一项，然后双击树形视图中的 Main 方法。

```
.method public hidebysig static void  Main(string[] args) cil managed
// SIG: 00 01 01 1D 0E
{
    .entrypoint
    // Method begins at RVA 0x2050
    // Code size       11 (0xb)
    .maxstack  8
    IL_0000:  /*  72   | (70)000001     */ ldstr      "Hi"
    IL_0005:  /*  28   | (0A)000002     */
    call          void [mscorlib]System.Console::WriteLine(string)
    IL_000a:  /*  2A   |                */ ret
} // end of method App::Main
```

当CLR以JIT的方式编译该段代码时，它会检测到所有引用到的类型和成员，并加载定义它们的程序集(如果还没有被加载)。如我们所见，上面的IL代码中有一个引用为System.Console.WriteLine。特别地，该IL call指令引用的元数据标记为 0A000002。这个标记标识了MemberRef元数据表中的一个条目。CLR会查找该MemberRef条目，并发现其中的一个字段指向了TypeRef表中的一个条目(即System.Console类型)。从TypeRef条目中，CLR会被导向到一个AssemblyRef条目上：“MSCorLib, Version=1.0.3300.0, Culture="neutral", PublicKeyToken=b77a5c561934e089”。到了这里，CLR将知道自己需要哪个程序集。要加载该程序集，CLR必须首先找到它的位置。

在解析一个被引用的类型时，CLR可以在以下三个地方中的其中之一找到该类型：

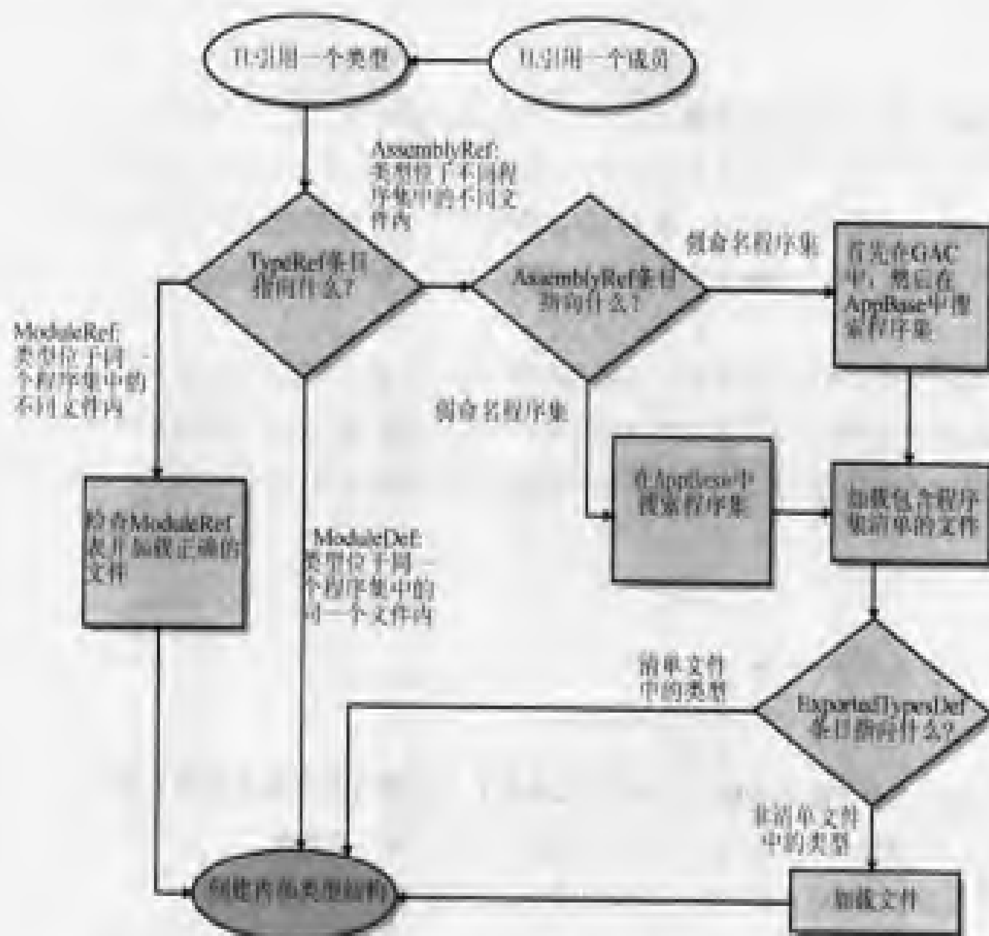
- **同一个文件** 对同一文件中类型的访问在编译时就确定了下来(有时称作早绑定，即early bound)。CLR直接从该文件中加载被引用的类型。完成加载后，程序将继续执行。
- **不同的文件，相同的程序集** CLR首先确保被引用的文件在当前程序集清单中的FileRef表(译注：这里应为FileDef表，不存在FileRef表)内。CLR然后会在加载程序集清单文件的目录中查找到被引用的文件。该文件被加载的同时，CLR会检查它的散列值以确保文件的完整性，之后便会找到相应类型的成员。完成加载后，程序将继续执行。
- **不同的文件，不同的程序集** 当被引用的类型在一个不同的程序集文件中时，CLR会首先加载包含被引用程序集的清单所在的文件。如果该文件没有包含需要的类型，CLR则会根据此清单文件加载适当的文件。这样也会找到相应类型的成员。完成加载后，程序将继续执行。

注意 ModuleDef、ModuleRef和FileDef元数据表中引用的文件使用的是文件名和扩展名。而AssemblyRef元数据表中引用的程序集使用的是文件名，而不带扩展名。当绑定一个程序集时，系统会在第2.7节中提到的目录内自动添加.dll和.exe文件扩展名来定位文件。

如果在解析类型引用的过程中出现任何错误——如文件找不到，文件不能被加载，散列值不匹配，等等诸如此类——系统将会抛出相应的异常。

在前面的例子中，CLR 会发现 System.Console 被实现在一个和调用者不同的程序集中。CLR 必须搜索其实现程序集，并加载包含程序集清单的 PE 文件。CLR 然后会扫描清单以确定该 PE 文件实现了所需的类型。如果清单文件中包含了被引用的类型，加载过程便告完成。但如果被引用的类型实现在程序集的另外一个文件中，CLR 会加载该文件，扫描其中的元数据并定位需要的类型。完成类型的加载后，CLR 会创建一个内部的数据结构来表示该类型。JIT 编译器接着将完成 Main 方法的编译，并开始执行 Main 方法。

图 3.6 演示了类型的绑定过程。



注意：如果上述操作任何一步失败，都将有相应的异常抛出。

图 3.6 流程图显示了 CLR 如何使用元数据来定位某个类型所在的程序集（类型由引用一个方法或者类型的 IL 代码导出）

重要 严格地讲，刚才描述的例子并非百分之百准确。对于定义在不同于MSCorLib.dll的程序集中的方法和类型的引用来说，上面的讨论是正确的。然而，MSCorLib.dll实际上和当前运行的CLR是紧密关联的。任何引用MSCorLib.dll(其ECMA公有密钥为b77a5c561934e089)的程序集总是被绑定到包含CLR本身的目录中的MSCorLib.dll上。所以在前面的例子中，不管程序集中的AssemblyRef元数据表内引用的是哪个版本的MSCorLib.dll，对System.Console中的WriteLine方法的引用都将被绑定到和当前CLR相匹配的MSCorLib.dll上。

本节中，我们看到了 CLR 使用默认策略来定位一个程序集的过程。然而，CLR 允许系统管理员或者程序集的发布者改写这种默认的策略。下面两节将会讨论如何改变这些默认的绑定策略。

3.10 高级管理控制(配置)

第 2.7 节为大家简要介绍了一个管理员怎样影响 CLR 搜索和绑定程序集的方式。其中演示了怎样将被引用程序集文件移动到应用程序基目录中的子目录内，以及 CLR 怎样使用应用程序的 XML 配置文件来定位这些移动后的文件。

第 2 章仅仅讨论了定位元素 privatePath 属性，本节将向大家介绍其他一些 XML 配置文件元素。下面是一个 XML 配置文件的例子：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <probing privatePath="AuxFiles;bin\subdir" />

         <dependentAssembly>

            <assemblyIdentity name="JeffTypes"
               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/JeffTypes.dll" />

         </dependentAssembly>

         <dependentAssembly>

            <assemblyIdentity name=FredTypes"
               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>
```

该 XML 文件为 CLR 提供了很多信息，下面是对它们的一个详述：

- **probing 元素** 该元素的内容指示 CLR 在应用程序基目录下的 AuxFiles 和 bin\subdir 子目录中查找弱命名程序集。对于强命名程序集，CLR 会在 GAC 中或者由 codeBase 元素指定的 URL 中查找。仅当没有指定 codeBase 元素时，CLR 才会在应用程序的私有路径下查找强命名程序集。

- **第一个 dependentAssembly、assemblyIdentity 和 bindingRedirect 元素** 这些元素的内容指示 CLR 在定位版本为 1.0.0.0、语言文化为中性、由控制着 32ab4ba45e0a69a1 公有密钥标记的组织发布的 JeffTypes 程序集时，应该转而去定位该程序集的 2.0.0.0 版本。

- **codeBase 元素** 该元素的内容指示 CLR 在定位版本为 2.0.0.0、语言文化为中性、由控制着 32ab4ba45e0a69a1 公有密钥标记的组织发布的 JeffTypes 程序集时，应该尝试在下面的 URL 地址中寻找程序集：http://www.Wintellect.com/JeffTypes.dll。虽然第 2 章中没有提及，但是 codeBase 元素也可以被用于弱命名程序集。在这种情况下，CLR 会忽略程序集的版本号以及

XML codeBase 元素中指定的版本号。另外，这时候的 codeBase URL 必须指向应用程序基目录下的一个目录。

- **第二个 dependentAssembly、assemblyIdentity 和 bindingRedirect 元素**　这些元素的内容指示 CLR 在定位版本从 3.0.0.0 到 3.5.0.0(包括 3.5.0.0)、语言文化为中性、由控制有 1f2e74e897abbcfe 公有密钥标记的组织发布的 FredTypes 程序集时，应该转而去定位该程序集的 4.0.0.0 版本。

- **publisherPolicy 元素**　该元素的内容指示如果开发 FredTypes 程序集的组织部署了一个发布者策略文件(将在下一节予以讨论)，CLR 应该忽略该文件。

当编译一个方法时，CLR 会确定其中所引用的类型和成员。由此，CLR 进一步确定当初生成调用程序集时，都引用了哪些程序集，这可以通过查找调用程序集的 AssemblyRef 表来获得。之后，CLR 会在应用程序的配置文件中查找程序集，同时会应用其中指定的版本号重定向策略。

如果 publisherPolicy 元素的 apply 属性被设置为 yes——或者如果忽略该元素——CLR 将会检查 GAC 并应用任何程序集发布者认为必要的版本号重定向策略。下一节将会对发布者策略有一个详细的讨论。

接着，CLR 会到机器内的 Machine.config 文件中查找程序集，并应用任何其中指定的版本号重定向策略。最后，CLR 将知道它应该加载的是哪个版本的程序集，并且首先试图从 GAC 中加载它。如果程序集不在 GAC 中，并且没有指定 codeBase 元素，CLR 会按照第 2 章所述的规则来定位程序集。如果执行最后一次重定向的配置文件也包含有 codeBase 元素，CLR 将会从该 codeBase 元素指定的 URL 中加载程序集。

利用这些配置文件，管理员可以完全控制 CLR 应该加载哪个程序集。如果应用程序出现了 bug，管理员可以和出错程序集的发布者联系。发布者可以发送给管理员一个新的程序集来安装。默认情况下，CLR 不会加载这个新的程序集，因为已经生成的程序集并没有引用它。然而，管理员可以修改应用程序的 XML 配置文件来指示 CLR 加载这个新的程序集。

如果管理员希望机器内所有的应用程序都采用最新的程序集，则应该修改机器中的 Machine.config 文件，这样在应用程序引用旧的程序集的时候，CLR 将会去加载最新的程序集。

如果新的程序集没有修复原来的 bug，管理员可以从配置文件中删除绑定重定向的内容，应用程序的行为将回到从前的状态。值得注意的是，系统允许我们使用和元数据中记录的版本并不匹配的程序集。这种灵活性非常方便。本章稍后会详细地介绍作为管理员怎样来轻易地修复一个应用程序。

.NET 框架配置工具

如果不喜欢手动编辑 XML 文本文件——谁会呢?——我们可以使用和.NET 框架一起发布的.NET 框架配置工具。打开【控制面板】，选择【管理工具】，然后选择 Microsoft .NET Framework Configuration 工具。在该工具中，我们可以选择【配置程序集】，该操作将弹出一个程序集的属性对话框。从该对话框中，我们可以设置所有的 XML 配置信息。图 3.7、图 3.8 和图 3.9 显示了程序集的属性对话框中三个不同的页面。

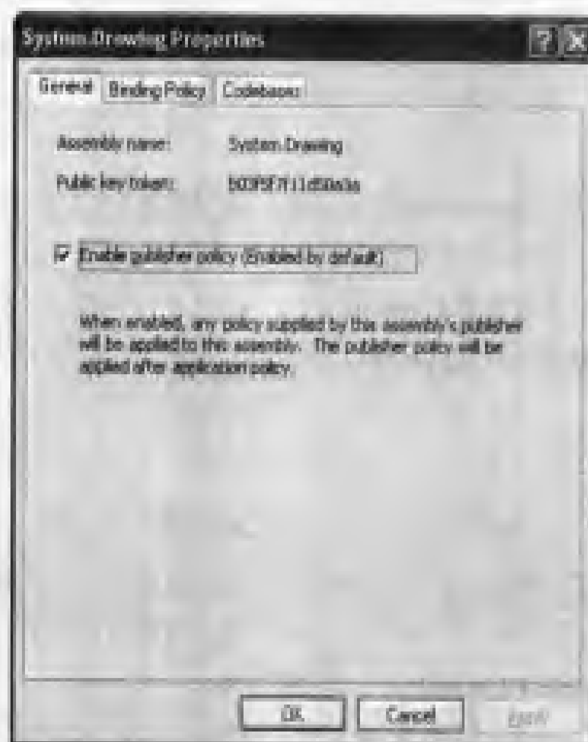


图 3.7 【System.Drawing 属性】对话框中的【常规】选项卡

.NET 框架配置工具(续)

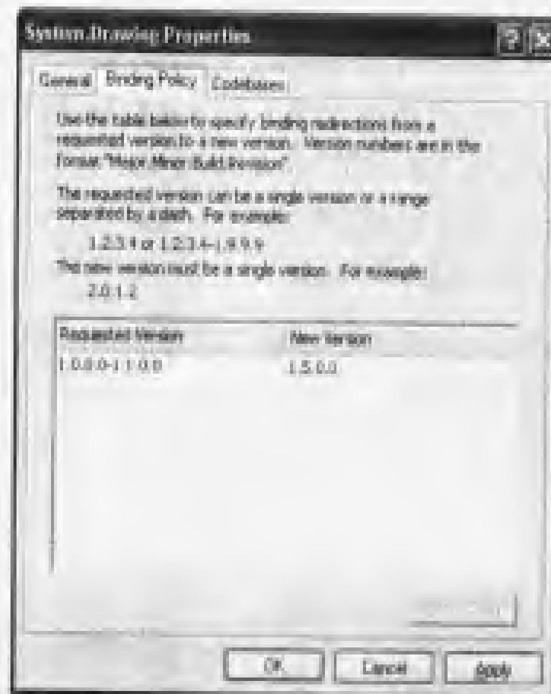


图 3.8 【System.Drawing 属性】对话框中的【绑定策略】选项卡

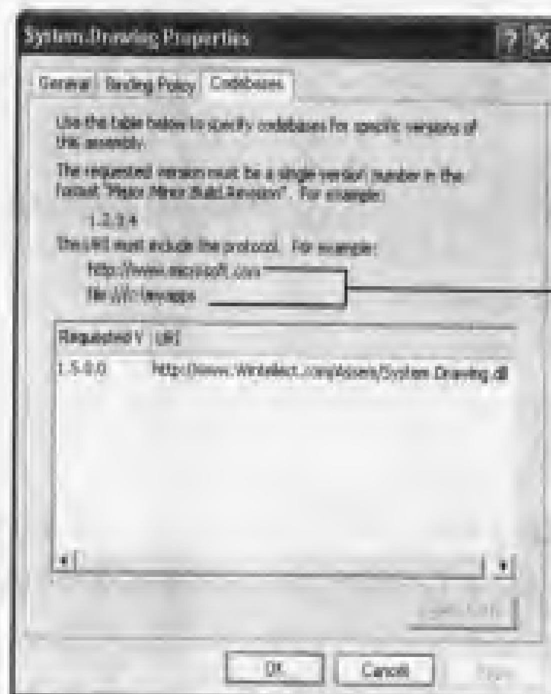


图 3.9 【System.Drawing 属性】对话框中的 codeBase 选项卡

3.10.1 发布者策略控制

在前面一节中描述的情景中，程序集的发布者只将一个新版程序集发送给管理员，管理员则要负责安装程序集，并手动编辑应用程序或者机器的 XML 配置文件。一般情况下，当发布者修复了一个程序集中的 bug，他应该用一种简单的方式对新版程序集进行打包并分发给所有的用户。他还需要一种方式来告知用户的 CLR 用新版的程序集来代替旧版的程序集。当然，每个用户都能够修改他的应用程序或者机器上的 XML 配置文件，但这非常的不方便，并且也很容易出错。发布者需要的是一种方式，使得在安装新的程序集时，能够创建可以安装到用户机器上的“策略信息”。本节即向大家展示作为程序集的发布者怎样来创建这样的策略信息。

假设我们是一个程序集的发布者，并且刚刚创建了一个新版的程序集，其中修复了一些 bug。当我们对该程序集进行打包后并发送给所有的用户时，我们也应该创建一个 XML 配置文件。它看起来很类似于我们讨论过的配置文件。下面是用于 JeffTypes.dll 程序集的一个示例文件(名称为 JeffTypes.config)：

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

                <assemblyIdentity name="JeffTypes"
                    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/JeffTypes.dll" />

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

当然，发布者只能为自己所创建的程序集设置策略。另外，上面显示的元素是可以应用于发布者策略配置文件的所有元素。例如，我们不能指定 probing 或 publisherPolicy 元素。

上面的配置文件告诉 CLR 无论何时引用 1.0.0.0 版本的 JeffTypes 程序集时，都应该加载该程序集的 2.0.0.0 版本。现在作为发布者，我们就可以创建包含发布者策略配置文件的程序集了。我们可以用如下的方式运行 AL.exe 来创建发布者策略程序集。

```
AL.exe    /out:policy.1.0.JeffTypes.dll
          /version:1.0.0.0
          /keyfile:MyCompany.keys
          /linkresource:JeffTypes.config
```

下面解释对这些AL.exe命令行开关的含义做一解释：

- **/out 开关**　该命令行开关告诉AL.exe创建一个新的PE文件，名称为Policy.1.0.JeffTypes.dll，其中除了清单外不包括任何东西。该程序集的名称很重要。名称的第一部分(即 Policy)告诉CLR该程序集包含发布者策略信息。名称的第二部分和第三部分(即 1.0)告诉CLR发布者策略程序集针对的是任何主次版本为1.0的JeffTypes程序集。发布者策略只应用于程序集的主版本号和次版本号，我们不可能创建一个具体到生成版本号或修订版本号的发布者策略。名称的第四部分(即 JeffTypes)表示与该发布者策略相对应的程序集名称。名称的第五部分也是最后一部分(即 dll)，为生成的程序集文件的扩展名。

- **/version 开关**　该命令行开关标识了发布者策略程序集的版本，它和JeffTypes程序集的版本没有任何关系。我们可以看到，发布者策略程序集也可以拥有版本。今天，发布者可能会创建一个发布者策略将版本为1.0.0.0的JeffTypes程序集重定向到版本为2.0.0.0的JeffTypes程序集。将来，发布者也许希望将版本为1.0.0.0的JeffTypes程序集重定向到版本为2.5.0.0的程序集上。CLR使用这个版本号来选择最新版本的发布者策略程序集。

- **/keyfile 开关**　该命令行开关将使AL.exe利用发布者的公钥/私钥对为发布者策略程序集签名。该密钥对必须和JeffTypes程序集所有版本的密钥对相匹配。因为这将使得CLR确保JeffTypes程序集和发布者策略文件源自同一个发布者。

- **/linkresource 开关**　该命令行开关告诉AL.exe将XML配置文件看作是一个和程序集分离的文件。最后生成的程序集将包括两个文件，两者都必须和新版的JeffTypes程序集一起打包并部署给用户。顺便说一下，不可以用AL.exe的/embedresource开关将XML配置文件嵌入到程序集文件中而生成一个单文件的程序集，因为CLR要求将XML作为一个单独的文件保存。

一旦生成了发布者策略程序集，它就可以和新版的JeffTypes.dll程序集文件一起打包并部署给用户了。发布者策略程序集文件必须被安装到GAC中。JeffTypes当然也可以安装到GAC中，但是这并非必须，它可以被部署在一个应用程序的基目录中，或者其他由codeBase URL标识的目录。

重要 仅当部署一个程序集的bug修复版或者补丁版时，才应该创建一个发布者策略程序集。当安装一个全新的应用程序时，则不应该再安装发布者策略程序集。

关于发布者策略再谈最后一点。假设一个发布者分发了一个发布者策略程序集，但是因为某种原因新的程序集又引入了比它所能修复的更多的 bug。如果是这种情况，管理员应该告诉 CLR 忽略发布者策略程序集。要使 CLR 实现这一点，管理员可以编辑应用程序的配置文件，并加入下面的 publisherPolicy 元素：

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

该元素可以放在应用程序的配置文件中来作用于所有的程序集。也可以放在应用程序的配置文件中使其作用于某个特定的程序集。当 CLR 处理应用程序的配置文件时，它将会发现不应该再到 GAC 中检查发布者策略程序集了。这样，CLR 就会使用旧版的程序集继续操作。注意，CLR 仍旧会检查并应用任何在 Machine.config 文件中指定的策略。

重要 发布者策略程序集本身就表明了程序集不同版本之间是兼容的。如果一个新版的程序集不能和旧版的程序集兼容，发布者就不应该去创建发布者策略程序集。一般来说，当生成一个修复了某些bug的新版程序集时，才会使用发布者策略程序集。并且还应该测试新版程序集的向后兼容性。另一方面，如果是为程序集添加一些新的功能，我们应该将新增的程序集看作和先前的版本没有任何关系，也不应该再添加发布者策略程序集，这时也就没有做向后兼容性测试的必要了。

3.11 修复错误的应用程序

当控制台或者 Windows 窗体应用程序正在一个用户帐号下运行时，CLR 会保持一个应用程序实际加载的程序集的记录。但 ASP.NET 或者 XML Web 服务应用程序没有这样的记录。这些程序集加载信息被累积在内存中，并在应用程序结束时被写入磁盘中。包含这些信息的文件被写入以下目录：

```
C:\Documents and Settings\UserName\Local Settings\
    Application Data\ApplicationHistory
```

其中 UserName 表示登录用户的名称。

如果浏览一下该目录，我们将会看到以下一些文件：

```
Volume in drive C has no label.
  Volume Serial Number is 94FA-5DE7

   Directory of C:\Documents and Settings\vjeffrr\
Local Settings\Application Data\ApplicationHistory

07/23/2002   10:46 AM      <DIR>       .
07/23/2002   10:46 AM      <DIR>       ..
07/22/2002   04:14 PM               1,014    App.exe.c4bc1771.ini
07/23/2002   10:46 AM               2,845    ConfigWizards.exe.c4c8182.ini
07/14/2002   05:51 PM               9,815    devenv.exe.49453f8d.ini
07/22/2002   02:25 PM               3,226    devenv.exe.7dc18209.ini
07/23/2002   10:46 AM               3,368    mmc.exe.959a7e97.ini
07/15/2002   03:06 PM               2,248    RegAsm.exe.18b34bd3.ini
                6 File(s)          22,516 bytes
                2 Dir(s)   14,698,717,184 bytes free
```

其中每一个文件都标识着一个特殊的应用程序。其中的 16 进制数是一个标识文件路径的散列值，它用来区分不同子目录下的同名文件。

当一个应用程序运行时，CLR 会维持一个应用程序所加载的程序集的集合“快照”。当应用程序结束时，该信息将和应用程序相关联的.ini 文件中的内容比较。如果应用程序当前加载的程序集集合和先前加载的程序集集合相同，也就是说.ini 文件中的信息和内存中的信息相同，内存中的信息将被丢弃。另一方面，如果内存中的信息不同于.ini 文件中的信息，CLR 会将内存中的信息追加到.ini 文件中。默认情况下，.ini 文件能够存储五个快照。

基本上来说，CLR保持着一个应用程序使用的程序集记录。现在，假设我们安装了一些新的程序集，并且可能有一些新的发布者策略程序集。一周后，当我们运行一个应用程序的时候突然发现它不能正常运行了。怎么办呢？在Windows时代，最好的做法就是重新安装出错的应用程序，并且还要祈祷这种重装不会破坏其他的应用程序(事实上是有可能的)。

幸运的是对于终端用户，CLR保持了一个应用程序使用的程序集的历史记录。所有我们需要做的就是为应用程序创建一个XML配置文件，其中的元素告诉CLR使用最近一次运行正常时加载的程序集。

为了使创建和修改应用程序的配置文件更加容易，我们可以使用.NET框架配置工具。运行该工具，并右击树形面板上的【应用程序】节点，选择【修复应用程序】菜单项，将会弹出如图3.10所示的对话框。

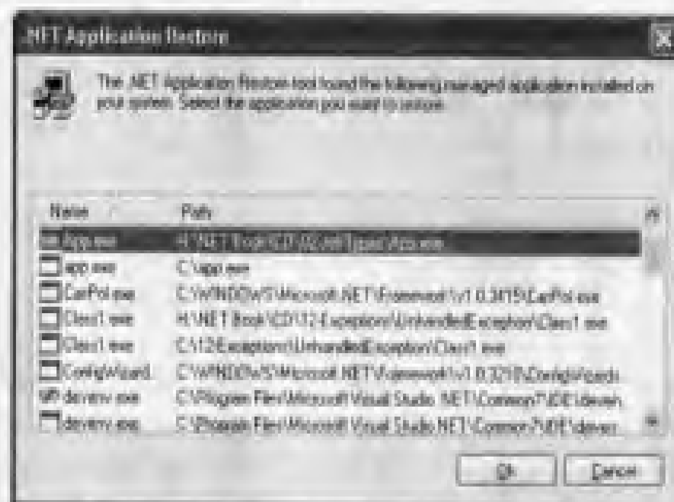


图3.10 .NET应用程序配置工具中显示了所有应用程序的加载信息记录

注意 .NET框架配置工具是一个微软管理控制单元(Microsoft Management Console，简称MMC)插件，因此没有安装到Windows 98、Windows 98 第2版，以及Windows Me上。但是，在这些操作系统上，我们可以使用【.NET框架向导】实用程序来执行本节描述的操作。可以从【开始】菜单起，选择【程序】，然后选择【管理工具】，最后点击【.NET框架向导】来调用该工具。

图3.10中的对话框显示了一些应用程序，以及CLR为它们累积的一些程序集加载信息。基本上，ApplicationHistory子目录中的每个.ini文件都在这里有一个条目出现。如果我们选择一个应用程序，就会有如图3.11所示的对话框出现。

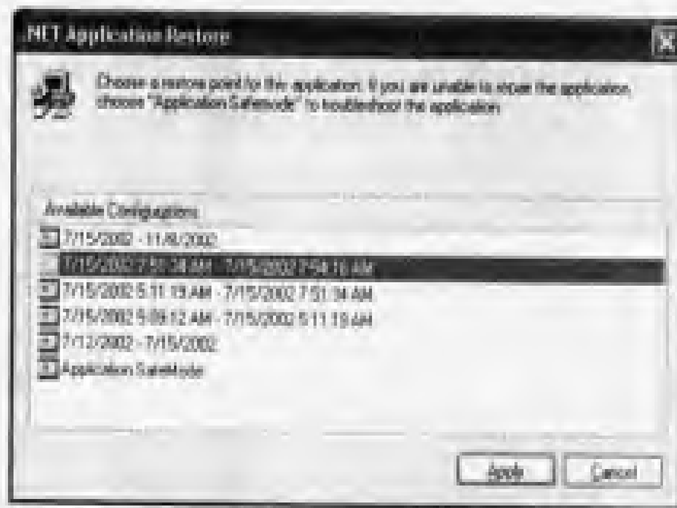


图3.11 .NET应用程序配置工具中显示了不同程序集加载的日期

图3.11中对话框内的每一个条目都表示一个被应用程序加载的程序集集合。用户可以选择一个日期范围(在这期间应用程序可以正常工作)，该工具就会创建或者修改应用程序的配置文件以使CLR现在可以加载其最近一次正常运行时使用的程序集集合。其中Application SafeMode条目可以确保应用程序加载的是当初被生成和测试时使用的程序集集合，它将阻止CLR将程序集重定向到一个不同的版本上。

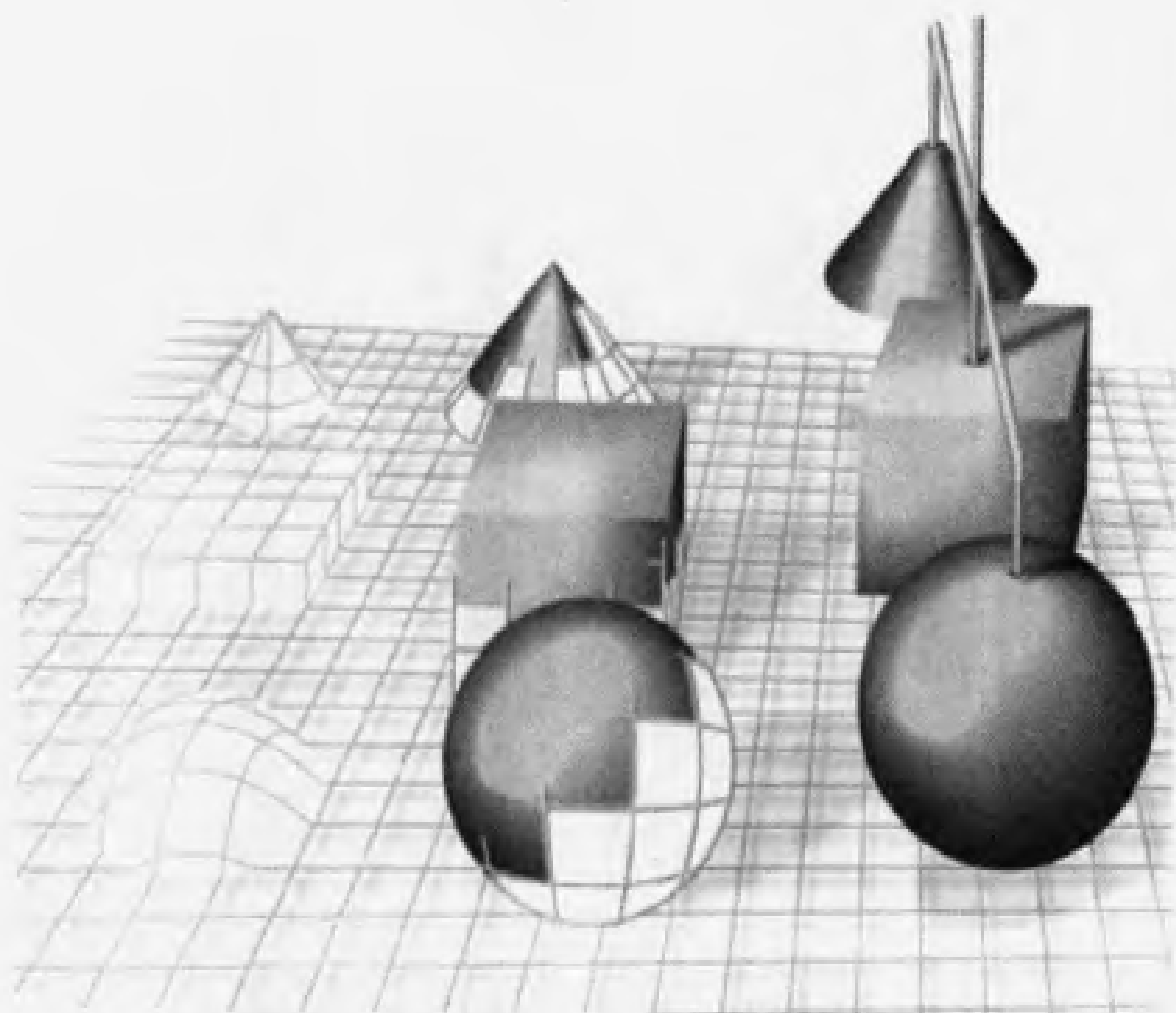
对应用程序XML配置文件所做的改变会被周围的注释元素所标识，这些元素包括有“.NET Application Restore BeginBlock”和“.NET Application Restore EndBlock”。另外，“.NET Application Restore RollBackBlock”元素包含着应用程序恢复为特定快照之前原来的XML配置。下面是一个改变后的XML配置文件示例：

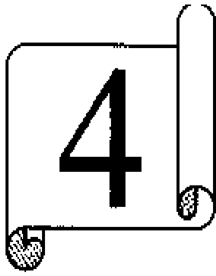
```
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
         <dependentAssembly>
```

```
            <!--.NET Application Restore BeginBlock #1 29437104.-387708080
               8/24/2001 6:48:25 PM-->

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

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

            <publisherPolicy apply="no"/>
            <!--.NET Application Restore EndBlock #1-->

            <!--.NET Application Restore RollBackBlock #1
               8/24 2001 6:48:25 PM<assemblyIdentity name="JeffTypes"
               publicKeyToken="32ab4ba45e0a69a1" culture=""/>-->

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

第Ⅱ部分

类型与通用语言运行时





类型基础

本章主要介绍有关类型和通用语言运行时(CLR)的一些基础知识。特别地，我们将会讨论每个类型都具有的一组公共行为。另外，我们还会谈到类型安全、对象转型以及有关命名空间与程序集的一些话题。

4.1 所有类型的基类型：System.Object

CLR 要求每个类型最终都要继承自 System.Object 类型。这意味着下面两种类型定义(使用 C#描述)是等同的：

```
// 隐式继承 Object
class Employee {
    ...
}

// 显式继承 Object
class Employee : System.Object {
    ...
}
```

因为每个类型最终都会继承自 System.Object，所以我们就可以确保任何对象都有一组方法的最小集合。表 4.1 列出了 System.Object 类提供的几个公有实例方法。

表 4.1 System.Object 的公有方法

公有方法	描 述
Equals	如果两个对象具有相同的值，方法将返回 true。有关该方法的更多信息，请参见第 6 章
GetHashCode	方法返回对象的值的散列码。如果一个对象被用作散列表的一个键值，那么该对象的类型应该重写 GetHashCode 方法。该方法应该为类型的不同对象之间提供一个良好的区分。有关该方法的更多信息，请参见第 6 章
ToString	默认情况下，方法返回类型全名(this.GetType().FullName.ToString())。但是，还有一种常见的做法是重写该方法使其返回一个表示对象状态的字符串。例如，.NET 框架中的一些核心类型(如 Boolean 和 Int32)就重写了该方法，使其返回表示它们的值的一个字符串。另一种常见的做法是重写该方法以用于调试目的：我们可以通过调用它来得到一个表示对象字段值的字符串。注意 ToString 通常会利用与调用线程相关联的 CultureInfo。第 12 章将讨论 ToString 方法的更多细节
GetType	方法返回一个类型为继承自 Type 的对象实例，其标识了该方法所属对象的类型。返回的 Type 对象可以和反射类一起使用来获得类型的元数据信息。反射将在本书第 20 章讨论。注意 GetType 方法是一个非虚方法，这可以防止一个类通过重写该方法而隐瞒它的类型，从而破坏类型安全

另外，继承自 System.Object 的类型还可以访问表 4.2 中列出的受保护方法。

表 4.2 System.Object 的受保护方法

受保护方法	描 述
MemberwiseClone	这是一个非虚方法，它创建一个新的类型实例，并将其字段设置为和 this 对象的字段相同，最后返回新创建的实例引用。关于该方法的更多信息，请参见第 6 章
Finalize	这是一个虚方法，当垃圾收集器判定某个对象为可回收的垃圾时，垃圾收集器会在对象内存被回收之前调用此方法。那些内存回收时需要资源清理的类型应该重写该方法。这是一个非常重要的方法，本书第 19 章将探讨有关它的更多细节

CLR 要求所有的对象都要用 new 操作符来创建(该操作符将产生 newobj IL 指令)。下面的语句展示了怎样创建一个 Employee 对象：

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

下面是 new 操作符所执行的工作：

1. 从托管堆(managed heap)中分配指定类型所需数量的字节来作为存储其对象的内存空间。

2. 初始化对象的附加成员(overhead members)。每一个对象实例都有两个附加成员。其中第一个成员为指向类型方法表的指针，第二个成员为 SyncBlockIndex(译注：CLR 使用该字段来进行线程同步控制。该字段的某些位还用作垃圾收集时的标记。另外 FCL 中 System.Object 类型默认的 GetHashCode 方法也用到了该字段)。CLR 使用这两个成员来管理对象实例。

3. 传入 new 操作符中指定的参数(上面的例子为"ConstructorParam1")，调用类型的实例构造器。虽然大多数语言在编译构造器时都会要求它们调用基类型中相应的构造器，但 CLR 本身没有这样的要求。

在 new 完成所有这些操作后，它将返回一个指向新创建对象的引用。在上面的例子中，该引用被保存在变量 e 中，其类型为 Employee。

顺便说一句，没有和 new 操作符对应的 delete 操作符。也就是说，在 CLR 中，我们无法显式释放对象所占用的内存。CLR 引入一种垃圾收集环境(将于第 19 章讨论)来自动检测那些不再被使用或访问的对象，并自动释放它们的内存。

4.2 类型转换

CLR 最重要的一个特性就是类型安全。CLR 在运行时总能知道一个对象的类型。我们也可以通过调用 GetType 方法来得到对象的准确类型。因为该方法是一个非虚方法，所以我们不可能利用它来篡改一个类型的信息。例如，我们不可能重写 Employee 类的 GetType 方法使之返回一个 SpaceShuttle 类型。

开发人员经常会发现需要将一个对象转换为其他类型。CLR 允许我们将对象转换为其原来的类型或者它的任何一个基类型。各个编程语言自己决定如何提供这些转型操作。例如，C#不需要任何特殊的语法就可以将对象转换为其任何一个基类型，因为转换为基类型被认为是安全的隐式操作。然而在

将对象转换为它的任何派生类型时，C#要求进行显式转型，因为这样的转型有可能会失败。下面的代码演示了如何将对象转换为它的基类型和派生类型：

```
// 该类型隐含继承自 System.Object.
class Employee {
    ...
}

class App {
    public static void Main() {
        // 这里不需要转型，因为 new 返回的是一个 Employee 对象。
        // 而 Object 又是 Employee 的基类型
        Object o = new Employee();

        // 这里需要转型，因为 Employee 继承自 Object.
        // 其他一些语言 (如 Visual Basic)可能不需要转型
        // 就可以编译通过
        Employee e = (Employee) o;
    }
}
```

上面的例子显示了要使代码通过编译我们需要做的一些事情。下面解释运行时发生的行为。在运行时，CLR会检查转型操作以确保总是将对象转型它的实际类型、或者它的任何基类型。例如，下面的代码虽然能够通过编译，但在运行时，却会抛出InvalidCastException异常：

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

class App {
    public static void Main() {
        // 构造一个 Manager 对象并将其传递给 PromoteEmployee。
        // 一个 Manager "IS-A" Object，PromoteEmployee 将正常运行
        Manager m = new Manager();
        PromoteEmployee(m);

        // 构造一个 DateTime 对象并将其传递给 PromoteEmployee。
        // 由于 DateTime 并非继承自 Employee，因此 PromoteEmployee
        // 会抛出一个 System.InvalidCastException 异常
        DateTime newYears = new DateTime(2001, 1, 1);
        PromoteEmployee(newYears);
    }
```

```
    public static void PromoteEmployee(Object o) {
        // 这里，编译器并不知道对象 o 引用的实际类型。
        // 所以编译器允许代码通过编译。然而，在运行时，
        // CLR 会获知 o 引用的类型(每当进行转型操作时)，
        // 并且会检查对象的类型是否为 Employee，或者任何
        // 继承自 Employee 的类型
        Employee e = (Employee) o;
        ...
    }
}
```

在 Main 方法中，我们首先构造了一个 Manager 对象，并将其传递给 PromoteEmployee。这段代码会成功编译并执行，因为 Manager 继承自 Object，而 Object 正是 PromoteEmployee 期望的参数类型。一旦进入 PromoteEmployee 内部，CLR 将确认 o 引用的对象或者是一个 Employee，或者是继承自 Employee 的一个类型。因为 Manager 继承自 Employee，所以 CLR 将成功执行转型操作，并允许 PromoteEmployee 继续执行。

在 PromoteEmployee 返回之后，Main 又构造了一个 DateTime 对象并将其传递给 PromoteEmployee。同样 DateTime 也继承自 Object，所以调用 PromoteEmployee 的代码将通过编译。但是，在 PromoteEmployee 内部，CLR 会检查转型操作，并将检测出 o 引用的对象为 DateTime 类型，而不是 Employee 或者任何继承自 Employee 的类型。这时，CLR 将禁止执行转型操作，并抛出一个 System.InvalidCastException 异常。

如果 CLR 允许这样的转型操作，代码也就失去了类型安全，并且结果也将变得不可预期——包括应用程序可能崩溃、以及由于类型欺骗引起的安全漏洞。类型欺骗是很多安全漏洞的原因，它会极大地危及应用程序的稳定性和健壮性。类型安全是.NET 框架中非常重要的一个部分。

顺便说一句， PromoteEmployee 方法的正确定义应该是接受一个 Employee 对象，而不是 Object 对象作为参数。这里使用 Object 的目的仅仅是为了演示编译器和 CLR 是如何处理转型操作的。

4.2.1 使用 is 和 as 操作符转型

C#提供了一种利用 is 操作符进行转型的方式。它可以检查对象是否和给定的类型兼容，并返回判断结果：true 或者 false。另外，is 操作符永远不会抛出异常。看下面的代码：

```
System.Object o = new System.Object();
System.Boolean b1 = (o is System.Object);      // b1 为 true
System.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，因为没有对象可以用来检查其类型。下面演示了is操作符典型的使用方法：

```
if (o is Employee) {
    Employee e = (Employee) o;
    // 在 'if' 语句中使用e
}
```

在上面的代码中，CLR实际上对对象的类型检查了两次：is操作符首先检查o所引用的对象是否和Employee类型兼容。如果兼容，在if语句内，CLR在执行转型时又会检查o是否为一个Employee引用。由于这种编程范式十分常见，C#便为我们提供了一种新的转型方式，即as操作符，它可以在简化代码的同时提高性能。看下面的代码：

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

在上面的代码中，CLR会检查o所引用的对象是否和Employee类型兼容。如果兼容，as返回一个指向同一个对象的非空指针。如果不兼容，as返回null。注意，在as操作符执行过程中，CLR只检查了一次对象的类型。紧接着的if语句只需要检查e是否为null就可以了——这种检查要比检查对象的类型高效得多。

除了不会抛出异常外，as操作符所做的和通常的转型操作没什么不同。如果对象不能顺利转型，as操作符的结果将为null。我们应该检测其结果是否为null，然后再进行操作。否则，试图使用空引用将会导致抛出System.NullReferenceException异常。

```
System.Object o = new System.Object();     // 创建一个新对象
Employee e = o as Employee;                // 将o转型为一个Employee
// 上面的转型失败，但并没有异常抛出，只是e被设为null
e.ToString();     // 访问e将导致抛出一个NullReferenceException
```

为了使大家理解上面讨论的所有内容，我们来做一个小测验。假设存在以下两个类定义：

```
class B {
    Int32 x;
}

class D : B {
    Int32 y;
}
```

现在来看表 4.3 中的 C#代码。对于表中的每一行，我们来判断其是否能够通过编译并且顺利执行(标记为 OK)，或者导致编译时错误(标记为 CTE)，或者导致运行时错误(标记为 RTE)。

表 4.3　类型安全小测验

语　句	OK	CTE	RTE
System.Object o1 = new System.Object();	√		
System.Object o2 = new B();	√		
System.Object o3 = new D();	√		
System.Object o4 = o3;	√		
B b1 = new B();	√		
B b2 = new D();	√		
D d1 = new D();	√		
B b3 = new System.Object();		√	
D d3 = new System.Object();		√	
B b4 = d1;	√		
D d2 = b2;		√	
D d4 = (D) d1;	√		
D d5 = (D) b2;	√		
D d6 = (D) b1;			√
B b5 = (B) o1;			√
B b6 = (D) b2;	√		

4.3　命名空间与程序集

命名空间允许我们对相关类型进行逻辑上的组织，这使得我们可以很方便地定位一个类型。例如，System.Collections 命名空间定义了一组集合类型，而 System.IO 定义了一组执行 I/O 操作的类型。下面的代码构造了一个 System.IO.FileStream 对象和一个 System.Collections.Queue 对象：

```
class App {
   static void Main() {
      System.IO.FileStream fs = new System.IO.FileStream(...);
      System.Collections.Queue q = new System.Collections.Queue();
   }
}
```

如我们所见，上面的代码非常冗长。如果能够有一些 FileStream 和 Queue 的缩写方式来减少代码录入就好了。幸运的是，许多编译器都提供了这样的机制来减少开发人员的录入工作。C#编译器通过 using 指示符提供了这种机制，而 Visual Basic 则通过 Imports 语句来提供这种机制。下面的代码和前面的例子是等同的：

```
// 导入一些命名空间
using System.IO;           // 试着在类型名上加前缀 "System.IO."
using System.Collections; // 试着在类型名上加前缀 "System.Collections."

class App {
    static void Main() {
        FileStream fs = new FileStream(...);
        Queue q = new Queue();
    }
}
```

对于编译器来说，命名空间仅仅是在类型名称前加上了一些由点号隔开的符号而已。这使得一个类型的名称更长，从而也更具惟一性。在上面的例子中，编译器将把 FileStream 解析为 System.IO.FileStream。类似地，它还会把 Queue 解析为 System.Collections.Queue。

使用 C#中的 using 指示符或者 Visual Basic 中的 Imports 语句完全是可选的；如果愿意，我们仍然可以键入类型的完全限定名。C#的 using 指示符会指示编译器试着在类型名上添加不同的前缀，直到找到一个匹配为止。

重要 CLR 实际上对命名空间一无所知。当我们访问一个类型时，CLR 需要知道该类型的完整名称，以及哪个程序集包含着类型的定义，这样它才能加载正确的程序集，找到需要的类型，并执行相关的操作。

在前面的例子中，编译器需要确保每一个引用的类型都存在，并且代码以正确的方式使用了类型：即类型中存在调用的方法，调用方法时传递了数量适当的参数，对应的参数和期望的类型一致，正确地使用了方法的返回值，等等。如果编译器不能在源代码、或者任何引用程序集中找到指定名称的类型，它会试着在类型名称上添加 "System.IO." 前缀，并检查产生的名称是否匹配一个已有的类型。

如果编译器仍然找不到一个匹配，它将试着在类型名称上添加“System.Collections.”前缀。这样，前面代码中的两个 using 指示符就允许我们在代码中简单地键入“FileStream”和“Queue”——编译器会自动将其扩展为“System.IO.FileStream”和“System.Collections.Queue”。可以想象这会节省我们大量的时间。

当查找一个类型的定义时，编译器必须被告知到哪些程序集中进行查找。编译器将扫描它知道的所有程序集来查找类型的定义。一旦编译器找到了正确的程序集，程序集信息和类型信息会被添加到生成托管模块的元数据中。要得到程序集信息，我们必须将引用的类型所在的程序集传递给编译器。C#编译器默认情况下会自动在 MSCorLib.dll 程序集中进行查找，即使我们没有显式告诉它这一点。MSCorLib.dll 程序集包含了所有.NET 框架类库(FCL)中定义的核心类型，例如 Object、Int32、String，等等。

> **注意**　当微软刚开始进行.NET 框架的工作时，MSCorLib.dll 是 Microsoft Common Object Runtime Library(微软通用对象运行时库)的首字母缩写。当 ECMA 开始标准化 CLR 以及部分的 FCL 时，MSCorLib.dll 正式成为 Multilanguage Standard Common Object Runtime Library(多语言标准通用对象运行时库)的首字母缩写。☺

可以想象，这种对待命名空间的方式有一些潜在的问题：即不同的命名空间中有可能存在同名的两个(或多个)类型。微软强烈建议我们为自己的类型定义一个独一无二的名字。然而，在某些情况下，这有些不可能。CLR 鼓励组件重用。例如，我们的应用程序可能同时使用了微软创建的一个组件和 Wintellect 创建的一个组件。两个公司可能都提供了一个名为 Widget 的类型——微软的 Widget 类型实现的是某一种功能，而 Wintellect 的 Widget 类型实现的是完全不同的另一种功能。在这种情况下，我们将只能用它们的完全限定名来区分两个 Widget。当引用微软的 Widget 类型时，我们应该使用 Microsoft.Widget，而当引用 Wintellect 的 Widget 类型时，我们应该使用 Wintellect.Widget。

在下面的例子中，由于对 Widget 的引用具有二义性，C#编译器将产生以下错误："error CS0104: 'Widget' 是不明确的引用"：

```
using Microsoft;      // 试着在类型名上加前缀 "Microsoft."
using Wintellect;     // 试着在类型名上加前缀 "Wintellect."

class MyApp {
    static void Main() {
        Widget w = new Widget();        // 这里具有二义性
    }
}
```

要消除这种二义性，我们必须明确告诉编译器我们希望创建的是哪一个 Widget：

```
using Microsoft;      // 试着在类型名上加前缀 "Microsoft."
using Wintellect;     // 试着在类型名上加前缀 "Wintellect."

class MyApp {
    static void Main() {
        Wintellect.Widget w = new Wintellect.Widget();       // 这里不具二义性
    }
}
```

C#还提供了另一种形式的 using 指示符允许我们为一个类型或者命名空间创建另外的别名。如果我们有来自某一命名空间的几个类型，并且不想在所有这些类型前都加上全局命名空间，这种方式就显得非常方便。下面的代码就采用了这种方式来解决前面代码中存在的二义性问题：

```
using Microsoft;      // 试着在类型名上加前缀 "Microsoft."
using Wintellect;     // 试着在类型名上加前缀 "Wintellect."

// 定义 WintellectWidget 符号作为 Wintellect.Widget 的别名
using WintellectWidget = Wintellect.Widget;

class MyApp {
    static void Main() {
        WintellectWidget w = new WintellectWidget();         // 不存在任何错误
    }
}
```

这些消除类型二义性的方法非常有用，但是在某些情况下，我们需要的可能更多。设想 Australian Boomerang Company (ABC)和 Alaskan Boat Corporation (ABC)都创建了一个名为 BuyProduct 的类型，他们都希望在各自的程序集中发布该类型。很有可能两个公司都创建了一个名为 ABC 的命名空间，并且其中包含着名为 BuyProduct 的类型。这时试图开发需要购买 boomerang 和 boat 的应用程序将会陷入一些麻烦，除非编程语言提供一种按照程序集，而不仅仅是命名空间，来区别类型的方法。

不幸的是，C#的 using 指示符仅支持命名空间，它没有提供任何指定程序集的方式。还好，在程序设计实践中，这种情况并不常见，很少成为问题。如果我们正在设计期望为第三方使用的组件类型，我们应该将这些类型定义在一个易于被编译器区分的命名空间中。实际上，为了减少冲突的可能性，我们应该使用自己公司的全名(而非缩写)作为顶级的命名空间名称。浏览.NET 框架 SDK 文档，我们会看到微软针对自己专有的一些类型就使用了名为“Microsoft”的命名空间(例如 Microsoft.CSharp，Microsoft.VisualBasic，Microsoft.Win32)。

命令空间的创建很简单，我们只需将其声明直接放在代码中就可以了，看下面的示例(C#描述)：

```
namespace CompanyName {          // CompanyName
   class A {                     // CompanyName.A
      class B { ... }            // CompanyName.A.B
   }

   namespace X {                 // CompanyName.X
      class C { ... }            // CompanyName.X.C
   }
}
```

一些编译器根本不支持命名空间，而一些编译器允许定义针对某一特定语言的命名空间。在 C#中，命名空间隐含为公有，而且我们不能用任何访问修饰符来修改它。但是，C#允许我们在命名空间中定义内部类型(不能在程序集之外访问)，或者公有类型(可以在任何程序集中访问)。

命名空间和程序集的关系

注意，命名空间和程序集(实现类型的文件)并非必然相关。特别地，多个属于同一命名空间的类型可能被实现在多个程序集中。例如，System.IO.FileStream 类型被实现在 MSCorLib.dll 程序集中，而 System.IO.FileSystemWatcher 类型被实现在 System.dll 程序集中。实际上，根本没有 System.Collections.dll 程序集。

一个程序集也可以包含位于不同命名空间中的类型。例如 System.Int32 和 System.Collections.ArrayList 类型都位于 MSCorLib.dll 程序集中。

当我们在.NET 框架 SDK 文档中查找一个类型时，该文档会清楚地说明类型所属的命名空间，以及其实现所在的程序集。在图 4.1 中的【要求】部分，我们可以看到 ResXFileRef 类型属于 System.Resources 命名空间，但是却实现在 System.Windows.Forms.dll 程序集中。要编译包含引用 ResXFileRef 类型的代码，我们需要在源代码中添加 using System.Resources;指示符，同时还要使用/r:System.Windows.Forms.dll 编译器开关进行编译。

图 4.1 【要求】部分显示了类型的命名空间和程序集信息

基元类型、引用类型与值类型

本章讨论.NET 框架开发人员经常遇到的各种数据类型。熟悉这些类型的不同行为对于一个开发人员来说至关重要。当我刚开始接触.NET 框架时，就没有完全理解基元类型、引用类型和值类型之间的一些差别。这种模糊的认识甚至无意间导致了一些难以查找的 bug 以及性能问题。我希望通过本章的解释，能够帮助大家在提升代码效率的同时避免我曾遇到的一些麻烦。

5.1 基 元 类 型

某些数据类型的使用非常频繁，许多编译器都允许我们用某种简化的语法来操作它们。例如，我们可以用下面的语法来分配一个整数：

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

相信大家都会感到用这样的语法来声明和初始化一个整数未免太过麻烦。所幸，许多编译器(包括 C#)都允许我们使用类似下面的语法来对整数进行声明和初始化：

```
int a = 0;
```

这种语法大大提高了代码的可读性，并且经编译后和前一种语法生成的是同样的 IL 代码。编译器直接支持的数据类型称为**基元类型**(primitive type)。

基元类型和.NET 框架类库(FCL)中的类型有直接的映射关系。例如，在 C#中，int 直接映射为 System.Int32 类型。因此，下面 4 行代码都能通过编译，并产生同样的 IL 代码：

```
int a = 0;                                    // 最便捷的语法
System.Int32 a = 0;                           // 较便捷的语法
int a = new int();                            // 不便捷的语法
System.Int32 a = new System.Int32();          // 不便捷的语法
```

表 5.1 显示了 FCL 中的类型和它们在 C#中对应的基元类型。对于那些和通用语言规范(CLS)兼容的类型，其他开发语言也都提供了类似的基元类型。但对于那些与 CLS 不兼容的类型，则无此必要。

表 5.1 FCL 类型及其在 C#中对应的基元类型

C#中的基元类型	FCL 类型	是否与 CLS 兼容	描 述
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(不像非托管 C++中那样，char 表示的是一个 8 位值)
float	System.Single	是	IEEE 32 位浮点数
double	System.Double	是	IEEE 64 位浮点数
bool	System.Boolean	是	一个 True 或者 Flase 值
decimal	System.Decimal	是	128 位高精度浮点值，通常用于不容许有舍入误差的金融计算场合。在这 128 位中，1 位表示浮点值的符号，96 位表示浮点值本身(译注：一个整数值，小数点位置由下面 8 个位来确定)，8 位表示用 96 位值除以浮点值所得结果的 10 的幂次(幂次范围为 0~28)。其余的位尚未使用
object	System.Object	是	所有类型的基类型
string	System.String	是	一个字符数组

C#语言规范声称“作为一种编码风格，使用关键字应该优于使用完整的系统类型名称”。我个人不同意这段论述。我更喜欢使用 FCL 类型名，并且完全避免使用基元类型名称。实际上，我希望编译器甚至不要提供基元类型名称，并强制开发人员使用 FCL 类型名。理由如下：

- 我发现很多开发人员都困惑于不知在代码中使用 string 还是 String。因为 C#中的 string(关键字)实际上映射了 System.String(FCL 类型)，所以两者之间没有任何不同，都可以使用。

- 在 C#中，long 映射为 System.Int64，但是在其他的编程语言中，long 可能映射为一个 Int16 或者 Int32。实际上，托管扩展 C++将 long 看作是一个 Int32。如果习惯了在一种语言下编程，再转而去阅读用另一种语言编写的代码就很容易误解其中的意图。实际上，大多数语言甚至不将 long 看作是关键字，也不会编译使用它的代码。

- FCL 中有很多类型的方法都将一些类型名作为方法名称的一部分。例如，BinaryReader 类型就提供了诸如 ReadBoolean、ReadInt32、ReadSingle 之类的方法；而 System.Convert 类型也提供了诸如 ToBoolean、ToInt32、ToSingle 之类的方法。虽然下面的代码是合法的，但其中含有 float 的代码行在我看来总有些不自然，这段代码的正确性也并不明显：

```
BinaryReader br = new BinaryReader(...);
float val = br.ReadSingle();            // 正确，但是有些不自然
Single val = br.ReadSingle();           // 正确，并且感觉也很自然
```

因为上面所有这些原因，本书将通篇采用 FCL 类型名。

在许多编程语言中，我们可能希望下面的代码能够正确地编译并执行：

```
Int32 i = 5;        // 一个 32 位的值
Int64 l = i;        // 隐式转型为一个 64 位的值
```

然而，基于第 4 章中对转型的讨论，大家可能会认为这段代码不能通过编译。毕竟，System.Int32 和 System.Int64 是不同的类型。但是，我们可能会很高兴看到 C#编译器能够正确地编译这段代码，并且能够按我们的预期运行。为什么呢？

这是因为 C#编译器熟悉基元类型，并且在编译代码时会应用自己的规则。换句话说，我们所选的编译器能够识别一些通用的编程模式，并产生必要的 IL 指令来使代码按期望的方式运行。

具体而言，编译器一般会支持和类型转换、文本常量(literals)、操作符相关的一些模式。看下面的例子。

首先，编译器能够在基元类型之间进行隐式或者显式的转型：

```
Int32 i = 5;                    // 从 Int32 到 Int32 的隐式转型
Int64 l = i;                    // 从 Int32 到 Int64 的隐式转型
Single s = i;                   // 从 Int32 到 Single 的隐式转型
Byte b = (Byte) i;              // 从 Int32 到 Byte 的显式转型
Int16 v = (Int16) s;            // 从 Single 到 Int16 的显式转型
```

如果两个类型之间的转换是“安全”的，那么 C#允许在它们之间进行隐式转型，这里“安全”的意思是指转换过程中不会造成数据丢失，例如将 Int32 转换为 Int64。但是，如果转型存在潜在的“不安全”，C#将要求显式转型。对于数值类型，“不安全”的转型意味着我们可能会因此丢失精度或者数量级。例如将 Int32 转换为 Byte 就要求为显式转换，因为如果是一个较大的 Int32 数值，这样的转换将丢失精度。将 Single 转换为 Int16 也要求为显式转换，因为 Single 可以表示的数量级比 Int16 所能表示的数量级要大。

注意不同的编译器可能产生不同的代码来处理这些转型操作。例如，当将一个值为 6.8 的 Single 转型为一个 Int32 时，一些编译器产生的代码可能会为一个值为 6 的 Int32，而另一些编译器产生的代码的结果可能为 7。顺便说一句，C#总是会舍弃小数部分。关于 C#中基元类型的转型规则，可参见 C#语言规范中“转换”一节。

注意 如果我们发现自己选择的编译器没有将某些类型作为基元类型来支持，我们可以使用 System.Convert 类型的静态方法来帮助在不同类型的对象之间进行转型。Convert 类型知道怎样在 FCL 中的各种核心类型之间进行转换：Boolean、Char、SByte、Byte、Int16、UInt16、Int32、UInt32、Int64、UInt64、Single、Double、Decimal、DateTime，以及 String。Convert 类型还提供了一个静态的 ChangeType 方法，该方法可以将一个类型转换为另外一个任意的类型，前提是被转换类型实现了 IConvertible 接口，特别是其中的 ToType 方法。

除了转型，基元类型还能够以文本常量的形式出现，看下面的代码：

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

另外，如果我们的表达式包含有文本常量，编译器将能够在编译时计算该表达式，从而提高代码性能。

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

最后，编译器会自动知道怎样解析出现在代码中的操作符(例如+、-、*、/、%、&、^、|、==、!=、>、<、>=、<=、<<、>>、~、!、++、--，等)：

```
Int32 x = 100;
Int32 y = x + 23;
Boolean lessThanFifty = (y < 50);
```

5.1.1 Checked 与 Unchecked 基元类型操作

我们知道许多基元类型的算术运算都会导致结果溢出：

```
Byte b = 100;
b = (Byte) (b + 200);          // b 的值将为 44
```

> **重要** CLR 只在 32 位和 64 位值上进行算术运算。所以，b 和 200 首先会被转换为 32 位的值，然后相加。得到的结果首先是一个 32 位的值，接着必须被转型为一个 Byte，然后才能将其放入变量 b 的存储堆栈内。C#不会隐式进行这样的转型，这也是上面第 2 行代码中需要做 Byte 转型的原因。

大多数情况下，这种暗地里发生的溢出并不是我们所期望的，而且如果没被检测出来的话，它们往往会导致应用程序出现一些奇怪的行为。但是在一些极少数的情况下(例如计算一个散列值或者校验和)，这种溢出不仅是可接受的，而且还是我们所期望的。

不同的语言处理溢出的方式不同。C 和 C++不把溢出认为是一种错误，并且允许对发生溢出的值做“绕回”(wrap)处理。如果发生了溢出，应用程序也会继续运行下去，我们惟一能够做的就是祈祷这种溢出不会造成任何问题。另一方面，默认情况下，Visual Basic 会把整数溢出视为一种错误，并且在检测到溢出时，会抛出一个异常。

CLR 提供的 IL 指令允许编译器选择自己期望的行为。CLR 提供了一个名为 add 的指令会直接对两个数做加法运算，而不做任何溢出检查。

CLR 还提供了一个名为 add.ovf 的指令，它在对两个数做加法运算的时候，一旦发生溢出，便会抛出一个 System.OverflowException 异常。除了这两个执行加法运算的指令外，CLR 还提供了类似的减法(sub/sub.ovf)、乘法(mul/mul.ovf)，以及数据转换(conv/conv.ovf)运算指令。

C#允许开发人员自己决定应该如何处理溢出。默认情况下，溢出检查是关闭的。这意味着编译器产生的 IL 指令中的加、减、乘以及转换运算使用的是不包含溢出检查的版本。这样的结果是代码的效率有所提高——但是我们必须确保不会发生溢出，或者是将代码设计为预期发生这些溢出。

让 C#编译器控制溢出的一种方法是使用/checked+命令行开关。该命令行开关告诉编译器使用带溢出检查的加、减、乘以及转换 IL 指令来产生代码。使用这样的命令行开关将使代码的效率有所降低，这是因为 CLR 需要检查代码中是否会出现溢出。一旦发生溢出，CLR 便会抛出一个 OverflowException 异常。我们应该在代码中注意捕获该异常，并从中顺利地恢复。

相对于对整个代码都打开或者关闭溢出检查而言，开发人员更喜欢按照情况来决定是否进行溢出检查。C#中的 checked 和 unchecked 操作符为我们提供了这种灵活性。看下面的例子。(假设编译器默认情况下使用“不检查溢出”的方式来编译代码。)

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

在上面的代码中，b 和 200 首先被转换为两个 32 位的值，然后又被加在一起。结果是 300。300 然后又被转换为一个 Byte，这时将出现 OverflowException 异常。如果执行 Byte 转型在 checked 操作符外面，那么将不会抛出异常。

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

除了 checked 和 unchecked 操作符外，C#还提供了 checked 和 unchecked 语句。它们可以使整个语句块中的表达式都接受溢出检查，或者都不接受溢出检查。

```
checked {                         // 溢出检查块开始
     Byte b = 100;
     b = (Byte) (b + 200);        // 该表达式将被执行溢出检查
}                                 // 溢出检查块结束
```

实际上，如果使用了 checked 语句，我们就可以使用 += 操作符来简化代码：

```
checked {                          // 溢出检查块开始
    Byte b = 100;
    b += 200;                      // 该表达式将被执行溢出检查
}                                  // 溢出检查块结束
```

重要　因为 checked 操作符和语句只影响加、减、乘以及转换 IL 指令产生的版本，在 checked 操作符或语句内调用一个方法并不会对该方法产生任何影响。

```
checked {
    // 假设 SomeMethod 试图将 400 加载到一个 Byte 中
    SomeMethod(400);
    // SomeMethod 是否抛出 OverflowException 要看其内的代码是
    // 否使用 checked 指令进行编译，与这里的 checked 语句无关
}
```

下面是使用 checked 和 unchecked 时的一些推荐原则：

- 当编写代码时，如果希望在出现溢出时抛出异常，我们就应该显式使用 checked。本书第 18 章将介绍如何使用异常处理，以及怎样从异常中恢复。
- 当编写代码时，即使出现了溢出，我们也不希望有异常抛出，那么就应该显式使用 unchecked。这时我们实际上是希望溢出能够默默地进行。
- 对于没有使用 checked 或 unchecked 的代码来说，这表明在应用程序的开发阶段我们希望出现溢出时能够抛出异常，而在应用程序发布后，便不希望再做溢出检查。

当开发应用程序时，我们应该使用编译器的/checked+命令行开关。这时应用程序运行的速度一般比较慢，因为系统会对任何没有显式标识 checked 或 unchecked 的代码做溢出检查(译注：系统当然也会对显式标识 checked 的代码做溢出检查)。如果有异常出现，我们便可以很容易检测到并及时修复。而当发布应用程序时，我们应该使用编译器的/checked-命令行开关，这样发布出去的代码运行速度会比较快，而且一般不会产生异常。

重要 System.Decimal是一个非常特殊的类型。虽然很多编程语言(包括C#和Visual Basic)都将Decimal看作是一个基元类型，但CLR却不是这样。这意味着CLR没有直接操作Decimal值的IL指令。如果我们查看.NET框架文档中的Decimal类型，我们将会看到其中的几个静态方法，如Add、Subtract、Multiply、Divide等。另外，Decimal类型还为 +、-、*、/ 等提供了操作符重载方法。

当我们编译使用Decimal值的代码时，编译器产生的代码实际上会通过调用Decimal类型的成员来执行相关的操作。这意味着操作Decimal值的代码效率会比操作其他的CLR基元类型的代码效率要低。另外，因为没有操作Decimal值的IL指令，所以checked和unchecked操作符、语句，以及相关的编译器命令行开关对它没有任何影响。如果对Decimal值的操作没有安全地执行，系统总会抛出System.OverflowException异常。

5.2 引用类型与值类型

CLR支持两种类型：引用类型和值类型。在这两者之中，我们和引用类型打交道的机会最多。引用类型(reference type)总是从托管堆上分配，C#的new操作符返回的就是对象位于托管堆中的内存地址——该内存地址指向对象占用的数据位。在使用引用类型时，我们需要有一些性能考虑。首先考虑以下事实：

- 内存必须从托管堆中分配。
- 每个在托管堆中分配的对象都有一些与之关联的额外附加成员必须被初始化。
- 从托管堆中分配对象可能会导致执行垃圾收集。

如果我们代码中的每个类型都是引用类型的话，应用程序的性能将会大打折扣。设想如果每使用一个Int32值，系统都会出现一次这样的内存分配，应用程序的性能该会有多糟糕！

为了提高使用那些简单、常用类型的性能，CLR 提供了一种称作**值类型**(value type)的“轻量级”类型。值类型实例通常分配在线程的堆栈上(虽然它们也可以被嵌入到一个引用类型的对象中)。表示值类型实例的变量不包含指向实例的指针——变量本身即包含了实例所有的字段。因为变量本身包含了实例的所有字段，所以操作实例时也就无需再解析指针引用。值类型实例不受垃圾收集器的控制，因此也减少了托管堆的压力，以及应用程序在整个生存期中需要垃圾回收的次数。

.NET 框架参考文档明确指出了哪些类型是引用类型，哪些类型是值类型。当我们在其中查找一个类型时，任何被称为“类”的类型都是引用类型。例如，System.Object 类、System.Exception 类、System.IO.FileStream 类，以及 System.Random 类都是引用类型。另一方面，文档中将“结构”或者“枚举”称作为值类型。例如，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。

如果我们再进一步查看.NET 框架参考文档，就会发现所有的结构都直接继承自 System.ValueType 类型。System.ValueType 类型本身又直接继承自 System.Object 类型。根据定义，所有的值类型都必须继承自 System.ValueType 类型。

注意 所有的枚举类型都继承自 System.Enum，System.Enum 本身又继承自 System.ValueType。CLR 和所有的编程语言都对枚举类型给予了特殊的对待。关于枚举类型的更多信息，可参见本书第 13 章。

即使我们在定义自己的值类型时不能为其选择任何的基类型，我们仍然可以为一个值类型实现一个或多个接口。另外，CLR 也不允许一个值类型被用作任何其他引用类型或值类型的基类型。例如，我们不可能定义任何把 Boolean、Char、Int32、UInt64、Single、Double、Decimal 等类型用作基类型的新类型。

重要 对于许多开发人员来讲(例如非托管 C/C++开发人员)，引用类型和值类型刚开始好像有些奇怪。在非托管 C/C++中，我们声明一个类型，然后由使用该类型的代码决定类型实例是分配在线程的堆栈上，还是分配在应用程序的堆上。在托管代码中，我们定义的类型决定了类型实例的分配位置，而使用类型的开发人员对此没有控制权。

下面的代码和图 5.1 演示了引用类型和值类型之间的一些差别：

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

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

static void ValueTypeDemo() {

   SomeRef r1 = new SomeRef();          // 分配在托管堆上
   SomeVal v1 = new SomeVal();          // 分配在堆栈上

   r1.x = 5;                            // 解析指针
   v1.x = 5;                            // 在堆栈上修改

   Console.WriteLine(r1.x);             // 显示为 "5"
   Console.WriteLine(v1.x);             // 也显示为 "5"

  // 图 5.1 左边的一幅展示了上面的代码执行后的情形

   SomeRef r2 = r1;                     // 仅拷贝引用(指针)
   SomeVal v2 = v1;                     // 先在堆栈上分配，然后拷贝成员

   r1.x = 8;                            // 改变了 r1.x 和 r2.x
   v1.x = 9;                            // 改变了 v1.x，没有改变 v2.x

   Console.WriteLine(r1.x);             // 显示为 "8"
   Console.WriteLine(r2.x);             // 显示为 "8"
   Console.WriteLine(v1.x);             // 显示为 "9"
   Console.WriteLine(v2.x);             // 显示为 "5"

   // 图 5.1 右边的一幅展示了上面所有的代码执行后的情形
}
```

在上面的代码中，SomeVal 类型被声明为一个 struct，而不是更常见的 class。在 C#中，用 struct 声明的类型被认为是值类型，而用 class 声明的类型是引用类型。我们可以看到，引用类型和值类型有很大的不同。我们在代码中使用一个类型时，一定要清楚该类型是引用类型还是值类型，因为这很大程度上影响着在代码中如何表达我们的意图。

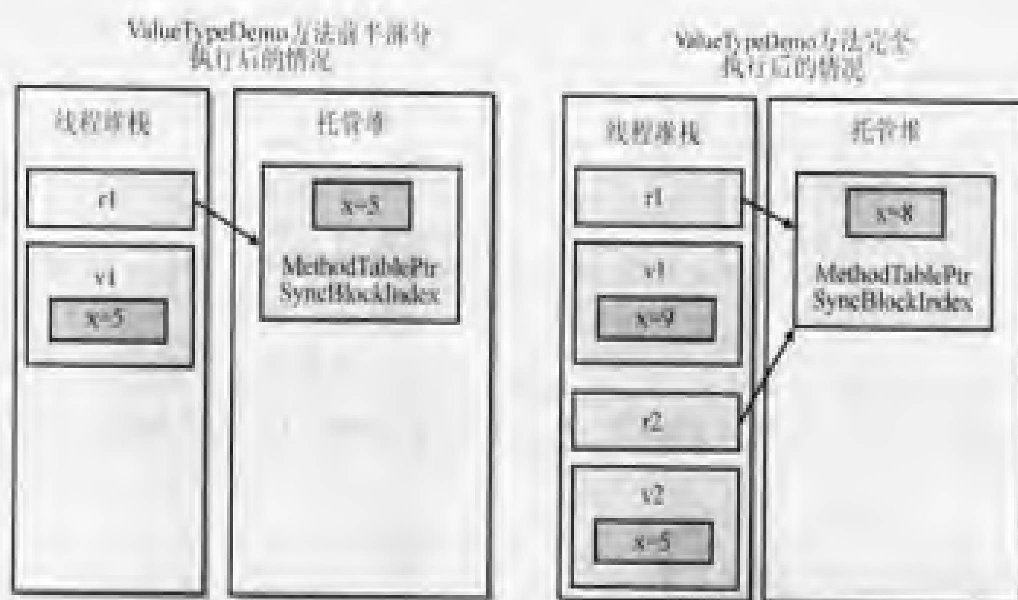


图 5.1 引用类型和值类型内存布局的差别

> **注意** 其他一些语言可能会使用不同的语法来描述值类型和引用类型。例如托管扩展 C++就使用_ _value 修饰符来声明值类型。

在前面的代码中，大家应该会看到下面这行代码：

```
SomeVal v1 = new SomeVal();        // 分配在堆栈上
```

该行代码的编写方式看起来很像是在托管堆上分配了一个 SomeVal 实例。然而，C#编译器知道 SomeVal 是一个值类型，因此会在线程的堆栈上分配 SomeVal 实例。另外，C#还会确保该值类型实例所有的字段都被置为 0(译注：二进制意义上的 0 值)。

上面的代码也可以这样写：

```
SomeVal v1;                        // 分配在堆栈上
```

该行代码也会产生将实例分配在线程堆栈上，并将其所有字段置 0 的 IL 指令。两者惟一的差别是如果我们使用了 new 操作符，那么 C#将认为实例已经得到了初始化。下面的代码会有助于我们的理解：

```
// 下面两行能够通过编译是因为 C#认为
// v1 的所有字段首先被初始化成了 0
SomeVal v1 = new SomeVal();
Int32 a = v1.x;

// 下面两行不能通过编译是因为 C#认为
// v1 的字段没有得到初始化
SomeVal v1;
Int32 a = v1.x;  //  error CS0170: 使用了可能未赋值的字段“x”
```

在设计自己的类型时，大家需要仔细考虑是将它们定义为值类型，还是引用类型。在某些情况下，值类型能够获得更好的性能。尤其是如果以下所有表述都是正确的，我们就应该考虑将类型声明为值类型。

- 该类型的行为类似于基元类型。
- 该类型不需要继承自任何其他类型。
- 该类型不会被任何其他类型继承。
- 该类型的实例不会频繁地用于方法的参数传递。默认情况下，参数以传值的方式传递，这会导致值类型实例中的字段被拷贝，从而损伤应用程序性能。
- 该类型的实例不会作为方法的结果频繁地返回。从方法中返回一个值类型也会导致实例中的字段被拷贝到调用者分配的内存中，因此也会损伤应用程序的性能。
- 该类型的实例不会被频繁地用于诸如 ArrayList、Hashtable 之类的集合中。这些管理一组通用对象集合的类会对值类型实例执行装箱操作，这将导致额外的内存分配，以及额外的内存拷贝操作，从而也会损伤应用程序的性能。(下一节将深入解释装箱与拆箱操作。)

值类型的主要优势在于它们不被分配在托管堆上。当然，相对于引用类型而言，值类型也有自己的一些限制。下面是值类型和引用类型之间的一些差别：

- 值类型对象有两种表示：一种是未装箱(unboxed)形式，一种是装箱(boxed)形式(下一节会有讨论)。引用类型总是装箱形式。

- 值类型都继承自 System.ValueType。System.ValueType 没有在 System.Object 之外定义任何新的方法。但是，System.ValueType 重写了其中的 Equals 方法。当两个 ValueType 对象的字段相互匹配时，Equals 方法会返回 true。另外，System.ValueType 还重写了 GetHashCode 方法，它会使用对象的实例字段并根据一定的算法来产生一个散列码值。当定义自己的值类型时，我们应该重写 Equals 方法和 GetHashCode 方法，为它们提供一个显式的实现。本书第 6 章会介绍 Equals 方法和 GetHashCode 方法。

- 因为我们声明的值类型或引用类型不能以一个值类型作为基类，所以我们不应该向值类型中引入任何新的虚方法(译注：实际上这样做在 C#语言中是非法的)。值类型中更不可以有任何的抽象方法，所有的方法都隐含为密封(sealed)方法(即不能重写)。

- 引用类型变量包含着对象在托管堆中的内存地址。默认情况下，当一个引用类型变量被创建时，它被初始化为 null，表示该引用类型变量目前没有指向一个有效的对象。试图使用 null 引用类型变量会抛出一个 NullReferenceException 异常。相反，值类型变量总是包含一个符合它的类型的值，并且所有的字段都被初始化为 0 值。当访问一个值类型时，不可能产生 NullReferenceException 异常。

- 当将一个值类型变量赋值给另一个值类型变量时，会进行一个“字段对字段”的拷贝。但当将一个引用类型变量赋值给另一个引用类型变量时，只会拷贝内存地址。

- 因为前面一点，两个或多个引用类型变量可以指向托管堆中的同一个对象，这样对一个变量的操作将会影响到其他变量引用的对象。另一方面，每个值类型变量都有一份自己的“对象”数据拷贝，对一个值类型变量的操作不可能影响到另一个。(译注：如果值类型中定义有引用类型字段，对一个值类型变量的操作还是有可能影响到另一个值类型变量的。当然，这本质上还是由于引用类型的特点所导致的。)

- 因为未装箱值类型没有分配在托管堆上，所以一旦定义该类型实例的方法不再处于活动状态，为它们分配的存储空间就会立即释放。这意味着值类型实例在内存被回收时不可能收到任何通知(通过 Finalize 方法)。

注意　实际上，为值类型定义一个 Finalize 方法显得非常古怪，因为该方法只有在其装箱形式的实例上才可能会被调用。因此，许多编译器(包括 C#和 Visual Basic)都不允许我们在值类型中定义 Finalize 方法。虽然 CLR 允许一个值类型定义 Finalize 方法，但是在值类型的装箱实例被执行垃圾收集时，CLR 并不会调用该方法。

CLR 如何控制类型中字段的布局

为了提高性能，CLR 会按照它自己选择的方式来排列类型实例中的字段。例如 CLR 可能会在内存中重新排列对象的字段，以使对象引用可以聚合在一起，并且能够恰当地对齐和包装数据字段。但是，当我们定义一个类型时，我们也可以告诉 CLR 是按我们指定的顺序来存储类型实例的字段，还是以任何它认为合适的顺序重新排列字段。

我们可以在自己定义的类或者结构上应用 System.Runtime.InteropServices.StructLayoutAttribute 特性(attribute)来告诉 CLR 我们期望的行为。根据该特性的构造器，我们可以传入 LayoutKind.Auto 来让 CLR 自己排列字段，或者传入 LayoutKind.Sequential 来让 CLR 保留我们设定的字段布局。如果我们定义类型时没有显式指定 StructLayoutAttribute 特性，编译器将选择它认为最好的布局方式。

微软的 C#编译器为引用类型(类)选择的是 LayoutKind.Auto 布局方式，而为值类型(结构)选择的是 LayoutKind.Sequential 布局方式。很明显，C#编译器小组认为结构常用于和非托管代码互操作的情形，这样类型中的字段就必须按开发人员当初定义的顺序来排列。但是，如果我们正在创建一个和非托管代码互操作没有任何关系的值类型，我们可能希望改变 C#编译器的默认规则。看下面的例子：

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

// 让 CLR 自己为类型排列字段以提高性能
[StructLayout(LayoutKind.Auto)]
struct Point {
    Int32 x, y;
}
```

5.3 值类型的装箱与拆箱

值类型是比引用类型更为轻量级的类型，因为它们没有被分配在托管堆中，不会被执行垃圾收集，也没有指向它们的指针。但是在很多情况下，我们还必须获得一个指向值类型实例的引用。例如，假设我们希望创建一个 ArrayList(一个定义在 System.Collections 命名空间中的类型)对象来保存一组 Point 结构。那么我们的代码看起来可能就像下面这样：

```
// 声明一个值类型
struct Point {
   public Int32 x, y ;
}

class App {
   static void Main() {
      ArrayList a = new ArrayList();
      Point p;                      // 分配一个 Point(不在托管堆上)
      for (Int32 i = 0; i < 10; i++) {
         p.x = p.y = i;             // 初始化值类型的成员
         a.Add(p);                  // 对值类型进行装箱并将得到
                                    // 的引用添加到 ArrayList 上
      }
      ...
   }
}
```

在上面的代码中，每一次迭代循环都会初始化一个 Point 值类型实例，并将其存储在 ArrayList 中。但是让我们再仔细考虑一下。ArrayList 中究竟存放的是什么？是 Point 结构，还是 Point 结构的地址，或者其他完全不同的东西？要得到正确的答案，我们必须查看 ArrayList 类型的 Add 方法，及其定义的参数类型。本例中的 Add 方法原型如下：

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

我们可以看到 Add 接受的参数为一个 Object，这表示 Add 需要一个指向托管堆中对象的引用(或指针)。但是前面的代码传递的却是一个 Point 值类型实例。要使代码正确运行，我们首先必须将 Point 值类型实例转换为一个真正的托管堆上的对象，然后再将指向该对象的引用传递给 Add 方法。

在.NET 框架中，一种称作**装箱**(boxing)的机制用来将一个值类型转换为一个引用类型。装箱操作通常由以下几步组成：

1. 从托管堆中为新生成的引用类型对象分配内存。分配的内存大小为，值类型实例本身的大小加上其他额外的将该值类型实例视为真正的引用对象所需的空间。这些额外的空间包括一个方法表指针和一个SyncBlockIndex。

2. 将值类型实例的字段拷贝到托管堆上新分配对象的内存中。

3. 返回托管堆中新分配对象的地址。该地址就是一个指向对象的引用。值类型实例也就变成了一个引用类型对象。

某些编译器(例如 C#)会根据需要自动产生对值类型实例进行装箱的 IL 代码。但我们仍然需要了解系统在后台到底做了些什么，因为只有这样我们才会清楚应用程序的代码大小以及一些相关的性能问题。

在前面的代码中，C#编译器检测到我们将一个值类型实例传递给了一个需要引用类型参数的方法后，它会自动产生装箱操作的指令。这样在运行时，Point 值类型实例(p)中的字段就会被拷贝到新分配的 Point 对象中。随后，这个经过装箱的 Point 对象(现在是一个引用类型)的地址被返回并传递给 Add 方法。该 Point 对象将驻留在托管堆中直到最后被垃圾收集器回收。而 Point 值类型实例(p)现在则可以被重用或者释放，因为 ArrayList 对它一无所知！注意，经过装箱的值类型实例的生存期超出了未装箱的值类型实例的生存期。

许多面向 CLR 的语言(如 C#和 Visual Basic)在必要的时候都会自动产生代码来将值类型实例装箱为引用类型。然而，也有一些语言(例如托管扩展 C++)要求开发人员编写显式装箱值类型的代码。

大家已经知道了装箱操作是如何执行的，下面我们来谈谈拆箱操作。假设我们在另一段代码中希望能够获取前面 ArrayList 对象中的第 1 个元素：

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

这里，我们首先得到的是包含在 ArrayList 中第 1 个元素的引用(指针)，然后我们又试图将它赋值给一个 Point 值类型(p)。要使这段代码正确运行，所有包含在已装箱的 Point 对象中的字段都必须被拷贝到值类型变量 p 中，其中值类型变量 p 位于当前线程的堆栈上。CLR 通过两个步骤来完成这样的拷贝操作，它首先获取已装箱 Point 对象中、属于 Point 值类型的那部分字段的地址。这个过程称作**拆箱**(unboxing)。然后，它又将这些字段的值从托管堆拷贝到位于线程堆栈上的值类型实例中。

拆箱和装箱并不是严格意义上的互反操作。拆箱操作的代价要比装箱操作小许多。拆箱操作仅仅是获取指向对象中包含的值类型部分(数据字段)的指针而已，它不会像装箱操作那样涉及到任何内存字节的拷贝。然而，紧接着拆箱之后典型的操作往往就是字段拷贝，这两个操作合起来与装箱操作才成为真正的互反操作。

很明显，装箱和拆箱/拷贝操作会从速度和内存两方面损伤应用程序的性能。因此我们应该清楚编译器会在何时自动产生执行这些操作的指令，并使我们编写的代码尽可能地减少导致这种情况的机会。

对一个引用类型的拆箱操作通常由以下几步组成：

1. 如果该引用为null，将会抛出一个NullReferenceException异常。
2. 如果该引用指向的对象不是一个期望的值类型的已装箱对象，将会抛出一个InvalidCastException异常。
3. 一个指向包含在已装箱对象中值类型部分的指针被返回。该指针指向的值类型对于引用类型对象通常所具有的附加成员(即一个方法表指针和一个SyncBlockIndex)一无所知。实际上，该指针指向的是已装箱对象中的未装箱部分。

其中上面的第2项意味着下面的代码不会如我们所期望的那样运行：

```
static void Main() {
   Int32 x = 5;
   Object o = x;              // 对x进行装箱；o指向已装箱对象
   Int16 y = (Int16) o;       // 抛出一个InvalidCastException异常
}
```

从逻辑上讲，将o指向的已装箱Int32对象转型为一个Int16值类型是有意义的。但是，当对一个对象执行拆箱操作时，转型的结果必须是它原来未装箱时的类型——本例中即为Int32。下面的代码演示了正确的做法：

```
static void Main() {
   Int32 x = 5;
   Object o = x;                    // 对x进行装箱；o指向已装箱对象
   Int16 y = (Int16)(Int32) o;      // 先拆箱为正确的类型，然后再转型
}
```

严格地讲，拆箱操作不会拷贝任何字段。但通常情况下，拆箱操作后会紧跟着一个字段拷贝操作，将字段从托管堆拷贝到堆栈中。实际上，在C#中，拆箱操作总是紧跟着一个字段拷贝操作。看下面的代码：

```
static void Main() {
   Point p;
   p.x = p.y = 1;
   Object o = p;          // 对p进行装箱；o指向已装箱对象

   p = (Point) o;         // 对o进行拆箱，并将字段从托管堆拷贝到堆栈上
}
```

对于上面最后一行代码，C#编译器会产生一个对o执行拆箱操作的IL指令(获取对象中属于值类型部分的字段地址)，以及另一个将字段从托管堆拷贝到堆栈变量p上的IL指令。

再来看下面一段代码：

```
static void Main() {
    Point p;
    p.x = p.y = 1;
    Object o = p;          // 对p进行装箱；o指向已装箱对象

    // 将Point的x字段改为2
    p = (Point) o;         // 对o进行拆箱，并将字段从托管堆拷贝到堆栈上
    p.x = 2;
    o = p;
}
```

其中后一部分代码的目的仅仅是希望将Point的x字段从1改为2。要实现这一点，首先需要执行一次拆箱操作，紧接着再执行一次字段拷贝操作，然后在堆栈上修改字段的值，最后再执行一次装箱操作(该操作将在托管堆上创建一个全新的对象)。希望大家由此能够看到装箱和拆箱/拷贝操作对应用程序性能所产生的影响。

一些语言(如托管扩展C++)允许我们在对一个已装箱的值类型实例执行拆箱操作时，不必拷贝其中的字段。这些拆箱操作将直接返回已装箱对象的值类型部分(忽略额外的方法表指针和SyncBlockIndex成员)的地址。我们可以使用该指针来操作其值类型部分的字段(这样的操作发生在托管堆中已装箱的对象上)。所以，前面的代码如果用托管扩展C++来编写，效率将会有所提升。因为我们将能够直接在已装箱的Point对象上改变其x字段。这样以来，在托管堆上分配新对象，以及两次拷贝所有字段的重复操作都将可以被避免。

重要 如果大家对应用程序的性能有所担心，那么首先必须搞清楚编译器何时会产生执行这些操作的指令。一些语言(如托管扩展C++)要求我们显式编写代码来对值类型实例执行装箱和拆箱操作(托管扩展C++中的__box操作符和dynamic_cast操作符就为我们提供了这样的功能)。一方面，这种要求增加了开发人员的工作难度，因为需要编写的代码因此增多了。另一方面，通过显式编写代码，开发人员将能够确切地知道何时会出现装箱和拆箱操作。然而，许多语言(例如C#和Visual Basic)都会自动产生必要的IL代码来执行装箱和拆箱操作。虽然这种内置的功能使得代码看起来更加漂亮，也减轻了开发人员的负担，但这也意味着开发人员难以清楚代码中何时会出现装箱和拆箱操作。

让我们再来看几个演示装箱和拆箱操作的例子：

```
public static void Main() {
  Int32 v = 5;                               // 创建一个未装箱的值类型变量
  Object o = v;                              // o 指向一个已装箱的、值为 5 的 Int32
  v = 123;                                   // 将未装箱值改为 123
  Console.WriteLine(v + ", " + (Int32) o);     // 显示"123, 5"
}
```

猜一下在这段代码中有多少次装箱操作？如果答案是 3 次，请不要感到奇怪。下面我们来仔细分析一下这段代码。为了帮助大家理解，下面附上 Main 方法编译后产生的 IL 代码，以及我对它们所做的详尽的注释。(译注：下面 IL 代码中 Main 方法的保护限制应该为 public，而非 private。下同。)

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      46 (0x2e)
    .maxstack 3
    .locals ( [0] int32 v,
              [1] object o)

    // 将 5 加载到 v 中
    IL_0000: ldc.i4.5
    IL_0001: stloc.0
    // 对 v 执行装箱，并将得到的指针存放在 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 执行装箱，并将得到的指针存放在堆栈上等待 Concat 操作
    IL_000c: ldloc.0
    IL_000d: box [mscorlib]System.Int32

    // 将字符串加载到堆栈上等待 Concat 操作
    IL_0012: ldstr    ", "

   // 对 o 执行拆箱：将指向已装箱对象内部的 Int32 字段的指针置于堆栈上
    IL_0017: ldloc.1
    IL_0018: unbox [mscorlib]System.Int32

    // 将已装箱的 Int32 中的字节拷贝到堆栈上
    IL_001d: ldind.i4
```

```
    // 对 Int32 值进行装箱操作，并将得到的指针放在堆栈上等待 Concat 操作
    IL_001e: box [mscorlib]System.Int32
    // 调用 Concat
    IL_0023: call string [mscorlib]System.String::Concat(   object,
                                                             object,
                                                             object)
    // 将 Concat 返回的字符串传递给 WriteLine
    IL_0028: call void [mscorlib]System.Console::WriteLine(string)
    IL_002d: ret
  } // end of method App::Main
```

上面的代码中首先创建了一个未装箱的 Int32 值类型(v)，并将其初始化为 5。然后，又创建了一个 Object 引用类型(o)，并希望将其指向 v。但是因为引用类型必须指向托管堆中的对象，所以 C#会产生适当的 IL 代码将 v 进行装箱，并将 v 的装箱“拷贝”的地址存放在 o 中。随后 123 被放入未装箱的值类型 v 中，这对已装箱的 Int32 值没有任何影响，它的值仍为 5。

接着，代码中调用了 WriteLine 方法。WriteLine 方法期望的参数为一个 String 对象，但代码中并没有这样的 String 对象，相反，代码中只有 3 个对象：一个未装箱的 Int32 值类型(v)、一个 String(引用类型)，以及一个指向已装箱的 Int32 值类型(o)的引用，并且这个已装箱的值类型正在被转型为一个未装箱的 Int32 类型。这 3 个对象必须被组合为一个 String 对象。

为了将这 3 个对象组合为一个 String 对象，C#编译器会产生代码来调用 String 的静态 Concat 方法。有好几个重载版本的 Concat 方法，它们做的事情都是一样的——仅有的差别在于参数的个数。因为是由 3 个对象连接而生成的字符串，所以 C#编译器选择调用的 Concat 方法应为以下签名：

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

首先 v 作为第 1 个参数被传递给 arg0。但是 v 是一个未装箱的值类型参数，而 arg0 是一个 Object，所以 v 必须被装箱，装箱后的 v 的地址才能被传递给 arg0。然后，字符串“,”作为第 2 个参数(一个 String 对象引用)被传递给 arg1。对于最后一个参数 arg2，变量 o(一个 Object 引用)首先被转型为一个 Int32，这会导致在 IL 的演算堆栈(evaluation stack)(译注：IL 汇编语言是一门基于堆栈的语言，这个堆栈指的就是 IL 的演算堆栈，它是方法状态的一部分。IL 的大多数指令或者是向演算堆栈上加载数据，或者是从演算堆栈上获取数据，其操作顺序遵循堆栈先进先出 FIFO 的原则)上创建一个临时的 Int32 值类型实例，该实例接受一份 o 当前所引用的对象的未装箱部分的拷贝。最后，这个临时的 Int32 值类型变量还必须被装箱，装箱后的内存地址才会被传给 Concat 方法的 arg2 参数。

Concat 方法调用每个传入对象的 ToString 方法，然后连接它们的字符串表示。从 Concat 方法返回的 String 对象然后被传递给 WriteLine 方法，最终显示在控制台上。

除了上面的做法外，我们还可以像下面这样调用 WriteLine 方法来提升代码的性能：

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

这里和前面的区别仅在于我们去掉了对变量 o 所进行的转型操作。这样的代码将变得更加高效，因为 o 已经是一个 Object 引用类型，因此它可以被直接传递给 Concat 方法。实际上，转型的移除为代码节省了以下 3 项操作：一次拆箱操作、一次字段拷贝操作，以及一次装箱操作。我们可以通过查看重新编译后产生的 IL 代码来看到这一点：

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 35 (0x23)
    .maxstack 3
    .locals ( [0] int32 v,
              [1] object o)

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

    // 对 v 执行装箱，并将得到的指针存放在 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 执行装箱，并将得到的指针存放在堆栈上等待 Concat 操作
    IL_000c: ldloc.0
    IL_000d: box [mscorlib]System.Int32

    // 将字符串加载到堆栈上等待 Concat 操作
    IL_0012: ldstr    ", "

    // 将已装箱 Int32 值的地址加载到堆栈上等待 Concat 操作
    IL_0017: ldloc.1

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

```
    // 将 Concat 返回的字符串传递给 WriteLine
    IL_001d: call void [mscorlib]System.Console::WriteLine(string)
    IL_0022: ret
} // end of method App::Main
```

对比前后两段代码，我们可以很容易看到不含 Int32 转型的 IL 代码量要比含有转型的少 11 个字节，这显然是由于额外的拆箱和装箱操作产生了更多的代码。然而，更值得注意的是其中的装箱操作，它会在托管堆上分配一个额外的对象，而该对象在将来某一时刻必须被执行垃圾收集。当然，上述两个版本的代码产生的结果是一样的，而且性能差别也微不足道。但如果这种不必要的装箱操作出现在一个循环中，那么应用程序的性能和内存的使用就会受到比较大的影响。

实际上，在上面代码的基础上，我们还可以做进一步的优化：

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

这段代码在值类型变量 v 上调用了 ToString 方法，然后返回一个 String 引用。由于 String 对象已经是一个引用类型，所以我们可以将其直接传递给 Concat 方法，而不需要做任何装箱操作。

我们再来看一个演示装箱和拆箱操作的例子：

```
public static void Main() {
    Int32 v = 5;                     // 创建一个未装箱值类型变量
    Object o = v;                    // o 指向 v 的装箱版本

    v = 123;                         // 将未装箱值类型改为 123
    Console.WriteLine(v);            // 显示 "123"

    v = (Int32) o;                   // 对 o 执行拆箱并赋值给 v
    Console.WriteLine(v);            // 显示 "5"
}
```

大家猜一下这段代码有多少次装箱操作？答案是 1。原因是 System.Console 类定义了一个接受 Int32 类型作为参数的 WriteLine 方法。

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

在上面两次调用 WriteLine 方法的过程中，变量 v(一个未装箱值类型实例)以传值的方式被传递。当然，WriteLine 方法也许在内部对这个 Int32 值执行了装箱操作，但这不属于我们的控制范围。重要的是我们已经尽可能地减少了自己代码中的装箱操作。

如果我们再进一步查阅 FCL，会发现许多重载方法的差别就在它们的值类型参数上。例如 System.Console 类就提供了好几个重载的 WriteLine 方法，下面列出了其中的一些：

```
public static void WriteLine(Boolean);
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);
```

大家还会发现其他一些类型也有着类似的重载方法，如 System.Console 类的 Write 方法，System.IO.BinaryWriter 类的 Write 方法，System.IO.TextWriter 类的 Write 和 WriteLine 方法，System.Runtime.Serialization.SerializationInfo 类的 AddValue 方法，System.Text.StringBuilder 类的 Append 和 Insert 方法，等等。所有这些方法都提供了重载的版本，其惟一的目的就是减少一些常用值类型的装箱操作。

关于装箱操作最后需要注意的一点是，如果我们知道自己编写的代码会导致编译器反复地对一个值类型进行装箱，那么我们应该自己来做这样的装箱操作，因为这将能够减少生成的代码量，并提高代码运行速度，看下面的例子：

```
using System;

class App {
   static void Main() {
      Int32 v = 5;        // 创建一个未装箱值类型变量

#if INEFFICIENT
      // 当编译下面的代码时，v 将被装
      // 箱 3 次，非常浪费时间和内存
      Console.WriteLine("{0}, {1}, {2}", v, v, v);
#else
      // 下面的代码产生同样的结果，但
      // 执行速度更快，使用的内存也更少
      Object o = v;          // 手动对 v 装箱(仅一次)

      // 编译下面的代码将不会再出现装箱
      Console.WriteLine("{0}, {1}, {2}", o, o, o);
#endif
   }
}
```

如果上面的代码编译时定义了 INEFFICIENT 符号，那么编译器产生的代码将对 v 执行 3 次装箱，从而导致在托管堆上分配 3 个对象！这显然非常浪费，因为 3 个对象有着相同的值：5。如果上面的代码编译时没有定义 INEFFICIENT 符号，那么编译器产生的代码将对 v 仅执行一次装箱，这只会在托管堆上分配一个对象。最后这个经过装箱的 Int32 值分 3 次被传递给 Console.WriteLine 方法。显然，第二种实现执行的速度要快许多，并且从托管堆中分配的内存也更少。

在上面这些例子中，我们一般可以很容易识别出什么时候需要对一个值类型实例进行装箱。基本上来讲，如果我们希望有一个指向值类型实例的引用，该实例就必须被装箱。通常这种情况是因为我们有一个值类型，而又希望将它传递给一个需要引用类型的方法。然而，这并不是需要对值类型实例进行装箱的惟一情况。

大家回想一下未装箱值类型作为轻量级类型(相对于引用类型而言)的两个原因：

- 它们不被分配在托管堆上。
- 它们没有托管堆上的对象都有的额外的附加成员：一个方法表指针和一个 SyncBlockIndex。

因为未装箱值类型没有 SyncBlockIndex，所以我们不可能利用 System.Threading.Monitor 类型来同步多个线程对它们的访问。因为未装箱值类型没有方法表指针，所以我们也不可能通过值类型的未装箱实例来调用其上继承而来的虚方法。另外，将一个未装箱的值类型实例转型为一个该类型实现的接口类型也需要对该实例进行装箱，因为接口类型总是引用类型。(第 15 章将探讨接口)。看下面的代码：

```
using System;

struct Point : ICloneable {
   public Int32 x, y;

   // 重写从 System.ValueType 继承来的 ToString 方法
   public override String ToString() {
      return String.Format("({0}, {1})", x, y);
   }

   // 实现 ICloneable 接口的 Clone 方法
   public Object Clone() {
      return MemberwiseClone();
   }
}
```

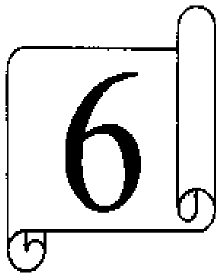
```
class App {
    static void Main() {
        // 在堆栈上创建一个 Point 值类型实例
        Point p;

        // 初始化该实例的字段
        p.x = 10;
        p.y = 20;

        // p 在调用 ToString 时不会被执行装箱
        Console.WriteLine(p.ToString());

        // p 在调用 GetType 时会被执行装箱
        Console.WriteLine(p.GetType());

        // p 在调用 Clone 时不会被装箱。
        // 但 Clone 返回的对象会被执行拆箱，
        // 拆箱后的字段值被拷贝到 p2 中
        Point p2 = (Point) p.Clone();

        // p2 被装箱，装箱后的引用被存放在 c 中
        ICloneable c = p2;

        // c 不会被装箱，因为它已经被装箱。
        // Clone 返回的对象引用被保存在 o 中
        Object o = c.Clone();

        // o 被拆箱，拆箱后的字段值被拷贝到 p 中
        p = (Point) o;
    }
}
```

上面的代码演示了几个与装箱、拆箱操作相关的情形：

- **调用 ToString** 在调用 ToString 时，p 不必被装箱。大家刚开始可能会认为 p 会被装箱，因为 ToString 是一个继承自 System.ValueType 的方法。正常情况下，调用一个从基类继承而来的方法，我们需要一个指向其类型方法表的指针——而 p 是一个未装箱的值类型，它没有指向 Point 方法表的指针。但是，C#编译器发现 Point 重写了 ToString 方法，它将产生直接调用 ToString 的指令。因为 Point 是一个值类型，而值类型不会被用作任何其他类型的基类型，所以编译器知道这里不会出现多态行为。

- **调用 GetType** 在调用 GetType 时，p 必须被装箱。原因是 Point 类型没有提供 GetType 方法的实现，它直接继承了 System.ValueType 的 GetType 方法。所以调用 GetType 时，我们必须有一个指向 Point 方法表的指针，这只能通过对 p 执行装箱来获得。
- **调用 Clone(第 1 次)** 在第 1 次调用 Clone 时，p 不必被装箱，因为 Point 实现了 Clone 方法，编译器可以直接调用它。注意 Clone 返回的是一个 Object，它是一个指向托管堆上已装箱的 Point 对象的引用。要将其内的字段拷贝到未装箱的值类型 p2 中，该对象必须被执行拆箱。
- **转型为 ICloneable** 当将 p2 转型为一个接口类型时，p2 必须被装箱，因为根据定义接口是一种引用类型。装箱后的指针被存放在变量 c 中。
- **调用 Clone(第 2 次)** 第 2 次调用 Clone 也不会出现装箱操作，Clone 方法将直接在托管堆中已装箱的对象上被调用。Clone 在托管堆上创建一个新的对象，并返回指向这个新对象的引用。该引用随后被保存在变量 o 中(一个引用类型)。
- **转型为 Point** 当将 o 转型为 Point 时，托管堆中被 o 引用的对象会被执行拆箱操作，其中的字段会被从托管堆拷贝到变量 p 中，其中 p 是一个驻留在堆栈上的 Point 值类型实例。

大家刚开始可能会对本章有关引用类型、值类型，以及装箱/拆箱操作的内容感到有些迷惑。但是，对这些概念坚实的理解是每一个.NET 框架开发人员获得长期成功的关键。只有深度把握住这些概念，才能更加快捷、轻松地创建出高效的应用程序来。



通用对象操作

本章为大家描述怎样正确实现所有对象都必须提供的一组通用操作。这些操作涉及到的主题包括对象的等值性、惟一性、散列码，以及克隆共4个方面。

6.1 对象的等值性与惟一性

如前所述，System.Object 类型提供了一个名为 Equals 的虚方法，其目的为判断两个对象是否有着相同的“值”。微软的.NET 框架类库(FCL)中包括的许多方法(例如 System.Array 的 IndexOf 方法、System.Collections.ArrayList 的 Contains 方法)在内部都调用到了 Equals 方法。因为 Equals 方法定义在 Object 中，而每个类型最终都派生自 Object，所以我们可以保证每个类型的实例都有一个这样的 Equals 方法。对于那些没有显式重写 Equals 方法的类型，Object(或者重写了 Equals 方法的最近的那个基类)(译注：一个类型的基类的“近”或“远”是针对此基类在该类型所有的基类构成的继承体系中的相对位置而言的，越靠近 Object 的类型的位置越远，反之则越近)提供的实现将被继承。下面的代码展示了 System.Object 类型中的 Equals 方法实现：

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

      // 如果两个引用指向的是同一个对象，
      // 它们肯定相等
      if (this == obj) return(true);

      // 假定两个对象不相等
      return(false);
   }
   ...
}
```

如我们所见，该方法采取的策略可能是最简单的：如果进行比较的两个引用指向的是同一个对象，方法将返回 true；否则在任何其他情况下，方法都将返回 false。如果我们定义了自己的类型，并且希望比较它们中的字段是否相等，Object 类型提供的默认实现对我们来说是不够的，我们必须重写 Equals 方法，提供自己的实现。

当实现自己的 Equals 方法时，我们必须确保它遵循以下 4 条规则：

- Equals 方法必须是自反的，也就是说，x.Equals(x)必须返回 true。
- Equals 方法必须是对称的，也就是说，x.Equals(y)和 y.Equals(x)必须返回同样的值。
- Equals 方法必须是可传递的，也就是说，如果 x.Equals(y)和 y.Equals(z)都返回 true，那么 x.Equals(z)也必须返回 true。
- Equals 方法必须是前后一致的，也就是说，如果两个对象的值没有发生改变，多次调用 Equals 方法的返回值应该相同。

如果我们为 Equals 方法提供的实现没有遵循上述 4 条规则，我们的应用程序将会发生一些奇怪的、不可预期的行为。

不幸的是，重写 Object 的 Equals 方法并不如想象的那么容易。我们必须执行许多步操作，并且要保证每一步操作都是正确的。另外，根据我们定义的类型的不同，这些操作也会有一些差别。所幸的是，实现 Equals 方法只有 3 种不同的方式。下面我们逐一讨论每一种模式。

6.1.1 为基类没有重写 Object.Equals 方法的引用类型实现 Equals

对于那些直接继承了 Object 的 Equals 实现的类型(译注：这句话准确的说法应该为“对于那些基类型直接继承了 Object 的 Equals 实现的类型”)，下面的代码展示了怎样为它们实现 Equals 方法：

```
// 这是一个引用类型('class'的缘故)
class MyRefType : BaseType {
   RefType refobj;        // 该字段是一个引用类型
   ValType valobj;        // 该字段是一个值类型

   public override Boolean Equals(Object obj) {
      // 因为 'this' 不为 null，所以如果 obj 为 null，
      // 那么两个对象将不可能相等
      if (obj == null) return false;

      // 如果两个对象的类型不同，那么它们不可能相等
      if (this.GetType() != obj.GetType()) return false;

         // 将 obj 转型为定义的类型以访问其中的字段。注意这里
         // 的转型不会失败，因为已经知道两个对象是同一个类型
      MyRefType other = (MyRefType) obj;
```

```
        // 比较其中的引用类型字段
        if (!Object.Equals(refobj, other.refobj)) return false;

        // 比较其中的值类型字段
        if (!valobj.Equals(other.valobj)) return false;

        return true;   // 到这里两个对象才算相等
    }

    // 重载 == 和!= 操作符(可选)
    public static Boolean operator==(MyRefType o1, MyRefType o2) {
        return Object.Equals(o1, o2);
        }

    public static Boolean operator!=(MyRefType o1, MyRefType o2) {
        return !(o1 == o2);
    }
}
```

这里实现的 Equals 首先将 obj 和 null 相比较。如果被比较的对象不为 null，那么接着比较两个对象的类型。如果两个对象的类型不同，那么它们不可能相等。如果两个对象有着相同的类型，就将 obj 转型为 MyRefType，这里的转型不可能抛出异常，因为我们已经知道两个对象为同一个类型。等上述所有步骤都正确执行完毕后，我们才开始比较两个对象中的字段。如果两个对象中所有的字段都相等，方法将返回 true。

在比较两个对象中的字段时，我们必须非常仔细。前面的代码展示了根据字段类型的不同，所进行的两种不同的比较方式。

- **比较引用类型的字段**　要比较引用类型的字段，我们应该调用 Object 的静态 Equals 方法。Object 的静态 Equals 方法是一个比较两个引用类型对象的辅助方法。下面展示了 Object 的静态 Equals 方法的内部实现：

```
public static Boolean Equals(Object objA, Object objB) {
    // 如果 objA 和 objB 指向的是同一个对象，方法返回 true
    if (objA == objB) return true;

    // 如果 objA 或者 objB 为 null，它们不可能相等，方法返回 false
    if ((objA == null) || (objB == null)) return false;

    // 判断 objA 和 objB 是否相等，返回比较结果
    return objA.Equals(objB);
}
```

我们采用这种方法来比较引用类型字段是因为即使两个字段出现了值为null的情况，我们的代码仍会正常运行。例如，如果refobj为null，调用refobj.Equals(other.refobj)将会抛出NullReferenceException异常。Object的静态Equals这一辅助方法会为我们对出现null的情况做正确的检测。

- **比较值类型字段** 要比较两个值类型字段，我们应该调用该字段类型的Equals方法来比较它们。我们不应该调用Object的静态Equals方法，因为值类型对象的值永远不可能为null，并且调用Object的静态Equals方法会对值类型对象执行装箱操作。

6.1.2 为基类重写了Object.Equals方法的引用类型实现Equals

对于那些继承了非Object.Equals方法实现的引用类型(译注：这句话准确的说法应该为“对于那些基类型提供了非Object.Equals方法实现的引用类型)，下面的代码展示了怎样为它们实现Equals方法：

```
// 这是一个引用类型('class'的缘故)
class MyRefType : BaseType {
      RefType refobj;          // 该字段是一个引用类型
      ValType valobj;          // 该字段是一个值类型

      public override Boolean Equals(Object obj) {
         // 首先让基类型比较其中的字段
         if (!base.Equals(obj)) return false;

         // 以下所有的代码和前一节中的实现相同

         // 因为 'this' 不为 null，所以如果 obj 为 null，
         // 那么两个对象将不可能相等。
         // 注意：如果确信基类型已经正确地实现了 Equals，
         // 下面一行可以删除
         if (obj == null) return false;

         // 如果两个对象的类型不同，那么它们不可能相等
         // 注意：如果确信基类型已经正确地实现了 Equals，
         // 下面一行可以删除
         if (this.GetType() != obj.GetType()) return false;

         // 将 obj 转型为定义的类型以访问其中的字段。注意这里
         // 的转型不会失败，因为已经知道两个对象是同一个类型
         MyRefType other = (MyRefType) obj;
```

```
            // 比较其中的引用类型字段
            if (!Object.Equals(refobj, other.refobj)) return false;

            // 比较其中的值类型字段
            if (!valobj.Equals(other.valobj)) return false;

            return true;    // 到这里两个对象才算相等
        }

        // 重载 == 和!= 操作符(可选)
        public static Boolean operator==(MyRefType o1, MyRefType o2) {
            Object.Equals(o1, o2);
        }

        public static Boolean operator!=(MyRefType o1, MyRefType o2) {
        return !(o1 == o2);
        }
}
```

这段代码和前面一节中展示的代码大体上是一样的。惟一的差别是这里还要求比较基类型中定义的字段。如果基类型认为对象不相等，那么它们就不可能相等。

如果调用 base.Equals 会导致调用 Object.Equals 方法，那么就不应该再调用它，这一点很重要。因为只有在两个引用指向同一个对象时，Object.Equals 方法才会返回 true。如果两个引用没有指向同一个对象，那么它将返回 false，这样我们实现的 Equals 方法将总是返回 false!

当然，如果我们定义的类型直接继承自 Object，我们就应该像前一节中展示的那样来实现 Equals。如果我们定义的类型不是直接继承自 Object，我们必须首先确定该类型的基类型(除 Object 之外的任何基类型)是否重写了 Equals 方法。如果其中任何一个基类型重写了 Equals 方法，那么我们就应该像本节所示的那样首先调用 base.Equals 方法。

6.1.3 为值类型实现 Equals 方法

第 5 章曾经说过，所有的值类型都继承自 System.ValueType。ValueType 重写了 System.Object 提供的 Equals 方法实现。System.ValueType.Equals 方法在内部首先使用反射机制(本书第 20 章将予以探讨)来得到类型所有的实例字段(译注：尽管这样的 Equals 方法是在 ValueType 中实现的，但是反射机制本身可以保证它永远能够反射出实际类型——也就是继承自 ValueType 的值类型——中定义的所有字段，这里有一个“预知”的意思。当然这样的“预知”是有代价的)，然后再比较它们是否相等。这种比较的过程效率很低，但却是一个所有的值类型都能继承的、相当不错的默认实现。至少，这样的做法意味着引用类型继承的 Equals 判断的是引用相等，而值类型继承的 Equals 判断的是值相等。

对于没有显式重写 Equals 方法的值类型，ValueType 提供的实现将被继承。下面的代码展示了 System.ValueType.Equals 方法的内部实现。

```
class ValueType {
    public override Boolean Equals(Object obj) {

        // 因为 'this' 不为 null，所以如果 obj 为 null，
        // 那么两个对象将不可能相等
        if (obj == null) return false;

        // 得到 'this' 的类型
        Type thisType = this.GetType();

        // 如果'this' 和 'obj'是不同的类型，它们不可能相等
        if (thisType != obj.GetType()) return false;

        // 取得该类型所有的公有和私有实例字段
        //(译注：准确地讲应该是取得类型的所有实例字段)
        FieldInfo[] fields = thisType.GetFields(BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.Instance);

        // 比较每个实例字段是否相等
        for (Int32 i = 0; i < fields.Length; i++) {

            // 从两个对象中得到字段的值
            Object thisValue = fields[i].GetValue(this);
            Object thatValue = fields[i].GetValue(obj);

            // 如果字段值不相等，对象也不可能相等
            if (!Object.Equals(thisValue, thatValue)) return false;
        }
        // 如果其中所有的字段值都相等，那么两个对象将被认为相等
        return true;
    }
    ...
}
```

虽然 ValueType 已经提供了一个相当好的 Equals 实现，并且也适用于我们定义的绝大多数值类型，但我们仍然应该提供自己的 Equals 实现。原因是我们自己的实现执行起来效率较高，并且可以避免额外的装箱操作。下面的代码展示了怎样为一个值类型实现 Equals 方法：

```
// 这是一个值类型（'struct'的缘故）
struct MyValType {
    RefType refobj;        // 该字段是一个引用类型
    ValType valobj;        // 该字段是一个值类型
```

```
    public override Boolean Equals(Object obj) {
        // 如果 obj 不是我们定义的类型，那么两个对象不可能相等
        if (!(obj is MyValType)) return false;

        // 调用类型安全的那个重载版本
        return this.Equals((MyValType) obj);
    }

    // 实现一个强类型版本的 Equals
    public Boolean Equals(MyValType obj) {
        // 比较引用类型字段
        if (!Object.Equals(this.refobj, obj.refobj)) return false;

        // 比较值类型字段
        if (!this.valobj.Equals(obj.valobj)) return false;

        return true;   // 到这里两个对象才算相等
    }

    // 重载 == 操作符(可选)
    public static Boolean operator==(MyValType v1, MyValType v2) {
        return (v1.Equals(v2));
    }

    // 重载 != 操作符(可选)
    public static Boolean operator!=(MyValType v1, MyValType v2) {
        return !(v1 == v2);
    }
}
```

对于值类型，我们应该为它定义一个强类型版本的 Equals 方法，让其接受定义类型作为参数。这样做不仅可以提供类型安全，而且还可以避免额外的装箱操作。除此之外，我们也应该为 == 和 != 提供强类型的重载操作符。下面的代码演示了怎样判断两个值类型是否相等：

```
MyValType v1, v2;

// 下面的代码调用强类型版本的 Equals
// (不会发生装箱操作)
if (v1.Equals(v2)) { ... }

// 下面的代码调用接受 Object 参数
// 的那个 Equals(4 将被装箱)
if (v1.Equals(4)) { ... }
```

```
// 下面的代码不能通过编译，因为操作符 ==
// 不接受 MyValType 和 Int32 作为操作数
if (v1 == 4) { ... }

// 下面的代码能够通过编译，并且不会发生装箱操作
if (v1 == v2) { ... }
```

在强类型版本的 Equals 方法内部，比较字段相等的方式和引用类型中 Equals 方法中比较的方式是完全相同的。注意这里的代码没有进行任何转型，它既不会比较两个实例的类型，也不会调用基类型的 Equals 方法。没有必要进行这些操作是因为方法的参数已经确保两个实例是同样的类型。(译注：实际上这种做法并不能绝对确保两个实例都是同样的类型，因为如果 MyValType 和别的类型之间定义了隐式转换，它们将有可能被判为相等，而它们的类型却不相同！而参数为 System.Object 的那个 Equals 方法虽然会对传入的参数执行装箱操作，但却永远不会丢失类型信息。值得指出的是即使我们在强类型版本的 Equals 方法开始处加上“if (!(obj is MyValType)) return false;”一句也没有作用，因为隐式转换是在传入参数之前就完成了。所以万无一失的做法是不要实现强类型版本的 Equals——不管是引用类型还是值类型。当然如果能确保一个类型和其他类型之间永远没有隐式转换，实现强类型版本的 Equals 方法还是有很多好处的。但一个类型本身并不能保证这一点，因为在另外一个类型中完全可以定义一个和 MyValType 之间的隐式转换。)另外，因为所有的值类型都直接继承自 System.ValueType，所以我们便可以确保基类型没有需要比较的字段。

大家可能已经注意到了我们在上面接受 Object 作为参数的 Equals 方法中，使用的是 is 操作符来检查 obj 的类型。这里使用 is 而不是 GetType 是因为在值类型实例上调用 GetType 会导致该值类型实例被执行装箱。(译注：因为 obj 已经是引用类型，所以调用 obj.GetType 不会导致装箱。但是调用 this.GetType 会导致装箱。另外请读者注意，对于前面引用类型 MyRefType 的 Equals 实现，则只能使用 GetType 方法，而不能使用 is 操作符。因为 is 操作符在判断一个对象与它的任何一个基类型之间的关系时，都会返回 true。假设 obj 的实际类型为 MyRefTypeDerived，其中 MyRefTypeDerived 派生自 MyRefType 类，那么“obj is MyRefType”也将返回 true，这有可能会导致 MyRefType 中的 Equals 方法执行错误。值类型不会出错是因为它们是密封类型，不存在派生类型。)本书 5.3 节中对此有描述。

6.1.4 Equals 方法与 ==/!= 操作符的实现总结

本节对怎样在自己定义的类型中实现等值性判定操作做一总结。

- **编译器认为的基元类型** 我们选择的编译器对它认为的基元类型提供了 == 和 != 操作符实现。例如 C#编译器知道怎样对 Object、Boolean、Char、Int16、UInt16、Int32、UInt32、Int64、UInt64、Single、Double、Decimal 等类型进行判等比较。另外，这些类型也重写了 Equals 方法，所以我们可以和使用操作符一样来调用它们的 Equals 方法。

- **引用类型**　对于自己定义的引用类型，我们应该重写 Equals 方法，在其内部比较对象的状态。如果我们定义的类型的基类型没有继承 Object.Equals 方法的实现，那么我们应该调用基类型的 Equals 方法。如果愿意，我们还可以重载== 和 != 操作符，让它们调用重写后的 Equals 方法。
- **值类型**　对于值类型，我们应该为其定义一个类型安全的 Equals 来比较对象的状态。然后在实现"非类型安全"的 Equals 时，在其内部调用类型安全的那个版本。我们还应该重载== 和 != 操作符，让它们在内部调用类型安全的 Equals 方法。(译注：由于可能的隐式转换带来的问题，所以这种做法是不被推荐的。上一节的译注对此有详细讨论。)

6.1.5 对象惟一性识别

Equals 方法的目的是比较两个类型实例是否相等，如果两个实例有相同的状态或值，方法将返回 true。然而，有时我们也需要判断两个引用是否指向了同一个对象。为了实现这一点，System.Object 提供了一个名为 ReferenceEquals 的静态方法，它的实现如下：

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

如我们所见，ReferenceEquals 简单地使用 == 操作符来比较两个引用是否相等。这样的做法可行是因为 C#编译器内在的规则。当 C#编译器看到我们使用 == 操作符来比较两个类型为 Object 的引用时，编译器会产生比较两个对象引用是否相同的 IL 代码。(译注：这里 Object 类型是两个变量的编译时类型，而非运行时类型。编译时类型又称声明类型、或静态类型，它是我们在代码中显式声明的类型。运行时类型又称实际类型、或动态类型，它是变量所引用的对象的真实类型，GetType 方法返回的就是变量的运行时类型。ReferenceEquals 方法的参数限定了 objA 和 objB 的编译时类型总为 Object，所以该方法中的 == 操作符编译后的结果总是比较引用是否相同。)

如果我们正在编写 C#代码，我们也可以使用 == 操作符来代替调用 Object.ReferenceEquals 方法，这取决于个人的喜好。然而在使用 == 操作符时，我们必须非常小心。只有 == 操作符两边的变量都为 System.Object 类型时，它才会去比较两个引用是否相同。如果有一个变量不为 Object 类型，并且该变量的类型重载了== 操作符，那么 C#编译器产生的代码将会去调用重载的操作符实现。(译注：这里的说法不够准确。如果一个变量的编译时类型不为 Object，而另一个变量的编译时类型为 Object，那么除非我们自己为该类型和 Object 类型重载了== 操作符——这种做法显然很少见，否则在它们之间直接使用 == 操作符会导致 C#编译器报错。另外，在这种情况下，C#编译器也不会将非 Object 类型的变量隐式转型为 Object 类型。)所以为了代码的清晰和保险起见，我们一般不要使用 == 操作符来判断两个对象引用是否相同，而应该使用 Object 的 ReferenceEquals 静态方法。下面的代码演示了 ReferenceEquals 的使用方法：

```
static void Main() {
    // 构造一个引用类型对象
    RefType r1 = new RefType();

    // 使另一个变量指向该对象
    RefType r2 = r1;

    // r1 和 r2 是否指向同样的对象?
    Console.WriteLine(Object.ReferenceEquals(r1, r2));      // "True"

    // 再构造一个引用类型对象
    r2 = new RefType();

    // r1 和 r2 是否指向同样的对象?
    Console.WriteLine(Object.ReferenceEquals(r1, r2));      // "False"

    // 创建一个值类型实例
    Int32 x = 5;

    // x 和 x 是否指向同样的对象?
    Console.WriteLine(Object.ReferenceEquals(x, x));        // "False"
    // 显示 "False"是因为 x 被两次装箱
    // 到两个不同的对象中去了
}
```

6.2 对象的散列码

FCL 的设计者们认为如果任何对象实例都能被放入一个散列表集合中将是非常有用的。为此，System.Object 提供了一个 GetHashCode 虚方法，这样我们就可以从任何对象上得到一个 Int32 类型的散列码。

如果我们定义了一个类型，并且重写了 Equals 方法，我们也应该重写 GetHashCode 方法。实际上，如果我们定义的类型仅重写了这两个方法中的一个，微软的 C#编译器会产生一个警告信息(译注：实际上笔者的 C#编译器——版本为 7.00.9466——只有在重写了 Equals 方法，而没有重写 GetHashCode 方法的情况下会有警告信息。重写了 GetHashCode 方法，而没有重写 Equals 方法，则没有警告信息)。例如，编译下面的类型会产生这样的警告信息："warning CS0659: 'App' 重写了 Object.Equals(object o) 但没有重写 Object.GetHashCode( )."

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

一个类型必须同时重写 Equals 方法和 GetHashCode 方法是因为 System.Collections.Hashtable 类型

的实现要求任何两个相等的对象都必须有相同的散列码值。所以如果我们重写了 Equals 方法，我们也应该重写 GetHashCode 方法以确保用来判等的算法和用来计算对象散列码的算法一致。

基本上来讲，当我们向一个 Hashtable 对象中添加一个“键/值对”时，其中“键对象”的散列码会首先被获取。该散列码指出了“键/值对”应该被存储在哪个“散列桶(bucket)”中。当 Hashtable 对象需要查找某个“键”时，它会取得指定“键对象”的散列码。然后在该散列码所标识的那个“散列桶”中进一步查找和指定的“键对象”相等的“键对象”。使用这种存储和搜索“键”的算法意味着如果我们改变了 Hashtable 中的一个“键对象”，我们在 Hashtable 中将不能再找到该对象。如果我们要改变一个散列表中的“键对象”，我们应该首先删除原来的“键/值对”，然后改变“键对象”，最后再将新的“键/值对”添加到散列表中。

实现一个 GetHashCode 方法可以相当容易。但是，根据我们的数据类型和数据的分布，如果要得到一个数值范围分布良好的散列算法，还需要一些技巧。下面是一个简单的例子，对于 Point 对象来说可能已经足够了：

```
class Point {
    Int32 x, y;
    public override Int32 GetHashCode() {
        return x ^ y;           // 对 x 和 y 做异或操作
    }
    ...
}
```

当选择计算类型实例的散列码算法时，我们应该尽力遵循以下原则：

- 算法应该使所得的数值有一个良好的随机分布，这样散列表可以获得最佳的性能。
- 算法还可以调用基类型的 GetHashCode 方法，并将其返回值包含在我们自己的算法中。但在一般情况下，我们不应该调用 Object 或者 ValueType 的 GetHashCode 方法，因为这两个类型的 GetHashCode 方法实现都不会获得高性能的散列算法。
- 算法应该使用至少一个实例字段。
- 理想情况下，我们在算法中使用的字段都应该是恒定不变的；也就是说在构造对象时字段被初始化后，它们就不应该再在对象的生存期内有任何改变。
- 算法应该执行的尽可能快。
- 有着相同值的对象应该返回相同的散列码。例如，两个有着相同文本的 String 对象应该返回同样的散列码值。

System.Object 中实现的 GetHashCode 方法对于它的派生类型及其内的字段一无所知。出于这个原因，Object 的 GetHashCode 方法返回的是一个在应用程序域(AppDomain)范围内确保惟一的数值。该数值在对象的整个生存期中保证不会改变。但是，在对象被执行垃圾收集后，这个惟一的数值可以被重新利用作为一个新的对象的散列码。

System.ValueType中实现的GetHashCode方法使用反射来返回定义在类型中第一个实例字段的散列码。这个简单的实现对于某些值类型来说可能已经够用，但是还是建议大家最好提供自己的GetHashCode方法实现。即使我们实现的算法同样返回第一个实例字段的散列码，我们的实现也要比ValueType中实现的快一些。下面是ValueType中实现的GetHashCode方法：

```
class ValueType {
   public override Int32 GetHashCode() {

      // 取得该类型所有的公有和私有实例字段
          //(译注：同样应该是取得类型的所有实例字段)
      FieldInfo[] fields = this.GetType().GetFields(
         BindingFlags.Instance |
         BindingFlags.Public | BindingFlags.NonPublic);

      if (fields.Length > 0) {
        // 返回第一个非空字段的散列码
        for (Int32 i = 0; i < fields.Length; i++) {
           Object obj = fields[i].GetValue(this);
           if (obj != null) return obj.GetHashCode();
        }
      }

      // 如果没有非空字段，为其返回一个与类型相关的惟一的值
      // 注意 GetMethodTablePtrAsInt 是一个内部的未记入文档的方法
      return GetMethodTablePtrAsInt(this);
   }
}
```

如果我们因为某种原因要实现自己的散列表集合，或者编写的任何代码中调用了GetHashCode方法，我们都不应该持久化(persist)散列码值。原因是散列码值可能会改变。例如，一个类型的下一个版本可能会使用不同的算法来计算对象的散列码。

6.3 对象克隆

在程序设计过程中，我们有时会希望得到一个现有对象的拷贝。例如，我们可能希望拷贝一个Int32、一个String、一个ArrayList、一个Delegate，或者其他某个对象。然而，对于某些类型，克隆其对象实例毫无意义。例如，克隆一个System.Threading.Thread对象就没有任何意义，因为创建一个Thread对象，并拷贝其中所有的字段并不会创建一个新的线程。另外对于某些类型，当构造一个实例时，该实例会被加入到一个链表或者其他某种数据结构中。简单的对象克隆可能会破坏该类型的语义。

一个类必须确定是否允许它的实例被克隆。如果希望自己的实例被克隆，该类应该实现ICloneable接口(第15章将深入探讨接口)，它的定义如下：

```
public interface ICloneable {
        Object Clone();
}
```

该接口只定义了一个方法，Clone。我们实现的Clone应该构造类型的一个新的实例，并将新对象的状态初始化为和原来对象相同的状态。IClonealbe接口没有显式表明Clone方法应该实现一个浅拷贝还是一个深拷贝。所以我们必须自己决定哪种拷贝对我们的类型有用，并且应该在文档中清楚地说明这一点。

注意 浅拷贝(shallow copy)是指当对象的字段值被拷贝时，字段引用的对象不会被拷贝。例如，如果一个对象有一个指向字符串的字段，并且我们对该对象做了一个浅拷贝，那么两个对象将引用同一个字符串。而深拷贝(deep copy)是对对象实例中字段引用的对象也进行拷贝的一种方式。所以如果一个对象有一个指向字符串的字段，并且我们对该对象做了一个深拷贝的话，我们将创建一个新的对象和一个新的字符串——新对象将引用新的字符串。需要注意的是执行深拷贝后，原来的对象和新创建的对象不会共享任何东西；改变一个对象对另一个对象没有任何影响。

许多开发人员都将Clone方法实现为浅拷贝。如果希望为自己的类型实现浅拷贝，我们可以像下面这样在Clone方法中调用System.Object的受保护方法MemberwiseClone即可。

```
class MyType : ICloneable {
    public Object Clone() {
        return MemberwiseClone();
    }
}
```

Object.MemberwiseClone方法首先会为新对象分配内存(新对象的类型和this指针引用的对象类型相同)。然后，MemberwiseClone方法会遍历类型(和它的基类型)中所有的实例字段，并将原对象中所有的位拷贝到新对象中。注意它不会为新对象调用构造器，而只是保证它的状态和原对象一致。

我们也可以实现自己的Clone方法，而不必调用Object的MemberwiseClone方法。看下面的示例：

```
class MyType : ICloneable {
    ArrayList set;

    // 供Clone方法调用的私有构造器
    private MyType(ArrayList set) {
        // 引用一个传入集合的深拷贝
        this.set =(ArrayList)set.Clone();
    }

    public Object Clone() {
        // 构造一个新的MyType 对象，构造器参数为
        // 原来对象中使用的ArrayList
        return new MyType(set);
    }
}
```

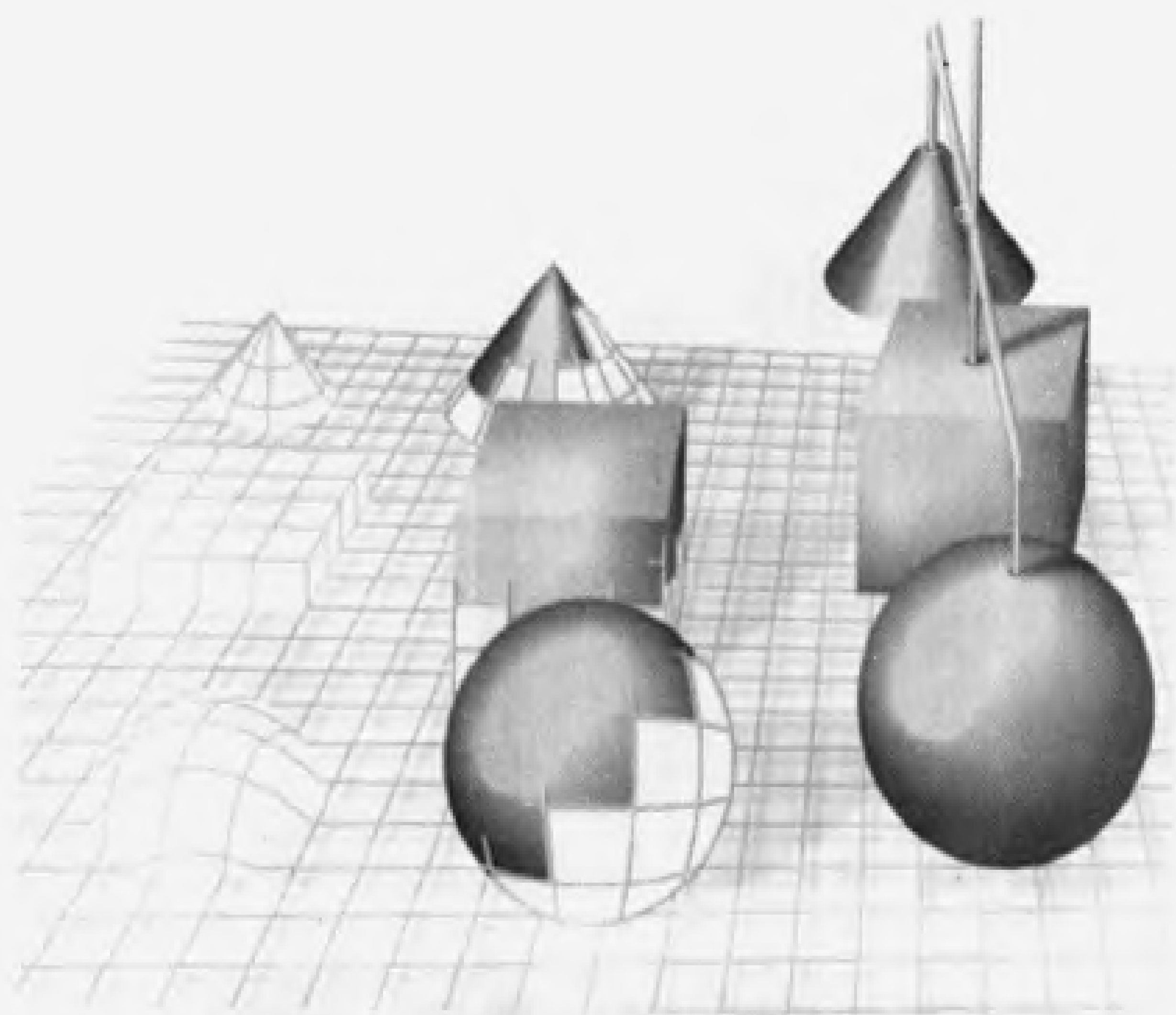
大家可能已经注意到本节的讨论主要针对的是引用类型。这是因为值类型实例本身就支持浅拷贝。毕竟，在进行装箱操作时，系统必须能够拷贝一个值类型实例中的字节。下面的代码演示了值类型的克隆：

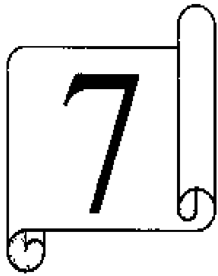
```
static void Main() {
    Int32 x = 5;
    Int32 y = x;        // 将字节从x 拷贝到 y中
    Object o = x;       // 对x执行装箱操作的同时会将字节从x 拷贝到托管堆上
    y = (Int32) o;      // 对o执行拆箱操作，并将字节从托管堆上拷贝到y中
}
```

当然，如果我们定义了一个值类型，并希望自己的值类型支持深拷贝，我们也应该使该值类型像前面一样实现ICloneable接口。(不要调用MemberwiseClone方法，而是分配一个新的对象，实现自己的深拷贝语义。)

第Ⅲ部分

类型设计





类型成员及其访问限定

本书第II部分主要讨论了类型以及每个类型的所有实例都具有的一组通用操作。另外，我们还详细探讨了两种不同的类型——引用类型和值类型。本章和接下来的几章将向大家展示如何使用各种不同的成员来设计一个类型。其中第 8 章到第 11 章会深入探讨各种成员的细节部分。

7.1 类型成员

一个类型可以定义零个或多个以下成员：

- **常数(第 8 章)** 常数是一个表示恒定不变的数值的符号。这些符号主要用来使得代码更具可读性和可维护性。常数总是和类型而非它们的实例相关联。从这个意义上说，它们总是静态的。

- **字段(第 8 章)** 字段表示一个数据的值，它或者是只读的，或者是可读可写(又称读写)的。字段还可分为静态字段和实例字段(非静态的)两种，静态字段被视为类型状态的一部分，实例字段被视为对象状态的一部分。强烈建议大家将字段的访问限定声明为私有方式，这样可以避免类型或者对象的状态为所定义类型外部的代码破坏。

- **实例构造器(第 9 章)** 实例构造器是一种特殊的方法，它用来将一个新对象的实例字段初始化到正常的初始状态。

- **类型构造器(第 9 章)** 类型构造器也是一种特殊的方法，它用来将一个类型的静态字段初始化到正常的初始状态。

- **方法(第 9 章)** 方法是一个函数，用来改变或查询一个类型(就静态方法而言)，或者一个对象(就实例方法而言)的状态。方法一般需要读写类型或对象的字段。

- **重载操作符(第 9 章)** 重载操作符同样也是一种方法，它用操作符的形式定义了怎样对对象进行某种操作。因为并非所有的编程语言都支持操作符重载，所以操作符重载方法也不是通用语言规范(CLS)的一部分。

- **转换操作符(第 9 章)** 转换操作符也是一种方法，它定义了怎样将一个对象从一种类型转换到另一种类型，这种转型可以是隐式的也可以是显式的。和重载操作符类似，一些编程语言也不支持转换操作符，所以它们也不是 CLS 的一部分。

- **属性(第 10 章)** 属性仍是一种方法，它以一种简单的、类似字段的方式实现了设置、或者查询一个类型或对象的状态，与此同时它又可以很好地保护它们的状态不会被破坏。

- **事件(第 11 章)** 事件分为静态事件和实例事件两种。静态事件通过类型发送通知，通知的接收者可以是一个类型，也可以是一个对象。实例事件通过对象发送通知，通知的接收者同样可以是一个类型，也可以是一个对象。事件通常在提供事件的类型或者对象的状态改变时被触发。事件包含两个方法，它们允许类型或对象(通常被称为“侦听者”)登记或者注销其感兴趣的“事件”。另外，事件采用委托字段来维护登记该事件的侦听者集合。

- **类型** 类型内部可以嵌套定义其他类型。这种策略通常用于将一个庞大复杂的类型划分成较小的代码块，从而达到简化实现的目的。

本章不会涉及以上各种成员的细节，我们将主要着眼于各成员中共同拥有的那一部分特性，以此为后面几章做一些铺垫性的工作。

不管使用何种编程语言，编译器总是要先对我们的源代码进行处理，然后为每一种成员产生相关的元数据，并为其中的方法成员产生 IL 代码。元数据的格式和编程语言之间无任何关系，这使得 CLR 成为名副其实的“通用语言运行时”。元数据信息对于所有语言都是通用的，这使得一门编程语言可以无缝地访问另一门语言编写的代码。

这种通用的元数据格式也为 CLR 所使用，CLR 用它们在运行时决定常数、字段、构造器、方法、属性、以及事件的行为。元数据在微软整个.NET 框架开发平台中占据着关键的角色，正是有了它，所有的语言、类型、对象才得以实现无缝集成。

下面的 C#代码定义了一个包括所有可能成员的类型。这段代码应该能够通过编译(虽然会出现警告信息)，但它并不能代表我们通常所要创建的类型，其中定义的大多数方法根本没有做任何有价值的事情。这里仅仅是为了展示编译器是怎样将这些类型和成员翻译成元数据的。我们将在接下来的几章中逐一讨论它们。

```
using System;

class SomeType {                                              1

    // 嵌套类
    class SomeNestedType  { }                                 2


    // 常数、只读字段、静态读写字段
    const Int32 SomeConstant = 1;                             3
    readonly Int32 SomeReadOnlyField = 2;                     4
    static Int32 SomeReadWriteField = 3;                      5

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

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

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

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

    // 实例索引器属性
    public Int32 this[String s]                             14
    {
       get { return 0; }                                    15
       set { }                                              16
    }

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

编译上面定义的类型，然后用ILDasm.exe来查看得到的元数据。我们将会看到如图7.1中所示的输出结果。

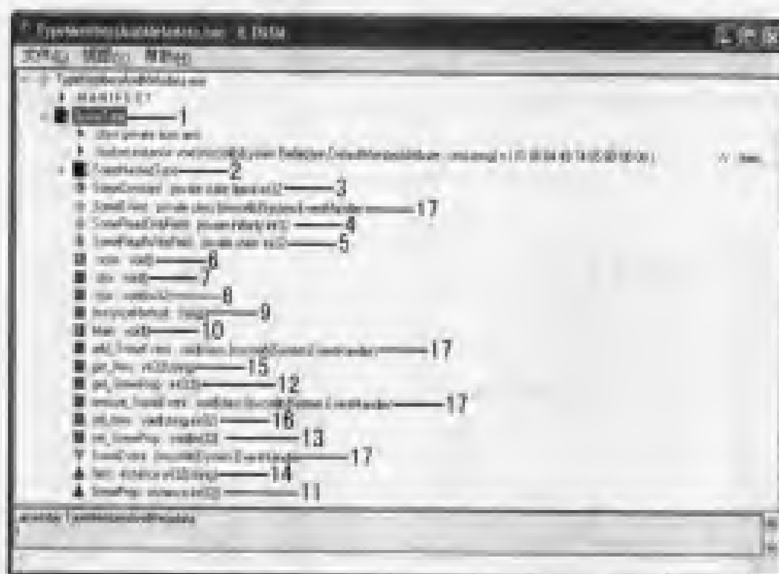


图 7.1 前面的代码编译后产生的元数据在ILDasm.exe中的显示输出

注意编译器为所有定义在源代码中的成员都产生了相关的元数据。实际上，对于其中一些成员(例如事件，17)编译器还为它们产生了额外的成员(一个字段和两个方法)与元数据。就目前而言，大家没有必要完全理解其中的内容。但在接下来几章的学习中，大家应该经常回顾这个示例来查看怎样定义这些成员，以及编译器都为它们产生了什么样的元数据。

7.2 访问限定修饰符和预定义特性

本节来对访问限定修饰符，以及应用于类型、字段和方法(包括属性和事件)的一些预定义特性做一总结。访问限定修饰符指出了哪些类型和成员可以被其他的代码合法地引用。预定义特性则在访问限定修饰符的基础上为我们提供了更多的选择，并且允许我们改变一个成员的语义。

CLR定义了所有可能的访问限定修饰符集合，但是各种编程语言都选择了自己的面向开发人员的语法和术语。例如，CLR用Assembly来表示一个成员可以被同一程序集中的代码所访问。而这在C#和Visual Basic分别用internal和Friend来表示。

表7.1显示了可以应用于类型、字段和方法的访问限定修饰符。其中，Private定义的访问限定最为严格，Public定义的最为宽泛。

表7.1 应用于类型、字段、或方法的访问限定修饰符

CLR术语	C#术语	Visual Basic术语	描 述
Private	private	Private	仅可以被所定义类型(或者其任何嵌套类型)中的方法访问
Family	protected	Protected	仅可以被所定义类型(或者其任何嵌套类型)及其派生类型中的方法访问，与所在程序集无关
Family 与 Assembly	(不支持)	(不支持)	仅可以被所定义类型(或者其任何嵌套类型)，及与其同在一个程序集中的派生类型中的方法访问
Assembly	Internal	Friend	仅可以被所定义程序集中的方法访问
Family 或 Assembly	protected internal	Protected Friend	仅可以被所定义类型、派生类型以及任何定义在同一程序集中方法访问
Public	public	Public	可以被所有程序集中的所有方法访问

当设计一个类型或者成员时，我们只能选择一个访问限定修饰符。比如，我们不能将一个方法同时标识为 Assembly 和 Public。嵌套类型(可以看作成员)可以使用以上六种访问限定修饰符中的任何一种。而非嵌套类型则只能标识为 Public 或者 Assembly，因为其他的访问限定修饰符没有任何意义。如果一个非嵌套类型没有显式标识访问限定修饰符，C#和 Visual Basic 将默认使用 Assembly(internal/Friend)来标识。

除了访问限定修饰符外，类型和成员还可以用一些预定义特性来标识。同样，CLR 也定义了一个预定义特性的集合，也允许各个语言为这些预定义特性选择使用自己的命名方式。

7.2.1 类型预定义特性

表 7.2 为我们展示了应用于类型的预定义特性。

表 7.2 应用于类型的预定义特性

CLR 术语	C# 术语	Visual Basic 术语	描 述
Abstract	abstract	MustInherit	不能被实例化。可以用作其他类型的基类型。如果派生类型不是抽象的，则可以构造它的实例
Sealed	sealed	NotInheritable	不能用作基类型

CLR 允许我们使用 Abstract 或 Sealed 来修饰一个类型，但两者不可以同时使用。这种限制有些令人遗憾，毕竟总有一些类型，我们既不希望创建它们的实例，也不希望它们作为基类型被继承。

比如，创建一个 Console 或者 Math 类型的实例就没有任何意义，因为它们仅包含了一些静态方法。通过继承它们来定义新的类型同样没有意义。这时如果能将它们同时标识为 Abstract(不允许创建实例)和 Sealed(不允许用作基类型)将是比较合适的。

因为 CLR 不支持这种标识，所以如果我们定义的类型仅包含静态成员，那么我们首先应该将此类型定义为 Sealed，同时为它定义一个私有的(Private)无参构造器。定义私有构造器会阻止 C#编译器为类型自动产生公有的(Public)无参构造器。因为类型外的代码不能访问私有构造器，自然也就不能创建它的实例了。

7.2.2　字段预定义特性

表 7.3 为我们展示了应用于字段的预定义特性。

表 7.3　应用于字段的预定义特性

CLR 术语	C# 术语	Visual Basic 术语	描　述
Static	static	Shared	字段是类型状态、而非对象状态的一部分
InitOnly	readonly	ReadOnly	字段仅可以在构造器方法中被赋值

CLR 允许字段有 3 种标记：Static，InitOnly，或者同时使用 Static 和 InitOnly。注意常数(第 8 章予以讨论)总被认为是 Static，也不能用 InitOnly 来标记。

7.2.3　方法预定义特性

表 7.4 为我们展示了应用于方法的预定义特性。

表 7.4　应用于方法的预定义特性

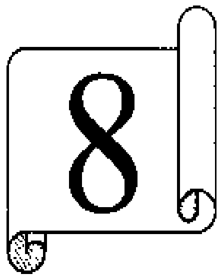
CLR 术语	C# 术语	Visual Basic 术语	描　述
Static	static	Shared	方法和类型、而非类型的实例相关。静态方法不能访问类型中的实例字段或实例方法，因为静态方法对对象的状态一无所知
Instance	(默认)	(默认)	方法和类型的实例而非类型本身关联。方法可以访问实例字段和实例方法，也可以访问静态字段和静态方法
Virtual	virtual	Overridable	当方法被调用时，无论对象是否被转换为其基类型，都只有位于对象继承链最末端的方法实现(most-derived method)会被调用。(译注：换句话说，虚方法是按照其运行时类型，而非编译时类型进行动态绑定调用的。)仅应用于实例(非静态)方法

续表

CLR 术语	C# 术语	Visual Basic 术语	描 述
Newslot	new	Shadows	方法的子类实现不会重写基类型中的实现，而仅仅是将其隐藏起来。仅应用于虚方法
Override	override	Overrides	显式表明方法在重写基类型中的虚方法。仅应用于虚方法
Abstract	abstract	MustOverride	表示派生类型必须提供和该抽象方法签名匹配的实现。(译注：派生类型也可以不提供这样的实现，而让基类中的抽象方法继续“抽象”下去。这时的派生类型必须为抽象类型。)含有抽象方法的类型是一个抽象类型。仅应用于虚方法
Final	sealed	NotOverridable	派生类型不能重写该方法。仅应用于虚方法

我们将在第 9 章详细地讨论这些预定义特性。任何一个多态实例方法(译注：多态实例方法即虚方法)都可以被标记为 Abstract 或者 Final，但两者不能同时使用。将一个虚方法标识为 Final 意味着不再允许任何派生类型重写该方法——这种情况并非经常发生。

当源代码被编译时，各个语言编译器负责检查它们是否正确地引用了类型和成员。如果源代码中非法引用了一些类型或成员，编译器有责任产生相应的错误信息。除此之外，在将 IL 代码编译为本地 CPU 指令时，JIT 编译器也要确保对字段和方法的引用合法。比如，当验证器(verifier)检测到有代码试图非法访问私有字段或者私有方法时，JIT 编译器将分别抛出 FieldAccessException 异常和 MethodAccessException 异常。这样，即使当某个语言编译器因忽略了这些工作而产生出无效的程序集时，IL 代码验证过程也能确保访问限定修饰符和预定义特性的限定作用在运行时得到正确的贯彻。



常数与字段

本章向大家介绍怎样向一个类型中添加数据成员。具体而言，我们将讨论常数和字段。

8.1 常 数

常数是一个表示恒定不变的值的符号。定义一个常数符号时，我们必须能够在编译时确定它的值。通过编译后，编译器将常数的值保存在其所定义模块的元数据内。这意味着常数的类型只能是那些编译器认为的基元类型。(译注：因为只有基元类型的数据成员才能利用文本常数，即 literal，在编译时直接进行初始化。而非基元类型的数据成员只能在运行时调用构造器来完成初始化。)另一个需要注意的是常数总被认为是类型(而非实例)的一部分，这在常数值恒定不变的含义下很容易理解。

> **注意** 在 C#中，下面的类型称为基元类型，可以被用来定义常数：Boolean、Char、Byte、SByte、Decimal、Int16、UInt16、Int32、UInt32、Int64、UInt64、Single、Double，以及 String。(译注：枚举类型由于本身以基元类型形式存储，故也可以被用来定义常数，虽然它并不是基元类型。)

当使用常数符号时，编译器首先从定义常数的模块的元数据中查找出该符号，直接取出常数的值，然后将之嵌入到编译后产生的 IL 代码中。因为常数的值是直接嵌入到代码中的，所以常数在运行时不再需要任何的内存分配。另外，我们也不能获取常数的地址，或者以引用的方式来传递一个常数。这些约束还意味着常数没有一个良好的跨模块版本特性。也就是说只有当确信一个符号的值永远不会改变时，我们才应该使用常数来定义它们(比如将 MaxInt16 定义为 32767)。

来看一个例子，首先将下面的代码编译成 DLL 程序集：

```
using System;
public class Component {
    // 注意：C#不允许为常数指定 static 关键字
    // 因为常数隐含为 static
    public const Int32 MaxEntriesInList = 50;
}
```

引用上面得到的程序集，将下面的代码编译成一个应用程序：

```
using System;

class App {
    static void Main() {
        Console.WriteLine("Max entries supported in list: "
          + Component.MaxEntriesInList);
    }
}
```

注意上面的程序代码引用了 MaxEntriesInList 常数。当该应用程序被编译时，编译器会发现 MaxEntriesInList 是一个值为 50 的常数符号，它将直接把整数值 50 嵌入到应用程序的 IL 代码中。实际上，在该应用程序代码完成编译之后，先前引用的 DLL 程序集在程序运行时甚至不会被加载，而且完全可以将之从磁盘中删除。

上面的示例清晰地展示了常数所隐含的版本问题。如果我们把前面 DLL 程序集中 MaxEntriesInList 常数值改为 1000 后并重新编译该 DLL 程序集，后面的应用程序代码将不会受到任何影响。要获得新的常数值，我们必须重新编译后面的应用程序。如果要求一个模块中的数值能够在运行时(而不是编译时)被另一个模块获取，那么就不应该使用常数。相反，我们应该使用只读字段，这将在本章后面予以探讨。

8.2 字　段

字段又称数据成员，它保存着一个值类型的实例、或者一个指向引用类型的引用。CLR 支持类型(静态)和实例(非静态)两种字段。对于类型字段，系统在该类型被加载进入一个应用程序域（参见第 20 章)时为其分配动态内存，这通常发生在引用该类型的方法第一次被 JIT 编译时。对于实例字段，系统在该类型的实例被构建时为其分配动态内存。

因为字段是以动态内存的形式存储的，因此只能在运行时刻获取它们的值。字段也没有常数的版本问题。另外，字段可以是任何类型，没有像常数那样的限制。

CLR 支持只读和读写两种字段。大多数的字段都是读写字段，这意味着代码在执行过程中字段可以被多次赋值。但是只读字段只能在构造器内被赋值(构造器在对象初次创建时被执行，且只执行一次)。(译注：值得注意的是在构造器内部只读字段却可以被多次赋值。另外作者这里指的是实例只读字段和实例构造器。对于静态只读字段，则只能在静态构造器内赋值，静态构造器在该类型初次被引用时执行。关于构造器更详细的介绍见第 9 章。)编译器和 IL 代码验证器可以确保除了构造器外没有其他方法会改写只读字段。

下面我们用静态只读字段来解决“常数”一节示例中的版本问题。下面是新版 DLL 程序集的代码：

```
using System;

public class Component {
    // 类型字段需要 static 关键字
    public static readonly Int32 MaxEntriesInList = 50;
}
```

这是惟一需要改变的地方，App 类的代码无需改变，但需要重新编译。当我们运行应用程序中的 Main 方法时，CLR 将加载 DLL 程序集(现在在运行时就需要该程序集)。然后从其动态内存中取出 MaxEntriesInList 字段。显然，这时的值将为 50。

假设我们将 MaxEntriesInList 字段的值从 50 改变为 1000，并重新编译该 DLL 程序集。当我们再次执行应用程序时，它将自动获取修改后的值 1000。值得指出的是，我们的应用程序代码并没有被重新编译，它仅仅是再次运行了一遍(虽然它的性能受到一点负面的影响)。注意，这种情形是在如下的假设前提下才成立的：新版的 DLL 程序集并非强命名程序集，或者该应用程序的版本策略要求 CLR 加载新版的程序集。

前面的例子演示了怎样定义静态只读字段。同样我们也可以定义静态读写字段，以及实例只读字段和实例读写字段。看下面的代码：

```
public class SomeType {

    // 一个静态只读字段；它的值将在运行时类
    // 被初始化时计算并存储在内存中。
    public static readonly Random random = new Random();

    // 一个静态读写字段
    static Int32 numberOfWrites = 0;

    // 一个实例只读字段
    public readonly String pathName ="Untitled";

    // 一个实例读写字段
    public System.IO.FileStream fs;

    public SomeType(String pathName) {
        // 该行修改只读字段 pathName
        // 因为是在构造器中，所以可行
        this.pathName = pathName;
    }

    public String DoSomething() {
        // 该行首先读取静态读写字段，然后又为其赋值
        numberOfWrites = numberOfWrites + 1;

        // 该行读取实例只读字段
        return pathName;
    }
}
```

在上面的代码中，很多字段都是以内联的方式进行初始化的(译注：这里内联的意思即为在声明字段的同时进行初始化赋值)。C#允许我们采用这种方便的内联初始化语法来初始化一个类的常数、读写字段和只读字段。在第 9 章我们将会看到 C#对字段的内联初始化仅仅是一种简化的表达方式，实际上它们的初始化是在构造器内完成的。

方　法

本章主要讨论一个类型中可以定义的各种不同的方法，以及与这些方法相关的一些问题。具体而言，本章将向大家展示怎样定义构造器方法(包括实例构造器和类型构造器)、操作符重载方法、转换操作符方法(包括隐式转型和显式转型)。另外，本章还会谈到怎样以引用的方式向方法传递参数，以及怎样定义接受可变数目参数的方法。最后，本章还会向大家详细解释虚方法的版本机制(这种机制可以防止在某类的编程接口发生变化时，应用程序出现潜在的不稳定性)。

9.1　实例构造器

实例构造器是一种特殊的方法，它们负责将类型的实例初始化到一个良好的状态。对于可验证的代码，通用语言运行时(CLR)要求每个类(引用类型)至少定义一个实例构造器。(如果希望阻止类外的代码创建该类的实例，可以将构造器的访问限制设为私有方式。)在创建一个引用类型的实例时，系统将执行以下三个步骤：首先为该实例分配内存，然后初始化对象的附加成员(即方法表指针和一个SyncBlockIndex)，最后调用类型的实例构造器设置对象的初始状态。

当构造一个引用类型实例时，在调用其实例构造器之前，系统为该对象分配的内存总是首先被设为0值(译注：二进制意义上的0值，即将对象字段所占的内存的所有位都置零)。因此，所有构造器没有显式赋值的字段都保证有一个0或null的值。

默认情况下，对于引用类型，如果我们没有显式为其定义实例构造器，许多编译器(包括C#)都会为我们定义一个公有的无参构造器(通常称作默认构造器)。例如下面的类型就会有一个公有的无参构造器，它允许任何可以访问该类型的代码来构造它的实例。

```
class SomeType {
    // C#编译器会为我们自动定义一个默认的公有无参构造器
}
```

上面的类型定义等同于下面的类型定义：

```
class SomeType {
   public SomeType() { }
}
```

一个类型可以定义多个实例构造器。每个构造器都必须有一个不同的签名。多个构造器可以有不同的访问限制。对于可验证的代码，一个类的实例构造器在访问其基类的继承字段之前，必须调用其基类的实例构造器。许多编译器(包括 C#)都会自动产生对基类默认构造器(如果有的话)的调用代码，所以一般情况下，我们不必担心这个问题。这样沿着继承层次，我们定义的每一个构造器最后总会调用到 System.Object 的公有无参构造器。该构造器不执行任何代码，只是简单地返回。

在少数的几种情况下，类型实例的创建不需要调用实例构造器。例如，调用 Object 的 MemberwiseClone 方法将执行以下几步：为对象分配内存，初始化对象的附加成员，将源对象的字节拷贝到新创建的对象中。另外，在反序列化一个对象时，通常也不会调用构造器。

C#为我们提供了一种简单的语法来初始化字段：

```
class SomeType {
   Int32 x = 5;
}
```

当构造 SomeType 对象时，它的 x 字段将被初始化为 5。这是怎样完成的呢？如果我们查看 SomeType 构造器方法(也称.ctor)的 IL 代码，我们会看到如图 9.1 所示的代码。

```
SomeType::.ctor : void()
.method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       14 (0xe)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  ldc.i4.5
  IL_0002:  stfld      int32 SomeType::x
  IL_0007:  ldarg.0
  IL_0008:  call       instance void [mscorlib]System.Object::.ctor()
  IL_000d:  ret
} // end of method SomeType::.ctor
```

图 9.1 SomeType 构造器方法的 IL 代码

在图 9.1 中，我们可以看到 SomeType 构造器首先将 5 赋值给了 x，紧接着它又调用了基类的构造器。换句话说，C#提供的允许我们以内联方式初始化实例字段的简化语法，实际上都被转换成了构造器中的代码。这意味着我们需要警惕代码的膨胀效应。试想下面的这个类：

```
class SomeType {
    Int32 x = 5;
    String s = "Hi there";
    Double d = 3.14159;
    Byte b;

    // 下面是一些构造器
    public SomeType() { ... }
    public SomeType(Int32 x) { ... }
    public SomeType(String s) { ...; d = 10; }
}
```

当编译器为以上三个构造器方法产生代码时，每一个方法的开始处都将包括初始化 x、s、和 d 的代码。在这些初始化代码之后，编译器才会为各个构造器添加出现在其中的代码。例如，对于接受 String 参数的那个构造器，编译器产生的代码首先是初始化 x、s 和 d，然后才是将 10 赋值给 d。注意虽然没有代码对 b 进行显式的初始化，但 CLR 会保证将其初始化为 0。

因为上面的类中有三个构造器，所以编译器共产生三次初始化 x、s 和 d 的代码——每一个构造器一次。如果我们有一些需要初始化的实例字段和许多重载的构造器方法，我们应该考虑在定义字段的时候避免同时对它们进行初始化，相反我们应该将这些公共的初始化语句放在一个初始化构造器中，然后使其他的构造器显式地调用这个初始化构造器。这将有助于减少生成代码的尺寸。看下面的代码：

```
class SomeType {
    // 没有显式初始化字段
    Int32 x;
    String s;
    Double d;
    Byte b;

    // 所有其他的构造器都必须调用下面的构造器
    // 该构造器中包含了初始化字段的代码
    public SomeType() {
        x = 5;
        s = "Hi There!";
        d = 3.14159;
    }

    // 下面的构造器首先调用默认的构造器
    public SomeType(Int32 x) : this() {
        this.x = x;
    }
```

```
    // 下面的构造器首先调用默认的构造器
    public SomeType(String s) : this() {
        this.s = s;
    }
}
```

值类型构造器和引用类型构造器的工作方式有很大的差别。首先，CLR 没有强制要求值类型中必须定义构造器方法。实际上，很多编译器(包括 C#)都不会为值类型产生默认的无参构造器。存在这种差别的原因是值类型可以被隐式地创建。看下面的代码：

```
struct Point {
    public Int32 x, y;
}
class Rectangle {
    public Point topLeft, bottomRight;
}
```

要构造一个 Rectangle 实例，我们必须使用 new 操作符，并指定一个构造器。对于上面的例子，C#编译器自动产生的构造器将被调用。当系统为 Rectangle 实例分配完内存时，其中将包括两个 Point 值类型的实例。基于性能的考虑，CLR 不会尝试为每个包含在引用类型中的值类型字段调用构造器。但是，如前所述，CLR 会保证值类型中所有的字段都被初始化为 0 或者 null。

虽然 C#编译器不会为值类型自动产生默认的无参构造器，但是 CLR 却允许我们为值类型定义构造器。只有当我们显式地调用这些构造器的时候，它们才会执行。看下面的代码：

```
struct Point {
    public Int32 x, y;
    public Point(Int32 x, Int32 y) {
        this.x = x;
        this.y = y;
    }
}

class Rectangle {
    public Point topLeft, bottomRight;

    public Rectangle() {
        // 对于C#来讲，在 1 一个值类型上调用 new 仅仅只是调用构造器
        // 来初始化已经分配好内存的值类型。
        // (译注：注意与在引用类型上调用 new 时执行的 3 个步骤对比)
        topLeft = new Point(1, 2);
        bottomRight = new Point(100, 200);
    }
}
```

一个值类型的实例构造器只有当被显式调用时才会执行。所以如果 Rectangle 类的构造器中没有用 new 操作符来调用 Point 的构造器初始化它的 topLeft 和 bottomRight 字段的话，两个 Point 中的字段 x 和 y 都将保持为 0。

在前面的 Point 值类型中，我们没有为其定义默认的无参构造器。让我们来对其做如下改写：

```
struct Point {
    public Int32 x, y;

    public Point() {
        x = y = 5;
    }
}

class Rectangle {
    public Point topLeft, bottomRight;

    public Rectangle() {
    }
}
```

现在当我们构造一个新的 Rectangle 实例时，大家猜猜两个 Point 字段 topLeft 和 bottomRight 中的 x 和 y 将被初始化为什么：0 还是 5？(提示：这是一个圈套。)

许多开发人员(尤其是那些有着 C++背景的)都期望 C#编译器会在 Rectangle 类中为两个字段产生自动调用 Point 的默认无参构造器的代码来。然而，为了提高应用程序的运行时性能，C#编译器并不会自动产生这样的代码。实际上，即使值类型提供了无参的默认构造器，很多编译器也不会自动产生代码来调用它。要使一个值类型的无参构造器执行，开发人员必须显式地调用它。

基于上面的讨论，大家可能会认为 Rectangle 类中两个 Point 字段的 x 和 y 会被初始化为 0，毕竟上面的代码中没有显式调用 Point 的构造器。

然而，上面说过这是一个圈套，因为 C#根本不允许我们为一个值类型定义无参构造器！所以前面的代码实际上根本就通不过编译。当我们试图编译它时，C#编译器会产生以下错误："error CS0568: 结构不能包含显式的无参数构造器"。

C#不允许值类型定义无参构造器是为了消除开发人员对于何时调用它们产生的混淆。如果不能定义无参构造器，编译器自然也不会产生自动调用它的代码。没有无参构造器，值类型的字段将总是首先被初始化为 0 或 null。

> **注意** 严格地讲，只有在值类型字段被嵌入到一个引用类型中时，CLR才保证值类型字段被初始化为0或null。基于堆栈的值类型字段一般不能保证得到0或null。但是，出于代码的可验证性，所有基于堆栈的值类型都必须在读取之前得到赋值。如果一个字段能在得到赋值之前就被读取，那么就可能出现安全漏洞。C#和其他产生可验证代码的编译器可以确保所有基于堆栈的值类型都或者被赋为0值，或者至少在读取之前得到赋值，这样的实现可以保证它们产生的代码在运行时不会抛出验证性异常。这意味着大多数情况下我们可以完全忽略这段注解，并假定值类型的字段已经被初始化为0或null。

注意虽然C#不允许值类型有无参构造器，但CLR却允许这样。所以如果我们需要这样的行为，我们可以使用另外一种编程语言(例如IL汇编语言)来为值类型定义无参构造器。

因为C#不允许值类型有无参构造器，所以编译下面的类型将产生以下错误："error CS0573:SomeValType.x：结构中不能有实例字段初始值设定项。"

```
struct SomeValType {
    Int32 x = 5;
}
```

另外，由于可验证代码要求所有的值类型字段在被读之前首先进行初始化，所以我们为值类型定义的任何构造器都必须要初始化其所有的字段。例如，下面的值类型定义了一个构造器，但是却没有初始化其所有的字段：

```
struct SomeValType {
    Int32 x, y;

    // C# 允许值类型有带参数的构造器
    public SomeValType(Int32 x) {
        this.x = x;
        // 注意 y 还没有被初始化
    }
}
```

当编译上面的类型时，C#编译器将产生以下错误："error CS0171: 在控制离开构造器之前，字段SomeValType.y必须完全赋值。"要修复这个问题，我们可以在构造器中为y赋一个值(通常为0)。

重要 在C#中，我们的源代码通过定义名称和类型名相匹配的方法来定义构造器。当C#编译器编译源代码时，它会检测到这样的构造器方法，并在模块的方法定义元数据表中添加一个相应的条目。在该表中，构造器方法的名称总为.ctor。但在Visual Basic源代码中，开发人员通过创建名为New的方法来定义构造器。这意味着在Visual Basic中，我们可以定义一个和类型本身名称相同的方法(非构造器方法)。然而，我们应该避免这种行为，因为这样的方法不能在其他一些语言中直接被调用(译注：事实上C#即使不通过反射也可以调用这样的函数)。同样的原因，我们在C#中也应该避免定义名为New的方法，因为它不能直接被Visual Basic开发人员所调用。最后，我们可以利用反射机制来调用这些特殊的方法，反射将在本书第20章予以详述。

9.2 类型构造器

除了实例构造器外，CLR还支持**类型构造器**(又称**静态构造器**，**类构造器**，或**类型初始化器**)。类型构造器适用于接口(虽然C#不支持)，引用类型(译注：接口当然也是引用类型，作者这里单独来提接口仅仅是强调的意思)，和值类型。就像实例构造器用来设置一个类型实例的初始状态一样，类型构造器用于设置一个类型的初始状态。默认情况下，一个类型中没有定义类型构造器。如果要定义，也只能定义一个。并且，类型构造器不能有任何参数。下面演示了在C#中怎样为引用类型和值类型定义类型构造器：

```
class SomeRefType {
   static SomeRefType() {
      // 当 SomeRefType 第一次被访问时执行
   }
}
struct SomeValType {
   // C#允许值类型定义无参类型构造器
   static SomeValType() {
      // 当 SomeValType 第一次被访问时执行
   }
}
```

大家会注意到我们定义类型构造器的方式很类似于定义无参的实例构造器，二者惟一不同的是我们必须将类型构造器标记为static。另外，类型构造器的访问限制应该总为私有方式，这由C#编译器自动来完成。实际上，如果我们在代码中显式地将一个类型构造器标记为私有方式(或者其他访问修饰符)的话，C#编译器会报告以下错误："error CS0515: SomeValType.SomeValType()：静态构造器中不允许出现访问修饰符"。

类型构造器的访问限制应该被设为私有方式以防止任何开发人员编写的代码调用它。类型构造器的调用总是由CLR负责。另外，CLR选择调用类型构造器的时机也比较自由。CLR会选择下列时间之一来调用类型构造器。

- 在类型的第一个实例被创建之前，或者在类型的非继承字段或成员第一次被访问之前——注意这个"之前"有一个精确邻接(just before)的意思。因为CLR会在确定的时刻调用类型构造器，故这被称作精确语义(precise semantics)。
- 在非继承静态字段被第一次访问之前的某个时刻。因为CLR只保证静态构造器总是在静态字段被访问之前的某个时刻运行(可能运行的非常早)，故这被称作字段初始化前语义(before-field-init semantics)。

默认情况下，编译器选择对我们定义的类型最有意义的语义，并通过设置beforefieldinit元数据标记来告诉CLR这种选择。类型构造器一旦被执行，它在整个应用程序域(AppDomain)的生命周期内都不会再次被调用。因为是CLR在负责调用类型构造器，所以我们应该避免编写需要以特殊顺序调用类型构造器的代码。

CLR只保证类型构造器开始执行——它并不保证类型构造器完成执行。这对于避免两个类型构造器互相引用时出现的死锁情况是有必要的。

最后，如果一个类型构造器抛出一个未处理的异常，CLR将认为该类型不可用。试图访问其中的任何字段或方法都将抛出System.TypeInitializationException异常。

类型构造器中的代码只能访问类型的静态字段，并且通常它的目的就是初始化这些静态字段。与实例字段一样，C#也提供了一种简单的初始化静态字段的语法：

```
class SomeType {
   static Int32 x = 5;
}
```

当这段代码被编译时，编译器会为 SomeType 自动产生一个类型构造器。其产生的类型构造器和下面的代码编译后的结果是等同的：

```
class SomeType {
    static Int32 x;
    static SomeType() { x = 5; }
}
```

使用 ILDasm.exe 我们可以很容易验证编译器为类型构造器产生的 IL 代码，如图 9.2 所示。类型构造器方法在方法定义元数据表中被称为.cctor(即 class contructor)。

```
SomeType::.cctor : void()
.method private hidebysig specialname rtspecialname static
        void  .cctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  1
  IL_0000:  ldc.i4.5
  IL_0001:  stsfld     int32 SomeType::x
  IL_0006:  ret
} // end of method SomeType::.cctor
```

图 9.2　SomeType 类型构造器方法的 IL 代码

在图 9.2 中，我们可以看到.cctor 方法是一个私有静态方法。另外，注意方法中代码的含义为将 5 赋值给静态字段 x。

还有一点需要注意的是，类型构造器不应该调用其基类型的类型构造器。不需要这样做是因为基类型中的静态字段并没有被派生类型所继承。(译注：这可能会与读者在程序代码中的“认识”相反，因为很多代码或者在派生类型内部引用了基类型的静态字段，或者通过派生类型引用了基类型的静态字段，这不是继承又是什么？事实上这确实不是继承，而是编译时静态绑定。另外，其他的静态成员，如静态方法、静态属性等，也不会被派生类型所继承，同样是编译时静态绑定。)

注意　一些语言，如 Java，期望访问一个类型时会导致它的类型构造器及其所有基类型的类型构造器都被调用。另外，实现了接口的类型也必须调用接口的类型构造器。CLR 本身没有提供这些语义，但是，CLR 通过 System.Runtime.CompilerServices.RuntimeHelpers 类型的 RunClassConstructor 方法为编译器和开发人员提供了这种语义。任何需要该语义的编译器都可以在一个类型的类型构造器中通过它调用所有基类型和实现接口中的类型构造器方法。当试图调用一个类型构造器时，CLR 会判断该类型构造器先前是否执行过，如果执行过，就不会再次被调用。

最后，假设我们有以下代码：

```
class SomeType {
    static Int32 x = 5;
    static SomeType() {
        x = 10;
    }
}
```

对于这段代码，C#编译器会为其产生一个类型构造器方法。该构造器首先将 x 初始化为 5，然后又将其改写为 10。换句话说，当编译器为类型构造器产生 IL 代码时，它首先会产生初始化静态字段所需的代码，然后才是类型构造器方法中显式包含的代码转换而得的 IL 指令。这里生成代码的顺序与实例构造器中的处理方式完全相同。

重要　偶尔也有一些开发人员来问我是否可以在一个类型被卸载的时候执行某些代码。要回答这个问题首先要清楚只有当应用程序域(AppDomain)关闭时类型才会被卸载。当应用程序域关闭时，标识类型的对象将变为不可达对象，垃圾收集器才会回收该对象的内存。这种行为使得许多开发人员认为他们能够为类型增加一个静态的 Finalize 方法，使其在类型被卸载的时候自动得到调用。遗憾的是，CLR 不支持静态的 Finalize 方法。但是，问题并非完全不可解决。如果希望应用程序域关闭时能执行某些代码，我们可以在 System.AppDomain 类型的 DomainUnload 事件上登记一个回调方法。

9.3　操作符重载方法

一些编程语言允许一个类型定义操作符来操作类型的实例。例如，许多类型(如 System.String)都重载了判等(==)和判异(!=)操作符。CLR 对操作符重载一无所知，因为它甚至都不认识操作符是什么。相反，是我们选择的编程语言定义了每个操作符的含义，以及当遇到它们时产生什么样的代码。

例如在 C#中，应用于基元数值类型上的＋符号会使编译器产生将两个数相加的代码。而当＋符号应用于字符串时，C#编译器将产生连接字符串的代码。对于判异操作，C#使用!=符号，而 Visual Basic

用<>符号。最后，^ 符号在C#中意味着异或操作(XOR)，而在Visual Basic中则意味着求幂操作。

虽然CLR对操作符一无所知，但它却规范了编程语言应该怎样提供操作符重载，以使它们可以很容易地被不同的编程语言编写的代码所使用。每个编程语言自己决定是否支持操作符重载，以及如果提供，表达和使用它们的语法是怎样的。对于CLR来讲，重载操作符仅仅是一些方法而已。

我们的编程语言选择了是否支持操作符重载以及它们的语法。当编译源代码时，编译器会产生一个方法来表示操作符的行为。例如，假设我们定义了一个类似于如下的类(在C#中)：

```
class Complex {
   public static Complex operator+(Complex c1, Complex c2) { ... }
}
```

编译器会产生一个名为op_Addition的方法定义；该方法定义条目上有一个specialname标记，表示这是一个"特殊"的方法。当编译器(包括C#编译器)看到源代码中的＋操作符时，它们会去看其中的操作数类型中有哪一个定义了参数类型和操作数类型兼容、名为op_Addition的specialname方法。如果存在这样的方法，编译器将产生调用该方法的代码。如果不存在这样的方法，就会出现编译错误。

表9.1显示了标准C#操作符的集合，以及对应的编译器产生和使用的推荐方法名。下一节会对表中的第3列做一解释。

表9.1　C#操作符和对应的与CLS兼容的方法名

C#操作符符号	特殊的方法名	推荐的与CLS兼容的方法名
+	op_UnaryPlus	Plus
-	op_UnaryNegation	Negate
~	op_OnesComplement	OnesComplement
++	op_Increment	Increment
--	op_Decrement	Decrement
(无)	op_True	IsTrue { get;}
(无)	op_False	IsFalse { get;}
+	op_Addition	Add
+=	op_AdditionAssignment	Add
-	op_Subtraction	Subtract

续表

C#操作符符号	特殊的方法名	推荐的与 CLS 兼容的方法名
-=	op_SubtractionAssignment	Subtract
*	op_Multiply	Multiply
*=	op_MultiplicationAssignment	Multiply
/	op_Division	Divide
/=	op_DivisionAssignment	Divide
%	op_Modulus	Mod
%=	op_ModulusAssignment	Mod
^	op_ExclusiveOr	Xor
^=	op_ExclusiveOrAssignment	Xor
&	op_BitwiseAnd	BitwiseAnd
&=	op_BitwiseAndAssignment	BitwiseAnd
|	op_BitwiseOr	BitwiseOr
|=	op_BitwiseOrAssignment	BitwiseOr
&&	op_LogicalAnd	And
||	op_LogicalOr	Or
!	op_LogicalNot	Not
<<	op_LeftShift	LeftShift
<<=	op_LeftShiftAssignment	LeftShift
>>	op_RightShift	RightShift
>>=	op_RightShiftAssignment	RightShift
(无)	op_UnsignedRightShiftAssignment	RightShift
==	op_Equality	Equals
!=	op_Inequality	Compare
<	op_LessThan	Compare
>	op_GreaterThan	Compare
<=	op_LessThanOrEqual	Compare
>=	op_GreaterThanOrEqual	Compare
=	op_Assign	Assign

重要 如果我们查看一些核心的.NET框架类库(FCL)类型(Int32，Int64，UInt32等)，我们会看到它们没有定义任何的操作符重载方法。其原因是CLR提供了直接操作这些类型实例的IL指令。如果类型为此提供了方法，并且编译器也产生的是调用这些方法的指令，那么在这些方法调用时就会有一些运行时的性能损失。毕竟，这些方法最后还是要执行一些IL指令来完成期望的操作。这也就是核心的FCL类型没有提供任何操作符重载方法的原因。对我们来说这意味着：如果我们使用的编程语言不支持某个 FCL 核心类型，那么我们将不能在其实例上进行任何操作。Visual Basic就是一个例子，因为它不支持无符号数值类型。

9.3.1 操作符与语言互操作性

操作符重载是一个很有用的工具，它允许开发人员通过简洁的语法来表达自己的思想。然而，我们知道并不是所有的编程语言(包括 Visual Basic 和 Java)都支持操作符重载。所以当一个 Visual Basic 开发人员在一个非基元类型(Visual Basic 认为的)上应用+操作符时，编译器将产生一个错误，并停止编译代码。这就产生出一个问题：一个使用不支持操作符重载的语言的开发人员怎样调用另一个支持操作符重载的语言定义的类型呢？

Visual Basic 没有提供特殊的语法来让一个类型定义+操作符。Visual Basic 也不知道怎样将使用+操作符的代码翻译成调用 op_Addition 方法的代码。然而，Visual Basic(像所有其他的语言一样)支持调用一个类型的方法。所以在 Visual Basic 中，我们可以调用由 C#编译器生成的类型中的 op_Addition 方法。

根据上面的信息，大家可能会认为可以在 Visual Basic 中定义一个 op_Addition 方法，在 C#中就可以用+操作符来调用了。然而，这是不正确的。当 C#编译器检测到+操作符时，它会查找带有 specialname 元数据标记的 op_Addition 方法，编译器据此才能确定 op_Addition 方法是一个操作符重载方法。因为由 Visual Basic 产生的 op_Addition 方法没有 specialname 元数据标记，所以 C#编译器会产生编译错误。当然，任何语言中的代码都可以显式调用一个名为 op_Addition 的方法，但是编译器不会将+操作符翻译成对该方法的调用。

下面的代码是对上述讨论的一个总结。其中VBType是一个提供了op_Addition方法的Visual Basic类型(定义在一个库中)。当然，这里的代码并非是正确的实现，仅仅是为了通过编译来演示前面讨论的内容。

```
Imports System

Public Class VBType

   ' 定义对两个VBType对象执行相加操作的op_Addition 方法。
   ' 这并不是一个真正的+操作符的重载方法，因为Visual Basic
   ' 编译器不会在该方法上添加specialname 元数据标记。
   Public Shared Function op_Addition(a as VBType, b as VBType) As VBType
      Return Nothing
   End Function
End Class
```

下面是对两个VBType实例执行相加操作的C#应用程序：

```
using System;

public class CSharpApp {
   public static void Main() {

      // 构造一个VBType实例
      VBType vb = new VBType();

      // 如果下面一行不注释掉，将会出现编译
      // 错误，因为VBType中的op_Addition
      // 方法没有specialname 元数据标记
      // vb = vb + vb;

      // 下面一行会通过编译并运行；
      // 只是代码看起来不够漂亮
      vb = VBType.op_Addition(vb, vb);
   }
}
```

从上面的代码中可以看到，虽然我们在VBType中定义了op_Addition这样的方法，但在C#中我们却不能使用+操作符来对两个VBType对象执行相加操作。当然，我们可以通过显式调用VBType的op_Addition方法来执行这样的操作。

现在让我们反过来，创建一个使用C#类型的Visual Basic应用程序。下面是一个提供了+操作符重载的C#类型(定义在一个库中)：

```
using System;

public class CSharpType {

    // 重载+操作符
    public static CSharpType operator+(CSharpType a, CSharpType b) {
        return null;
    }
}
```

下面是对两个CSharpType实例执行相加操作的Visual Basic应用程序：

```
Imports System

Public Class VBApp
    Public Shared Sub Main()

        ' 构造一个CSharpType实例
        Dim cs as new CSharpType()

        ' 如果下面一行不注释掉，将会出现编译错误，因
        ' 为Visual Basic 不知道怎样将+操作符翻译为
        ' 对CSharpType 的op_Addition方法的调用
        ' cs = cs + cs

        ' 下面一行会通过编译并运行：
        ' 只是代码看起来不够漂亮
        cs = CSharpType.op_Addition(cs, cs)
    End Sub
End Class
```

这里，Visual Basic代码不能使用+操作符来对两个CSharpType实例执行相加操作是因为Visual Basic不知道怎样将+操作符翻译为op_Addition方法。然而Visual Basic可以通过显式调用CSharpType的op_Addition方法来将两个对象加在一起(即使该方法有一个specialname元数据标记)。

关于微软的操作符方法的命名规则

相信大家会对所有这些调用操作符重载方法的规则感到非常的迷惑和不解。如果支持操作符重载的编译器不产生 specialname 元数据标记，整个规则就要简单许多，并且我们处理其中含有操作符重载方法的类型也要方便许多。因为这样以来只要我们在一个类型中定义了 op_方法，支持操作符重载的语言就能够使用操作符来方便地调用这样的方法，而其他语言也能够显式地调用它们。我不知道微软为什么要把这个问题搞得如此复杂，我希望他们的编译器能够在以后的版本中放宽这些限制。

对于定义了操作符重载方法的类型，微软建议我们在该类型中同时定义一个命名友好的公有实例方法，然后再在该方法内部调用操作符重载方法。例如，我们应该在定义了 op_Addition 或 op_AdditionAssignment 操作符重载方法的类型中再定义一个公有的、命名友好的 Add 方法。表 9.1 第 3 列中显示的就是对应每个操作符推荐使用的友好命名。所以前面展示的 Complex 类型就应该像下面这样定义：

```
class Complex {
    public static Complex operator+(Complex c1, Complex c2) { ... }
    public Complex Add(Complex c) { return(this + c); }
}
```

当然，任何语言编写的代码都可以调用这种类似于 Add 的命名友好的操作符方法。但是微软的这个建议无疑使得整个事情变得更加复杂。我个人认为这种额外的复杂性是完全没有必要的，并且如果 JIT 编译器没有内联这些命名友好的方法，调用它们还会带来额外的性能损失。内联将使 JIT 编译器能够对代码进行优化，消除额外的方法调用，从而提升代码的运行时性能。

9.4 转换操作符方法

有时候，我们可能需要将一个类型的对象转化为另一个类型的对象。例如，大家在自己的编程生涯中一定做过将 Byte 转化为 Int32 的工作。当源类型和目标类型都是编译器认为的基元类型时，编译器将知道怎样产生必要的代码来执行这样的转换。

但是，如果有一个不是编译器认为的基元类型时，编译器将不知道怎样执行转换。例如，假设 FCL 中包括一个 Rational 类型。那么将一个 Int32 或 Single 转换为一个 Rational 肯定会为我们的编程工作带来不少方便。再进一步，将一个 Rational 转换为一个 Int32 或 Single 也将是一件令人惬意的事情。

为了使这些转换成为可能，Rational 类型应该定义一些公有的构造器，这些公有构造器的参数应该为我们希望转换的类型实例。另外，我们还应该为 Rational 类型定义一些不接受任何参数的公有实例方法 ToXxx(就像很常见的 ToString 方法一样)，其中每一个方法将 Rational 类型的实例转换为 Xxx 类型。下面演示了怎样为 Rational 正确定义转换构造器及方法。

```
class Rational {
    // 由一个 Int32 构造一个 Rational
    public Rational(Int32 numerator) { ... }

    // 由一个 Single 构造一个 Rational
    public Rational(Single value) { ... }

    // 将一个 Rational 转换为一个 Int32
    public Int32 ToInt32() { ... }

    // 将一个 Rational 转换为一个 Single
    public Single ToSingle() { ... }
}
```

通过调用这些构造器和方法，使用任何编程语言的开发人员都可以将一个 Int32 或 Single 转换为一个 Rational，或者将一个 Rational 转换为一个 Int32 或 Single。能够进行这些转换会给编程工作带来很多方便，所以当设计一个类型的时候，我们应该仔细考虑定义哪些转换构造器和方法会为我们的类型带来实际的意义。

上一节中我们讨论了一些编程语言如何支持操作符重载。实际上，一些编程语言(例如 C#)还支持转换操作符重载。**转换操作符**本质上仍然是一些方法，它们可以将对象从一个类型转换为另一个类型。在 C#中我们使用特殊的语法来定义转换操作符方法。下面的代码演示了我们为 Rational 类型定义的 4 个转换操作符方法：

```
class Rational {
    // 由一个 Int32 构造一个 Rational
    public Rational(Int32 numerator) { ... }

    // 由一个 Single 构造一个 Rational
    public Rational(Single value) { ... }

    // 将一个 Rational 转换为一个 Int32
    public Int32 ToInt32() { ... }

    // 将一个 Rational 转换为一个 Single
    public Single ToSingle() { ... }

    // 由一个 Int32 隐式构造一个 Rational 并返回
    public static implicit operator Rational(Int32 numerator) {
        return new Rational(numerator);
    }

    // 由一个 Single 隐式构造一个 Rational 并返回
    public static implicit operator Rational(Single value) {
        return new Rational(value);
    }

    // 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();
    }
}
```

和操作符重载方法一样，转换操作符方法也必须为 public 和 static。但是对于转换操作符方法，我们还必须告诉编译器是隐式地产生代码来调用转换操作符方法，还是要求源代码显式地告诉编译器产生代码来调用转换操作符方法。

在 C#中，我们使用 implicit 关键字来告诉编译器，在源代码中不必做显式的转型就可以产生调用转换操作符方法的代码；而使用 explicit 关键字来告诉编译器只有当源代码中指定了显式的转型时，才产生调用转换操作符方法的代码。

在 implicit 或 explicit 关键字后面，我们需要指定 operator 关键字来告诉编译器该方法是一个转换操作符。在 operator 关键字后面，我们还需要指定对象转型时的目标类型；而在括号内，则需要指定对象转型时的源类型。

前面为 Rational 类型定义的转换操作符允许我们在 C#中像下面这样来编写代码：

```
class App {
    static void Main() {
        Rational r1 = 5;            // 将 Int32 隐式转型为 Rational
        Rational r2 = 2.5F;         // 将 Single 隐式转型为 Rational

        Int32 x = (Int32) r1;       // 将 Rational 显式转型为 Int32
        Single s = (Single) r1;     // 将 Rational 显式转型为 Single
    }
}
```

当 C#编译器编译上面的代码时，它会检测到代码中的转型操作，并在内部产生调用 Rational 类型定义的转换操作符方法的 IL 代码。这些方法的名称是什么呢？编译 Rational 类型并查看它的元数据，我们会看到编译器为每一个定义的转换操作符都产生了一个方法。对于 Rational 类型来讲，其 4 个转换操作符方法看起来就像下面的样子：

```
public static Rational op_Implicit(Int32 numerator)
public static Rational op_Implicit(Single value)
public static Int32 op_Explicit(Rational r)
public static Single op_Explicit(Rational r)
```

如我们所见，转换操作符方法的名称总为 op_Implicit 或者 op_Explicit。对于那些转换过程中不可能丢失精度或者数量级的转型操作，我们应该为其定义一个隐式的转换操作符，例如将 Int32 转型为 Rational。而对于那些转换过程中有可能丢失精度或者数量级的转型操作，我们应该为其定义一个显式的转换操作符，例如将 Rational 转型为 Int32。

重要　大家可能注意到了前面两个 op_Explicit 方法接受的是相同类型(Rational)的参数。它们唯一的不同是它们的返回值，其中一个是 Int32，另一个是 Single。CLR 完全支持一个类型定义多个只有返回值类型差别的方法。然而，很少有语言提供了这种能力。大家可能都知道，C++、C#、Visual Basic 以及 Java 就不支持这样做。但确有一些少数的语言(如 IL 汇编语言)允许我们在这些只有返回值类型差别的方法之间进行显式选择调用。当然，如果用 IL 汇编语言来编写代码，我们不应该利用这种能力，因为这样的方法不能被其他的编程语言所调用。虽然 C#编译器没有为开发人员提供这种能力，但是当一个类型定义了多个转换操作符方法时，C#编译器却可以在内部利用这样的机制。

C#完全支持转换操作符。当编译器检测到我们的源代码中正在使用某个类型的实例，而该处代码又期望的是另一种类型的实例时，它会去搜索能够执行这样的转换的隐式转换操作符方法。如果存在，编译器将产生调用该方法的 IL 代码。

当编译器检测到我们的源代码中正在将一个类型的实例显式转型为另一个类型时，它会去搜索能够执行这样的转换的隐式转换操作符方法，或者显式转换操作符方法(译注：在 C#中，不能为一种转型操作同时定义隐式转换操作符和显式转换操作符)。如果存在，编译器将产生调用该方法的 IL 代码。如果编译器找不到一个合适的转换操作符方法，它将报告错误并停止编译。

要真正理解操作符重载方法和转换操作符方法，强烈建议大家将 System.Decimal 类型作为一个研究学习的样板。Decimal 类型定义了好几个构造器来允许我们将各种类型的对象转换为一个 Decimal。当然，它也提供了一些 ToXxx 方法来允许我们将一个 Decimal 对象转换为各种其他的类型。最后，Decimal 还定义了一些转换操作符方法和操作符重载方法。

9.5　引用参数

默认情况下，CLR 假设所有的方法参数都是按值传递的。当参数为引用类型的对象时，参数的传递是通过传递指向对象的引用(或指针)来完成的——注意引用/指针本身是按值传递的。这意味着方法可以改变引用对象，并且调用代码可以看到这种改变的结果。

对于值类型实例的参数来说，传递给方法的将是值类型实例的一个拷贝。这意味着方法会得到一份属于它自己的值类型实例的拷贝，而调用该方法的代码中的实例不会受到任何影响。

> **重要** 对于一个方法，我们必须知道它的每个参数是引用类型的参数，还是值类型的参数，因为我们编写的操作参数的代码会因此有很大的差别。

除了按值传递参数外，CLR 还允许我们按引用的方式来传递参数(译注：按引用的方式传递的参数在本书中又称引用参数，注意区别它和引用类型参数的不同)。在 C#中，我们可以用 out 和 ref 关键字来做到这一点。这两个关键字告诉 C#编译器要产生额外的元数据来表示指定的参数是按引用的方式来传递的；编译器将使用该信息来产生传递参数地址(而不是参数本身的值)的代码。

关键字 out 和 ref 的不同之处在于哪个方法负责初始化参数。如果一个方法的参数被标识为 out，那么调用代码在调用该方法之前可以不初始化该参数，并且被调用方法不能直接读取参数的值，它必须在返回之前为该参数赋值。如果一个方法的参数被标识为 ref，那么调用代码在调用该方法之前必须首先初始化该参数。被调用方法则可以任意选择读取该参数、或者为该参数赋值。

引用类型参数和值类型参数在使用 out 和 ref 关键字时的行为有很大的差别。下面我们先来看一看在值类型参数上使用 out 关键字时的行为：

```
class App {
    static void Main() {
        Int32 x;
        SetVal(out x);              // x 不必被初始化
        Console.WriteLine(x);       // 显示 "10"
    }
    static void SetVal(out Int32 v) {
        v = 10;                     // SetVal 方法必须初始化 v
    }
}
```

在上面的代码中，x 首先被声明在线程的堆栈上。接着，x 的地址被传递给 SetVal。SetVal 的参数 v 是一个指向 Int32 值类型的指针。在 SetVal 内部，v 指向的 Int32 被赋值为 10。当 SetVal 返回后，Main 中 x 的值将为 10，控制台上的显示结果自然也将为"10"。在值类型参数上使用 out 关键字会为代码带来一定的效率提升，因为它避免了值类型实例的字段在方法调用时的拷贝操作。

我们再来看一看在值类型参数上使用ref关键字时的行为：

```
class App {
    static void Main() {
        Int32 x = 5;
        AddVal(ref x);              // x必须被初始化
        Console.WriteLine(x);       // 显示 "15"
    }

    static void AddVal(ref Int32 v) {
        v += 10;                    // AddVal方法可以直接使用经过初始化的v
    }
}
```

在上面的代码中，x首先被声明在线程的堆栈上，紧接着便被初始化为5。随后x的地址被传递给AddVal。AddVal的参数v是一个指向Int32值类型的指针。在AddVal内部，v指向的Int32必须为一个经过初始化的值。这样AddVal才可以在任何表达式中使用该初始值，也可以改变它，并且改变后的值会被"返回"给调用代码。在上面的例子中，AddVal将10加到该初始值上。当AddVal返回后，Main中x的值将为15，自然在控制台上显示的结果也将为"15"。

从IL或者CLR的角度来看，out和ref关键字的行为实际上是一样的：它们都会导致指向实例的指针被传递给方法。两者的不同之处在于编译器会根据它们选择不同的机制来确保我们的代码是正确的。例如，下面的代码试图向一个需要ref参数的方法传递一个未经初始化的值，从而导致编译错误：

```
class App {

    static void Main() {
        Int32 x;                // x 没有被初始化

        // 下面一行将导致编译失败，编译器将产生错误信
        // 息error CS0165: 使用了未赋值的局部变量'x'
        AddVal(ref x);

        Console.WriteLine(x);
    }

    static void AddVal(ref Int32 v) {
        v += 10;
    }
}
```

重要 经常有开发人员问我为什么C#要求调用方法的时候还要指定out或ref。毕竟，编译器知道正在被调用的方法是否需要out或ref，并且应该能够正确地编译代码。的确，编译器是能自动执行正确的操作。但是，C#的设计者们认为调用代码应该清晰地表达它的意图。只有这样，我们在调用代码处才能很明显地看出被调用方法是否有可能改变传入的变量值。

另外，CLR允许我们根据out和ref参数来重载方法。例如，下面的代码就是合法的：

```
class Point {
    static void Add(Point p) { ... }
    static void Add(ref Point p) { ... }
}
```

但是仅通过区分out和ref来重载方法又是不合法的，因为它们经JIT编译后的代码是相同的。所以我们不能在上面的Point类型中再定义下面的方法：

```
static void Add(out Point p) { ... }
```

在值类型参数上使用out和ref关键字与用传值的方式来传递引用类型的参数在某种程度上具有相同的行为。对于前一种情况，out和ref关键字允许被调用方法直接操作一个值类型实例。调用代码必须为该实例分配内存，而被调用方法操作该内存。对于后一种情况，调用代码负责为引用类型对象分配内存，而被调用方法通过传入的引用(指针)来操作对象。基于这种行为，只有当一个方法要“返回”一个它已知的对象引用时，在引用类型参数上使用out和ref关键字才有意义。看下面的代码：

```
class App {

    static public void Main() {
        FileStream fs;

        // 打开第一个待处理文件
        StartProcessingFiles(out fs);
```

```
        // 如果有更多需要处理的文件，则继续
        for (; fs != null; ContinueProcessingFiles(ref fs)) {
            // 处理文件
            fs.Read(...);
        }
    }

    static void StartProcessingFiles(out FileStream fs) {
        fs = new FileStream(...);
    }

    static void ContinueProcessingFiles(ref FileStream fs) {
        fs.Close();    // 关闭上一次操作的文件

        // 打开下一个文件；如果没有文件，则返回null
        if (noMoreFilesToProcess) fs = null;
        else fs = new FileStream (...);
    }
}
```

如我们所见，这段代码中最大的不同在于有着 out 或 ref 修饰的引用类型参数的方法创建一个对象后，指向新对象的指针会被返回给调用代码。另外注意 ContinueProcessingFiles 方法在返回新对象之前可以操作传入的对象，这是因为其参数被标识为 ref。

下面的代码是上述代码的一个简化版本：

```
class App {
    static public void Main() {
        FileStream fs = null;  // 初始化为null (必要的操作)

        // 打开第一个待处理文件
        ProcessFiles(ref fs);

        // 如果有更多需要处理的文件，则继续
        for (; fs != null; ProcessFiles(ref fs)) {

            // 处理文件
            fs.Read(...);
        }
    }

    static void ProcessingFiles(ref FileStream fs) {
        // 如果先前的文件打开的，则将其关闭
        if (fs != null) fs.Close();     // 关闭上一次操作的文件
```

```
        // 打开下一个文件；如果没有文件，则返回null
        if (noMoreFilesToProcess) fs = null;
        else fs = new FileStream (...);
    }
}
```

下面的例子演示了怎样使用ref关键字来交换两个引用类型：

```
static public void Swap(ref Object a, ref Object b) {
    Object t = b;
    b = a;
    a = t;
}
```

要交换两个String对象引用，大家可能会考虑像下面这样做：

```
static public 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"
}
```

然而，这段代码实际上不会通过编译。问题在于按引用方式传递的变量必须和方法声明的参数类型完全相同。换句话说，Swap期望的是两个指向Object类型的引用，而不是两个指向String类型的引用。要交换两个String引用，我们必须像下面这样做：

```
static public void SomeMethod() {
    String s1 = "Jeffrey";
    String s2 = "Richter";
    ...
    // 以引用方式传递的变量必须和
    // 方法期望的参数类型完全匹配
    Object o1 = s1, o2 = s2;
    Swap(ref o1, ref o2);

    // 将对象转型为字符串
    s1 = (String) o1;
    s2 = (String) o2;

    Console.WriteLine(s1); // 显示 "Richter"
    Console.WriteLine(s2); // 显示 "Jeffrey"
}
```

可以看到，修正后的 SomeMethod 会通过编译，并且会按我们所期望的行为执行。C#要求以引用方式传递的参数必须和方法期望的参数完全匹配的目的是为了确保类型安全。下面的代码(幸好它不会通过编译)展示了如果参数的类型不匹配可能导致的类型安全漏洞。

```
class SomeType {
   public Int32 val;
}

class App {
   static void Main() {
      SomeType st;

      // 下面一行将产生编译错误 error CS1503: 参数 '1' :
      // 无法从'out SomeType' 转换为 'out object'
      GetAnObject(out st);

      Console.WriteLine(st.val);
   }

   static void GetAnObject(out Object o) {
      o = new String('X', 100);
   }
}
```

在这段代码中，Main 期望 GetAnObject 返回一个 SomeType 对象。但是，因为 GetAnObject 的签名表示的是一个指向 Object 的引用，所以 GetAnObject 可以将 o 初始化为一个任何类型的对象。当 GetAnObject 返回到 Main 中时，st 将指向一个 String，这显然不是一个 SomeType 对象，对 Console.WriteLine 的调用自然会失败。幸运的是，C#编译器不会编译上面的代码，因为 st 是一个指向 SomeType 的引用，而 GetAnObject 要求的是一个指向 Object 的引用。

9.6 可变数目参数

有时候能够定义一个接受可变数目参数的方法会为我们的开发工作带来很多方便。例如，System.String 类型提供的一些方法就允许我们将任意个数的字符串连接在一起，还有一些方法允许我们指定一组被一起格式化的字符串。

下面演示了如何声明一个接受可变数目参数的方法：

```
static Int32 Add(params Int32[] values) {
   // 注意：如果愿意，我们可以
   // 将该数组传递给其他方法
```

```
        Int32 sum = 0;
        for (Int32 x = 0; x < values.Length; x++)
            sum += values[x];
        return sum;
    }
```

除了方法签名中的params关键字外，其他的大家都已经很熟悉了。目前我们先忽略params关键字。很明显，Add方法接受一个Int32数组引用，然后遍历数组中的元素，并将它们加在一起，最后返回相加的结果。

因为Add的参数是一个数组，所以我们可以像下面这样调用该方法：

```
static void Main() {
    // 显示 "15"
    Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 }));
}
```

显然，我们的数组可以用任意数目的元素初始化后，再传递给Add方法。虽然上面的代码会通过编译并正确运行，但看上去有些丑陋。我们当然更喜欢像下面这样来调用Add：

```
static void Main() {
    // 显示 "15"
    Console.WriteLine(Add( 1, 2, 3, 4, 5 ));
}
```

之所以能够这样做，是因为params关键字的缘故。params关键字告诉编译器在指定的参数上应用一个System.ParamArrayAttribute定制特性的实例(第16章将讨论定制特性)。因为params关键字仅仅是该特性的一个缩写，所以Add方法的定义原型可能应该像下面的样子：

```
static Int32 Add([ParamArray] Int32[] values) {
    ...
}
```

当C#编译器检测到方法调用时，它会使用指定的名称检查所有不含ParamArrayAttribute特性的方法。如果存在一个方法满足该调用，编译器会产生必要的代码来调用该方法。但是，如果编译器不能找到一个匹配，它会查找带有ParamArrayAttribute特性的方法看其是否能够满足调用要求。如果满足，它会首先产生一系列指令来构造数组，以及用指定的元素填充数组，在这些工作完成之后它才会产生调用方法的代码。

在前面的例子中，我们并没有定义带有5个参数类型和Int32兼容的Add方法。但是，编译器会看到源代码中定义了另外一个Add方法，并且其Int32数组参数应用有一个ParamArrayAttribute特性。于是，编译器认定此为一个匹配，并产生代码将传入的5个参数放进一个Int32数组中，然后再调用该Add方法。这使得我们编写传递一组参数给Add方法的代码大为简化。然而，编译器产生的指令显示，这种简化的代码和第一个版本的代码(即由我们自己显式构造并初始化数组的那个版本)实际上是一样的。

注意，只有方法的最后一个参数才可以用params关键字(即ParamArrayAttribute特性)来标记。该参数必须为一个一维数组，但类型可以任意。传递null或者一个0长数组给该参数是合法的(译注：但是传入null后，编译器并不保证不会出现空引用异常，例如用Add(null)调用前面定义的Add方法就会抛出一个System.NullReferenceException异常，所以最好对传入的参数做一下检测)。例如，下面对Add的调用会正常编译并运行，且产生总和为0的结果——正是我们的期望值。(译注：上面对Add方法的调用Add()并不等同于Add(null)，而是等同于Add(new Int32(){})，即传入一个0长数组的引用。)：

```
static void Main() {
    // 显示 "0"
    Console.WriteLine(Add());
}
```

目前所有的例子都向我们展示了怎样编写一个接受多个Int32参数的方法。但怎样编写接受多个任意类型参数的方法呢？答案很简单：仅仅改变方法的原型使其接受一个Object数组即可。下面的方法显示了每个传入对象的类型：

```
class App {
    static void Main() {
        DisplayTypes(new Object(), new Random(), "Jeff", 5);
    }

    static void DisplayTypes(params Object[] objects) {
        foreach (Object o in objects)
            Console.WriteLine(o.GetType());
    }
}
```

编译并运行上面的代码将产生以下输出：

```
System.Object
System.Random
System.String
System.Int32
```

9.7 虚方法的调用机理

方法表示可以在类型(静态方法)或者类型实例(非静态方法)上执行某些操作的代码。所有的方法都有一个名称、一个签名以及一个返回值。一个类型可以有多个同名的方法，只要每个方法的参数集合或者返回值互不相同即可。(译注：这里的参数集合更准确地讲应该是参数列表，即集合中的参数是有顺序的。两个参数列表相同当且仅当其中的参数个数相同，并且相同位置的参数类型也相同。)所以定义两个名称和参数集合相同，而返回值不同的方法是可能的。但是，除了 IL 汇编语言外，我目前还没有看到哪个语言使用了这种特性；大多数语言在确定方法的惟一性时，都要求名称相同的方法要有一个不同的参数集合，而忽略返回值的差别。

通过检查元数据，CLR 可以确定一个非静态方法是否为一个虚方法。然而，CLR 在调用方法时并不使用该信息。相反，CLR 提供了两个 IL 指令来调用方法：call 和 callvirt。其中 call 指令根据引用变量的类型(译注：即引用变量的静态类型、声明类型)来调用一个方法，而 callvirt 指令根据引用变量指向的对象类型(译注：即引用变量的动态类型、实际类型)来调用一个方法。当编译源代码时，编译器知道代码中是否在调用一个虚方法，并据此产生 call 或 callvirt 指令。这意味着 CLR 有可能以调用非虚方法的方式来调用一个虚方法，这种技巧通常在代码调用一个定义在基类中的虚方法时使用，看下面的代码：

```
class SomeClass {
    // ToString 是一个定义在基类 Object 中的虚方法
    public override String ToString() {
        // 编译器用 'call' IL 指令来调用 Object 的
        // ToString 方法(即调用非虚方法的方式)。
        // 如果编译器使用 'callvirt' 而不是'call'，
        // ToString 方法将递归地调用自己直到堆栈溢出
        return base.ToString();
    }
}
```

在用一个密封类型的引用调用虚方法时，编译器通常也会产生 call 指令。产生 call 而不是 callvirt 会提高代码的性能，因为 CLR 不必检查引用对象的实际类型。另外，对于值类型(总是密封类型)，使用 call 会阻止值类型实例被装箱，从而可以减少内存和 CPU 资源的占用。(译注：前面说过 C#有可能用 call 来调用一个虚方法，实际上 C#也有可能用 callvirt 来调用一个非虚方法。这种情况往往发生在调用一个引用类型的非虚方法的时候。C#这样做的原因是，如果引用变量为 null 时，调用 callvirt 会抛出 System.NullReferenceException 异常，但调用 call 却不会抛出 System.NullReferenceException 异常。而 C#的语言规范要求在一个空引用上调用任何方法都应该抛出 System.NullReferenceException 异常。当然这样做会带来一些性能损失，因为 callvirt 会去做不必要的动态类型辨析。)

不管最终是通过 call 还是 callvirt 来调用一个实例方法，所有的实例方法调用都会接受一个隐藏的 this 指针作为方法的第一个参数。其中 this 指针指向当前正在操作的对象。

9.8 虚方法的版本问题

在过去，一个公司往往负责构建应用程序的所有代码。如今，一个公司的应用程序的很多部分都来自于许多不同的公司。例如，很多应用程序都会用到其他公司创建的组件——实际上，COM(以及COM+)和.NET 技术鼓励这么做。当应用程序包含了由许多公司创建和分发的组件时，各种版本问题便会浮出水面。

在第 3 章解释强命名程序集、以及讨论一个管理员怎样确保应用程序绑定的就是它当初生成和测试时的程序集时，我们曾经探讨过其中的一些问题。然而，还有一些其他的版本问题会损害源代码的兼容性。例如，如果一个类型被用作其他类型的基类型，那么当我们在其中添加或者修改成员时就要非常小心。来看一些例子。

假设 CompanyA 设计了一个类型 Phone:

```
namespace CompanyA {
    class Phone {
        public void Dial() {
            Console.WriteLine("Phone.Dial");
            // 执行一些拨电话的动作
        }
    }
}
```

再假设 CompanyB 使用 CompanyA 的 Phone 作为基类又定义了另一个类型 BetterPhone(译注：应该将 Phone 和 BetterPhone 都声明为 public，因为这样才可以将它们放在不同的程序集中实现。下同)：

```
namespace CompanyB {
    class BetterPhone : CompanyA.Phone {
        public void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }
        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 执行一些建立连接的动作
        }
    }
}
```

当 CompanyB 试图编译这段代码时，C#编译器会产生以下警告信息：“warning CS0108: ‘CompanyB.BetterPhone.Dial()’上要求关键字 new，因为它隐藏了继承成员‘CompanyA.Phone.Dial()’。”

该警告通知开发人员 BetterPhone 中正在定义的 Dial 方法会隐藏 Phone 中定义的 Dial 方法。这个新方法可能会改变 Dial 方法(即原来在 CompanyA 中定义的 Dial 方法)的语义。

编译器对这种潜在的语义错乱给予警告是一个很好的特性。编译器还告诉我们通过在 BetterPhone 类的 Dial 定义前添加 new 关键字可以消除这种警告。下面是修正后的 BetterPhone 类：

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

        // 该方法和 Phone 中的 Dial 无关
        new public void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 执行一些建立连接的动作
        }
    }
}
```

这样 CompanyB 就可以在自己的应用程序中使用 BetterPhone 了。下面是 CompanyB 可能会编写的示例代码：

```
class App {
    static void Main() {
        CompanyB.BetterPhone phone = new CompanyB.BetterPhone();
        phone.Dial();
    }
}
```

执行上面的代码，将得到以下输出：

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
```

该输出显示 CompanyB 得到了它所期望的行为。显然，Main 中调用的是 BetterPhone 内新定义的 Dial 方法，在 Dial 内部它又调用了虚方法 EstablishConnection 和基类 Phone 中的 Dial 方法。

现在假设有几个公司决定使用 CompanyA 的 Phone 类型。再假设这些公司认为在 Dial 方法中建立连接是一个很有用的特性，他们将这种愿望反馈给 CompanyA，CompanyA 就对 Phone 类做了一些修改：

```
namespace CompanyA {
    class Phone {
        public void Dial() {
            Console.WriteLine("Phone.Dial");
            EstablishConnection();
            // 执行一些拨电话的动作
        }

        protected virtual void EstablishConnection() {
            Console.WriteLine("Phone.EstablishConnection");
            // 执行一些建立连接的动作
        }
    }
}
```

现在当 CompanyB 再编译它的 BetterPhone 类型时(继承自新版的 CompanyA 的 Phone)，编译器将产生警告信息：“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”。

编译器警告我们 Phone 和 BetterPhone 都提供了 EstablishConnection 方法，并且两个方法的语义可能不一样：简单地重新编译 BetterPhone 不会得到先前使用第一个版本的 Phone 时的行为。

如果 CompanyB 认为两个类型中的 EstablishConnection 方法语义不同，那么它可以告诉编译器 BetterPhone 中定义的 Dial 和 EstablishConnection 方法是“正确”的方法，并且它们和 Phone 基类型中定义的方法没有任何关系。CompanyB 可以在 Dial 和 EstablishConnection 方法上添加 new 来告知编译器这一点：

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

        // 保留 'new' 告诉编译器该方法和
        // 基类中的 Dial 没有任何关系
        new public void Dial() {
            Console.WriteLine("BetterPhone.Dial");
            EstablishConnection();
            base.Dial();
        }
```

```
        // 添加 'new' 告诉编译器该方法和基类中
        // 的 EstablishConnection 没有任何关系
        new protected virtual void EstablishConnection() {
            Console.WriteLine("BetterPhone.EstablishConnection");
            // 执行一些建立连接的动作
        }
    }
}
```

在这段代码中，new 关键字会告诉编译器产生相关的元数据，CLR 根据产生的元数据可以知道 BetterPhone 中的 Dial 和 EstablishConnection 方法应该被看成新引入的方法，它们和 Phone 中的方法没有任何关系。

注意　如果不用 new 关键字，BetterPhone 的开发人员可能不能使用 Dial 和 EstablishConnection 这样的方法名。这很可能会在整个源代码库中导致一种连锁反应，从而破坏源代码和二进制的兼容性。我们通常不应该做这种大面积的修改，尤其是在大中型项目中。然而，如果改变方法名称仅仅导致源代码一些有限的改动，这样的改变还是有必要的，因为这样可以避免两个类中不相关的 Dial 和 EstablishConnection 搞乱其他开发人员。

利用修改后的 Phone 和 BetterPhone，再执行同样的应用程序代码(即前面 Main 方法中的代码)，我们将会得到以下输出：

```
BetterPhone.Dial
BetterPhone.EstablishConnection
Phone.Dial
Phone.EstablishConnection
```

从输出可以看到，Main 首先调用了 BetterPhone 中新定义的 Dial。Dial 然后调用了 BetterPhone 中定义的 EstablishConnection 虚方法。当 BetterPhone 中的 EstablishConnection 虚方法返回后，Phone 中的 Dial 方法接着被调用，其中又调用了 EstablishConnection 方法。但是由于 BetterPhone 中的 EstablishConnection 方法被标记为 new，所以它不被认为是在重写 Phone 中的 EstablishConnection 虚方法。这样，Phone 中的 Dial 方法调用的就是 Phone 中定义的 EstablishConnection 方法——这也是期望的行为。

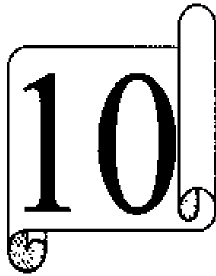
也有另外一种情况，CompanyB 可能在得到 CompanyA 定义的新版 Phone 类型后，认为 Phone 中的 Dial 和 EstablishConnection 方法的语义正是它所期望的。在这种情况下，CompanyB 可以完全删除 BetterPhone 中的 Dial 方法。另外，由于 CompanyB 现在希望告诉编译器 BetterPhone 中定义的 EstablishConnection 方法和 Phone 中定义的 EstablishConnection 方法是相关联的，所以 new 关键字就必须被去掉。然而，简单地去掉 new 关键字是不够的，因为现在编译器不能准确地判断出 BetterPhone 中 EstablishConnection 方法的意图。为了准确地表达意图，CompanyB 的开发人员必须将 BetterPhone 中 EstablishConnection 方法上的 virtual 改为 override。下面的代码展示了新版的 BetterPhone：

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

      // 删除 Dial 方法(直接从基类中继承 Dial)

      // 删除 'new' 并将 'virtual' 改为 'override' 使
      // 该方法和基类的 EstablishConnection 建立联系
      protected override void EstablishConnection() {
         Console.WriteLine("BetterPhone.EstablishConnection");
         // 执行一些建立连接的动作
      }
   }
}
```

现在再执行同样的应用程序代码(即前面 Main 方法中的代码)，我们将会得到以下输出：

```
Phone.Dial
BetterPhone.EstablishConnection
```

从输出可以看到，Main 调用的是 Phone 中定义的 Dial 方法(BetterPhone 直接继承了该方法)。当 Phone 中的 Dial 方法调用 EstablishConnection 虚方法时，它调用的将是 BetterPhone 中定义的 EstablishConnection 方法，因为 BetterPhone 重写了 Phone 中定义的 EstablishConnection 虚方法。



属　性

本章讨论属性。属性允许源代码以一种简化的语法来调用方法。通用语言运行时(CLR)提供了两种属性：无参属性和含参属性。前者通常就被称为**属性**(property)，后者在各种语言中有各自的叫法。例如，C#称含参属性为**索引器**(indexer)，而 Visual Basic 称含参属性为**默认属性**(default property)。

10.1 无参属性

许多类型都定义了可被外部代码读取或者改变的状态信息。这些状态信息通常被实现为类型的字段成员。例如，下面的类型定义中就包含有两个字段：

```
public class Employee {
   public String Name;    // 雇员姓名
   public Int32 Age;      // 雇员年龄
}
```

如果我们创建了一个该类型的实例，我们便可以很容易像下面这样读取或者设置它的状态信息：

```
Employee e = new Employee();
e.Name = "Jeffrey Richter";        // 设置雇员姓名
e.Age = 35;                        // 设置雇员年龄

Console.WriteLine(e.Name);         // 显示 "Jeffrey Richter"
```

像上面那样查询或者设置对象状态信息的做法十分常见。然而，建议大家不要那样做。面向对象设计和编程的其中一个原则就是**数据封装**(data encapsulation)。这意味着我们不应该将类型的字段以公有的方式提供给外界，因为外部代码很容易错误地使用这些字段，从而破坏对象的状态。例如我们很容易像下面这样破坏一个 Employee 对象：

```
e.Age = -5;  // 一个人的年龄怎么可能是 -5 呢?
```

还有一些原因支持我们对一个类型的数据字段的访问进行封装。例如，我们可能希望在访问一个字段的同时执行一些额外的操作，或者缓存某个数值，或者延迟创建某个内部对象。我们也可能希望对字段的访问是线程安全的。或者字段可能只是一个逻辑表示，其真正的值并不是以字节的形式存放在内存中，而是通过某种算法计算得来的。

基于上述这些理由，强烈建议大家在设计类型时，将其所有字段的访问限制都设为私有方式，或者至少是保护方式——永远不要设为公有方式，然后再以方法的形式让用户读取或者设置对象的状态信息。封装了对字段访问的方法典型地被称为**访问器方法**(accessor method)。访问器方法可以选择执行任何的数据合理性检查来确保对象的状态不会被破坏。例如，下面是对前面的类的一个改写：

```
public class Employee {

    private String Name;        // 私有字段
    private Int32 Age;          // 私有字段

    public String GetName() {
        return(Name);
    }

    public void SetName(String value) {
        Name = value;
    }

    public Int32 GetAge() {
        return(Age);
    }

    public void SetAge(Int32 value) {
        if (value < 0)
            throw new ArgumentOutOfRangeException(
                "Age must be greater than or equal to 0");
        Age = value;
    }
}
```

虽然这是一个简单的例子，但我们也能从中看到封装数据字段带来的很多好处。实现只读或者只写属性也变得非常方便：仅仅实现其中一个访问器方法就可以了。

然而，上面这种封装数据字段的做法有两个缺点：首先，我们需要编写更多的代码，因为我们必须实现额外的方法；其次，类型的用户必须调用方法而不是简单地引用字段名称。

```
e.SetAge(35);          // 修改年龄
e.SetAge(-5);          // 抛出 ArgumentOutOfRangeException 异常
```

就个人而言，我认为这些缺点是微不足道的。然而，CLR 提供了一种称作属性的机制稍微缓和了第 1 种缺点，并完全消除了第 2 种缺点。

下面演示的是一个使用了属性的类，它的功能和前面的实现是等同的：

```
public class Employee {
   private String _Name;           // 添加 '_' 避免命名冲突
   private Int32 _Age;             // 添加 '_' 避免命名冲突

   public String Name {
      get { return(_Name); }
      set { _Name = value; }       // 关键字 'value' 表示新设的值
   }

   public Int32 Age {
      get { return(_Age); }
      set {
          if (value < 0)           // 关键字 'value' 表示新设的值
             throw new ArgumentOutOfRangeException(
                "Age must be greater than or equal to 0");
         _Age = value;
      }
   }
}
```

如我们所见，属性使得类型的定义变得有些复杂化了。但使用它们的代码写起来却改善了许多：

```
e.Age = 35;            // 修改年龄
e.Age = -5;            // 抛出 ArgumentOutOfRangeException 异常
```

我们可以将属性看作是一种智能字段(smart field)，这里“智能”的含义是在它们的后端有一个额外的逻辑处理。CLR 支持静态属性、实例属性和虚属性。另外，属性可以标记任何的访问限定修饰符(第 7 章中有讨论)，也可以被定义在接口中(第 15 章中有讨论)。

每个属性都有一个名称和一个类型(不能为 void)。属性不能被重载(即两个属性名称相同，但类型不同)。当定义一个属性时，我们一般会同时指定一个 get 方法和一个 set 方法。但我们也可以通过去掉 set 方法来定义一个只读属性、或者通过去掉 get 方法来定义一个只写属性。

通过属性的 get/set 方法来操作私有字段的做法十分常见。这时的字段通常被称作属性的**后端字段**(backing field)。但是，get 和 set 方法并非必须要访问一个后端字段。System.Threading.Thread 类型提供的与操作系统直接通信的 Priority 属性就是一个例子，Thread 对象并没有为线程的优先级维护一个字段。另一个例子是那些在运行时计算的只读属性——例如，一个以 0 结束的数组(zero-terminated array)的长度，或者一个矩形的面积(知道了它的高度和宽度)。

当定义一个属性时，编译器会在生成的托管模块中产生以下 3 项：

- 一个表示属性的 get 访问器的方法。只有在为属性定义了 get 访问器方法时，才有这一项。
- 一个表示属性的 set 访问器的方法。只有在为属性定义了 set 访问器方法时，才有这一项。
- 一个位于托管模块元数据中的属性定义，不管是只读、只写、或者读写属性都有这一项。

再来看前面定义的 Employee 类型。编译器在编译该类型时，它会发现其中定义的 Name 和 Age 两个属性。因为这两个属性都有 get 和 set 访问器方法，所以编译器会在 Employee 类型中产生 4 个方法定义，这使得 Employee 类型就好像编译自下面的代码一样：

```
public class Employee {

   private String _Name;          // 添加 '_' 避免命名冲突
   private Int32 _Age;            // 添加 '_' 避免命名冲突

   public String get_Name(){
      return _Name;
   }
   public void set_Name(String value) {
      _Name = value;              // 'value' 表示新设的值
   }
```

```
    public Int32 get_Age() {
        return _Age;
    }

    public void set_Age(Int32 value) {
        if (value < 0)              // 'value' 表示新设的值
            throw new ArgumentOutOfRangeException(
                "Age must be greater than or equal to 0");
        _Age = value;
    }
}
```

C#编译器通过在我们指定的属性名称前添加 get_和 set_来自动为生成的访问器方法命名。

C#对属性有内置的支持。当 C#编译器看到有代码试图读取或者设置一个属性时，它实际上会产生对相应方法的一个调用。如果我们使用的编程语言不直接支持属性，我们仍可以通过调用期望的访问器方法来访问属性。两者的效果是一样的，只是后者实现起来的代码看起来不够漂亮而已。

除了产生访问器方法外，编译器还会在托管模块的元数据内为源代码中定义的每一个属性产生一个属性定义条目。该条目中包含了一些标记和属性的类型，并有一个对 get 和 set 访问器方法的引用。这些信息仅仅为属性和它的访问器方法之间提供了一层关联关系。编译器和其他一些工具可以使用这些信息，我们也可以通过 System.Reflection.PropertyInfo 类来获取它们。但 CLR 本身并不使用它们，它在运行时仅需要访问器方法即可。

因为访问属性和访问字段有着相同的语法表达，所以只有对那些执行时间比较短的操作，我们才应该使用属性。按通常的习惯，采用这种语法的代码不宜花费太长的执行时间。对于那些执行时间较长的操作，则应该使用方法来完成。例如计算一个矩形的面积通常就很快，这时采用只读属性就比较好。但是计算链表集合中元素的个数可能会很慢，这时可能就应该采用方法、而不是只读属性。

对于简单的 get 和 set 访问器方法，JIT 编译器会将代码进行内联(inline)处理，这样使用属性时就不会再有运行时的性能损失(相对于字段访问来讲)。内联会将一个方法(或者访问器方法)内的代码直接编译到调用它们的方法中，从而消除了调用方法时的运行时负担，但它的代价是编译后的方法代码变得较为庞大。由于属性的访问器方法通常包含有很少的代码，所以内联它们会使代码变得更小，执行效率也更高。

10.2 含参属性

在前面一节中，属性的 get 访问器方法没有接受任何参数。因此，我称之为**无参属性**(parameterless property)。这些属性很容易理解，因为对它们的访问很类似于对字段的访问。除了这种类似于字段的属性，CLR 还支持一种 get 访问器方法接受一个或者多个参数的属性，我个人将其称作**含参属性**(parameterful property)。不同的编程语言以不同的方式提供含参属性。如本章开始所述，各种语言中对含参属性有不同的术语：C#称它们为**索引器**(indexer)，Visual Basic 称它们为**默认属性**(default property)，而托管扩展 C++称它们为**索引属性**(index property)。本节将主要讨论 C#中的含参属性，也就是索引器。

在 C#中，含参属性(索引器)可以用类似数组的语法来访问。换句话说，我们可以将索引器看作是重载 [] 操作符的一种方式。下面演示的是一个 BitArray 类型的例子，它允许我们用类似数组的语法来访问该类型实例中保存的一组位。

```
public class BitArray {
    // 一个用于保存位的私有字节数组
    private Byte[] byteArray;
    private Int32 numBits;

    // 下面的构造器用于分配字节数组，并将所有的位设为 0
    public BitArray(Int32 numBits) {
        // 首先检验参数的有效性
        if (numBits <= 0)
            throw new ArgumentOutOfRangeException("numBits","numBits must be>0");
        // 保存位的个数
        this.numBits = numBits;
        // 为位数组分配字节
        byteArray = new Byte[(numBits + 7) / 8];
    }

    // 下面是一个索引器
    public Boolean this[Int32 bitPos] {
        // 下面是索引器的 get 访问器方法
        get {
            // 首先检验参数的有效性
            if ((bitPos < 0) || (bitPos >= numBits))
                throw new IndexOutOfRangeException();
            // 返回指定索引上的位的状态
            return (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 >= numBits))
                throw new IndexOutOfRangeException();

            if (value) {
                // 将指定索引上的位设为真值
                byteArray[bitPos / 8] = (Byte)
                    (byteArray[bitPos / 8] | (1 << (bitPos % 8)));
            } else {
                // 将指定索引上的位设为假值
                byteArray[bitPos / 8] = (Byte)
                   (byteArray[bitPos / 8] & ~(1 << (bitPos % 8)));
            }
        }
    }
}
```

使用BitArray类型的索引器非常简单：

```
// 分配一个含14个位的BitArray
BitArray ba = new BitArray(14);

// 调用set访问器方法将所有的偶数位设为真值
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BitArray例子中，索引器接受一个Int32类型的参数bitPos。所有的索引器都必须至少有一个参数，也可以有多个。这些参数(以及返回值)可以为任何类型。

创建一个接受Object作为参数的索引器来在一个相关的数组中查询数值的做法十分常见。实际上，System.Collections.Hashtable类型就提供了一个这样的索引器，其接受一个键(Object类型)，并返回和此键相关的一个值(同样也为Object类型)。不像无参属性，一个类型可以提供多个重载的索引器，只要它们的签名不同即可。

与无参属性的set访问器方法类似，索引器的set访问器方法也有一个隐含的参数(C#中称之为value)，该参数表示“被索引元素”期望的新值。

CLR本身并不区分无参属性和含参属性(索引器)。对于CLR来讲，属性仅仅是定义在类型中的一对(或一个)方法而已。如前所述，不同的语言要求用不同的语法来创建和使用含参属性。C#要求用this[...]作为表达索引器的语法仅仅是C#编译器组选择的方案而已。这种选择意味着C#只允许我们在对象实例上定义索引器。虽然CLR支持静态含参属性，但是C#却没有为我们提供定义静态索引器(含参属性)的语法。

因为CLR对待含参属性和无参属性的方式是一样的，所以编译器在编译含有索引器的类型时，会在生成的托管模块中产生同样的3项：

- 一个表示含参属性的get访问器的方法。只有在为含参属性定义了get访问器方法时，才有这一项。
- 一个表示含参属性的set访问器的方法。只有在为含参属性定义了set访问器方法时，才有这一项。
- 一个位于托管模块元数据中的属性定义，不管是只读、只写或者读写属性都有这一项。不存在特别的含参属性元数据定义表，因为对CLR来讲，含参属性就是属性。

对于前面演示的BitArray类型，编译器编译其中的索引器产生的结果就像编译自下面的代码一样：

```
public class BitArray {
    ...
    // 下面是索引器的get访问器方法
    public Boolean get_Item(Int32 bitPos) { ... }

    // 下面是索引器的set访问器方法
    public void set_Item(Int32 bitPos, Boolean value) { ... }
}
```

C#编译器通过在Item前添加get_或set_来自动为生成的访问器方法命名。因为C#不允许我们为索引器指定名称，所以C#编译器组不得不为访问器方法选择一个名称，Item就是他们选择的结果。另外，一个类型可以定义多个索引器，前提是这些索引器的参数集合各不相同。

当查看.NET框架参考文档时，我们通常可以通过查找名为Item的属性来判断一个类型是否提供了索引器。例如，System.Collections.SortedList类型就提供了一个名为Item的公有实例属性，该属性就是SortedList的一个索引器。

当使用C#编程时，我们永远也不会看到Item这一名称，所以通常我们无需关心编译器在这个上面所做的选择。但是如果我们设计的索引器要被其他语言所访问，那么我们可能希望改变编译器自动产生的名称。C#允许我们通过在索引器上应用System.Runtime.CompilerServices.IndexerNameAttribute特性来实现这一点。

看下面的代码：

```
public class BitArray {
    ...
    [System.Runtime.CompilerServices.IndexerName("Bit")]
    public Boolean this[Int32 bitPos] {
        ...
    }
}
```

上面的代码会导致编译器产生get_Bit和set_Bit访问器方法(而不是get_Item和set_Item)。

下面是一段Visual Basic代码，其中访问了上面C#编写的索引器：

```
' 构造一个BitArray类型实例
Dim ba As new BitArray(10)
...
' Visual Basic使用()而不是[]来指定数组元素
Console.WriteLine(ba(2))        ' 显示True或False

' Visual Basic也允许我们通过名称来访问索引器
Console.WriteLine(ba.Bit(2))  ' 显示结果和上一行相同
```

在某些语言中，IndexerName特性允许我们定义多个签名相同的索引器，因为每个索引器的名称可以不同。但是，C#不允许我们这么做，因为它不能通过名称来引用索引器；实际上，编译器根本就不知道我们引用的是哪个索引器。例如，编译下面的代码会产生以下错误："error CS0111: 类'SomeType'已经定义了一个具有相同参数类型的名为'this'的成员"。

```
using System;
using System.Runtime.CompilerServices;

public class SomeType {

    // 定义一个get_Item访问器方法
    public Int32 this[Boolean b] {
        get { return 0; }
    }

    // 定义一个get_Jeff访问器方法
    [IndexerName("Jeff")]
    public String this[Boolean b] {
        get { return ""; }
    }
}
```

我们可以很清楚看到C#将索引器看作是重载 [] 操作符的一种方式，而该操作符不能用来辨析具有不同方法名称的含参属性。

顺便提一句，System.String 类型就是一个改变了索引器名称的例子。String 索引器的名称为 Chars，而非 Item。该属性允许我们得到一个字符串中的单个字符。对于不使用 [] 操作符语法来访问含参属性的编程语言来讲，Chars 显然是一个更有意义的名称。

选择主要的含参属性

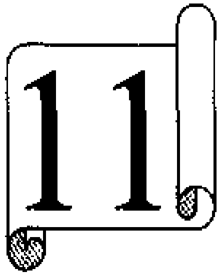
C#对索引器的限制会导致以下问题：如果其他语言的开发人员在一个类型中定义了多个不同名称的含参属性，那么C#如何使用该类型？答案是该类型必须选择一个含参属性方法名作为默认的属性，这可以通过在类型上应用一个 System.Reflection.DefaultMemberAttribute 特性的实例来实现。根据文档记录，DefaultMemberAttribute 特性可以应用在类、结构、或者接口上。在 C#中，当我们编译一个定义有含参属性的类型时，C#编译器会自动在该类型上应用一个 DefaultMemberAttribute 特性的实例。DefaultMemberAttribute 的构造器指定了用于一个类型默认含参属性的名称。

因此，如果我们在C#中定义了一个具有含参属性的类型，并且没有指定 IndexerName 特性，那么该类型将有一个表示索引器名称为 Item 的 DefaultMember 特性。但如果我们在一个含参属性上应用了 IndexerName 特性，那么该类型将有一个表示索引器为指定名称(由 IndexerName 特性决定)的 DefaultMember 特性。记住，如果我们的代码中包含有多个不同名称的含参属性，C#编译器将会报错。

如果我们在其他的语言中为一个类型定义了多个不同名称的含参属性，那么我们必须在其中选择一个，并用 DefaultMember 特性来标识。这是 C#能够访问的惟一一个含参属性。

当C#编译器遇到试图读取或者设置索引器的代码时，编译器实际上会产生对get或者set访问器方法的一个调用。某些编程语言可能不支持含参属性。要在这些语言中访问含参属性，我们必须显式调用期望的访问器方法。

对于CLR来讲，无参属性和含参属性之间没有任何区别，所以我们同样可以使用System.Reflection.PropertyInfo类来查找一个含参属性与它的访问器方法之间的关联关系。JIT编译器也会自由地选择是否将一个访问器方法的代码内联到调用方法的代码中。



事　件

本章讨论类型中可以定义的最后一种成员：**事件**(event)。定义了事件成员的类型允许类型(或者类型的实例)在某些特定事情发生的时候通知其他对象。例如，假设 Button 类定义了一个名为 Click 的事件。当 Button 对象被点击时，应用程序中的一些对象可能希望能够收到一个通知、并执行一些动作。作为一种类型成员，事件使得这种交互成为可能。具体而言，定义一个事件成员意味着类型为我们提供了以下三种能力：

- 允许对象登记该事件。
- 允许对象注销该事件。
- 允许定义事件的对象维持一个登记对象的集合，并在某些特定的事情发生时通知这些对象。

CLR 的事件模型建立在**委托**(delegate)这一机制之上。委托为我们提供了一种类型安全的方式来调用回调方法。本章会用到委托，但对其深入全面的解释要放到第 17 章。

为了帮助大家完全理解 CLR 中事件的工作机制，这里首先向大家描述一个使用事件的场景。假设我们希望设计一个电子邮件应用程序。当一个电子邮件消息到达时，用户可能希望将该消息转发给一个传真机(Fax)或者一个寻呼机(Pager)。在设计该应用程序时，假设我们首先会设计一个名为 MailManager 的类型负责接受发进来的电子邮件消息。然后再为 MailManager 类型定义一个名为 MailMsg 的事件。其他类型(例如 Fax 和 Pager)则可以登记该事件。当 MailManager 收到一个新的电子邮件消息时，它会触发该事件，将消息分发给每一个登记对象。各个对象则以自己期望的方式来处理该消息。

当应用程序初始化时，我们仅实例化一个 MailManager 对象—— 应用程序随后可以实例化任意数量的 Fax 和 Pager 类型。图 11.1 演示了应用程序的初始化过程，以及当一个新的电子邮件消息到达时发生的事情。

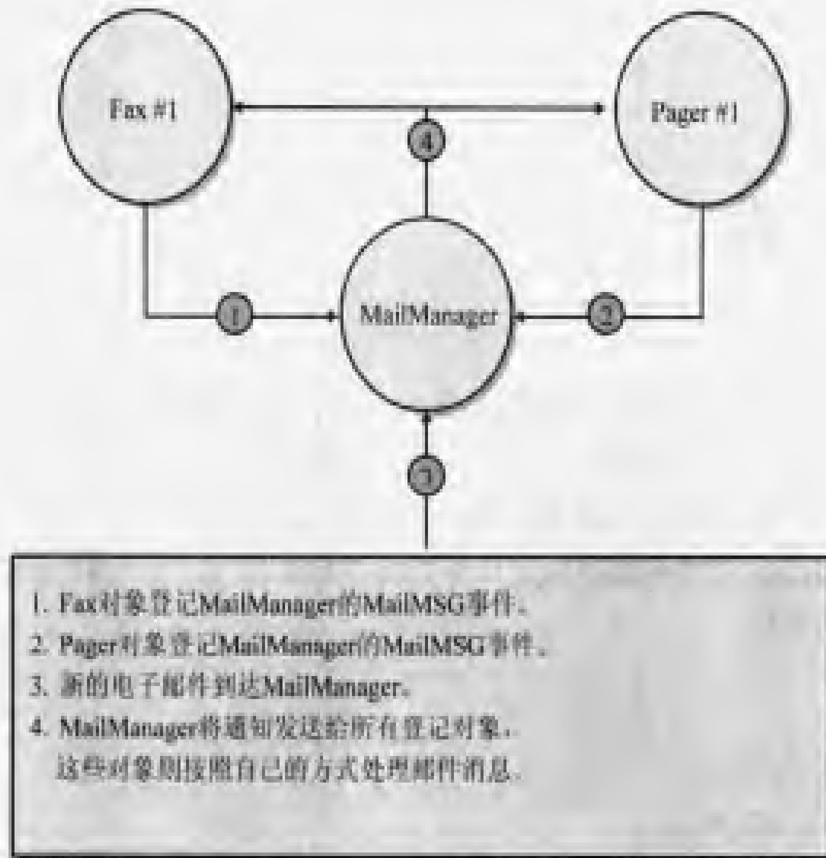


图 11.1 设计一个使用事件的应用程序

下面是图 11.1 中演示的应用程序所执行的工作。首先，应用程序通过创建一个 MailManager 的实例来进行初始化。其次，MailManager 提供了一个 MailMsg 事件。当 Fax 和 Pager 对象被构造时，它们将自己登记到 MailManager 的 MailMsg 事件上，这样在新的电子邮件消息到达时 MailManager 将知道要通知 Fax 和 Pager 对象。当 MailManager 收到一个新的电子邮件消息时(将来的某个时刻)，它会触发 MailMsg 事件，从而使所有的登记对象都有机会以它们期望的方式来处理新消息。

11.1 发布事件

让我们先来看一下 MailManager 的类型定义，这可以帮助我们理解微软推荐的定义事件成员时的设计模式：

```
class MailManager {
    // 在MailManager内部定义MailMsgEventArgs类型
    public class MailMsgEventArgs : EventArgs {

        // 1. 传递给事件接受者的类型定义信息
        public MailMsgEventArgs(
            String from, String to, String subject, String body) {

            this.from = from;
            this.to = to;
            this.subject = subject;
            this.body = body;
        }

        public readonly String from, to, subject, body;
    }

    // 2. 下面的委托类型定义了接受者必须实现的回调方法原型
    public delegate void MailMsgEventHandler(
        Object sender, MailMsgEventArgs args);

    // 3. 事件成员
    public event MailMsgEventHandler MailMsg;

    // 4. 下面的受保护虚方法负责通知事件的登记对象
    protected virtual void OnMailMsg(MailMsgEventArgs e) {

        // 有对象登记事件吗?
        if (MailMsg != null) {

            // 如果有，则通知委托链表上的所有对象
            MailMsg(this, e);
        }
    }

    // 5. 下面的方法将输入转化为期望的事件，该
    //    方法在新的电子邮件消息到达时被调用
    public void SimulateArrivingMsg(String from, String to,
        String subject, String body) {

        // 构造一个对象保存希望传递给通知接受者的信息
        MailMsgEventArgs e =
            new MailMsgEventArgs(from, to, subject, body);
```

```
            // 调用虚方法通知对象事件已发生，
            // 如果派生类型没有重写该虚方法，
            // 对象将通知所有登记的事件侦听者
            OnMailMsg(e);
        }
    }
```

设计 MailManager 类型的开发人员需要负责实现所有上述结构的相关工作。为此，开发人员必须定义以下几项：

1. 定义一个类型用于保存所有需要发送给事件通知接受者的附加信息

按照.NET 框架的约定，所有保存事件信息的类型都应该继承自 System.EventArgs，并且类型的名称应该以 EventArgs 结束。本例中，MailMsgEventArgs 类型定义的字段包括消息的发送者(from)，消息的接受者(to)，消息的主题(subject)，以及消息的正文(body)。

> **注意** EventArgs 类型在 FCL 中的定义如下：
>
> ```
> [Serializable]
> public class EventArgs {
>     public static readonly EventArgs Empty = new EventArgs();
>     public EventArgs() { }
> }
> ```

如我们所见，该类型的实现非常简单。它仅仅是作为一个让其它类型继承的基类型而出现的。许多事件都没有额外的信息需要传递。例如，当一个 Button 通知它的登记接受者自己被按下时，简单地调用回调方法就已经足够了。当我们定义一个不需要传递任何额外数据的事件时，可以直接使用 EventArgs.Empty，而不用再构造新的 EventArgs 对象。

2. 定义一个委托类型，用于指定事件触发时被调用的方法原型

按照.NET框架的约定，委托类型的名称应该以EventHandler结束。另外，回调方法的原型应该有一个void返回值，并且接受两个参数(尽管FCL中的某些事件处理器，如System.ResolveEvent Handler，违反了该约定)。第1个参数为Object类型，其指向发送通知的对象。第2个参数为一个继承自EventArgs的类型，其中包含所有通知接受者需要的附加信息。

如果我们定义的事件没有需要传递给事件接受者的附加信息，我们便不必定义新的委托类型。直接使用FCL中的System.EventHandler，并将EventArgs.Empty传递给第2个参数即可。EventHandler的原型如下：

```
public delegate void EventHandler(Object sender, EventArgs e);
```

3. 定义一个事件成员

本例中，事件的名称为MailMsg。该事件的类型为MailMsgEventHandler，其含义为所有事件通知的接受者都必须提供一个原型和MailMsgEventHandler相匹配的回调方法。

4. 定义一个受保护的虚方法，负责通知事件的登记对象

当一个新的电子邮件消息到达时，OnMailMsg方法会被调用。该方法接受一个经过初始化的MailMsgEventArgs对象(其中包含着事件的附加信息)。该方法应该首先检查是否有对象登记了事件，如果有，则触发事件。

继承自MailManager的类型可以根据需要重写OnMailMsg方法。这使得派生类型能够控制事件的触发。派生类型可以以任何它认为合适的方式来处理新的电子邮件消息。通常情况下，一个派生类型应该首先调用基类型的OnMailMsg方法，这样可以使所有已经登记事件的对象都能够收到通知。但是，派生类型也可以决定不让事件继续传递下去。

5. 定义一个方法，将输入转化为期望的事件

除了上述4步外，我们还必须定义一个方法来将外部的输入转换为触发事件的动作。本例中，调用SimulateArrivingMsg表示收到新的电子邮件消息。SimulateArrivingMsg接受一些相关的信息，然后构造一个MailMsgEventArgs对象，并通过调用虚方法OnMailMsg来通知MailManager对象收到了新的电子邮件消息。

通常情况下，这样的调用会导致事件被触发，从而使所有的登记对象都接受到通知。(如前所述，一个继承自 MailManager 的类型可以重写该方法。)

现在让我们更进一步来探究定义 MailMsg 事件时究竟发生了哪些事情。当编译器编译上面的源代码时，编译器会遇到下面定义事件的语句：

```
public event MailMsgEventHandler MailMsg;
```

C#编译器会将这段代码翻译成以下 3 个构造(译注：注意下面第 2 项中的 add_MailMsg 方法和第 3 项中的 remove_MailMsg 方法，这两个方法上的 virtual 关键字是 Jeffrey Richter 先生在本书 2003 年 5 月 30 日的英文版勘误中添加的。但是笔者使用 7.00.9466 版本的 C#编译器编译的结果显示它们是两个普通的实例方法，而非虚方法。另外，从事件的语义上来讲这两个方法也不应该为虚方法)：

```
// 1. 一个被初始化为 null 的“私有”委托类型字段
private MailMsgEventHandler MailMsg = null;

// 2. 一个允许对象登记事件的“公有”add_* 方法
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public virtual void add_MailMsg(MailMsgEventHandler handler) {
    MailMsg = (MailMsgEventHandler)
        Delegate.Combine(MailMsg, handler);
}

// 3. 一个允许对象注销事件的“公有”remove_* 方法
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public virtual void remove_MailMsg(MailMsgEventHandler handler) {
    MailMsg = (MailMsgEventHandler)
        Delegate.Remove(MailMsg, handler);
}
```

其中第一个构造是一个委托类型的字段。该字段引用的是一个委托链表的首部，链表中包含了那些期望在事件发生时被通知的委托对象。该字段被初始化为 null 意味着刚开始的时候没有对象登记事件。当有对象登记事件时，该字段会指向一个 MailMsgEventHandler 委托实例。每个 MailMsgEventHandler 委托实例内部都有一个指针指向另一个 MailMsgEventHandler 委托实例，或者一个 null(表示委托链表末尾)。当一个侦听者需要登记事件时，它只需将一个委托实例添加到该委托链表上就可以了。当然，注销事件意味着从委托链表上移除相应的委托实例。

大家可能已经注意到了即使我们的源代码中定义的是公有事件，委托字段 MailMsg 也总是为私有字段。将委托字段定义为私有方式可以防止类型外的代码错误地操作该字段。如果该字段为公有字段，那么任何代码都可以为其赋值，从而会删除所有已登记事件的委托实例！

C#编译器产生的第二个构造是一个允许其他对象登记事件的方法。C#编译器通过自动在事件名称(MailMsg)前添加 add_来命名该方法。编译器还会自动为该方法产生代码。产生的代码总是调用 System.Delegate 的静态 Combine 方法，将委托实例添加到委托链表上，然后返回新的链表首部。

C#编译器产生的第 3 个也是最后一个构造，是一个允许其他对象注销事件的方法。同样，C#编译器通过自动在事件名称(MailMsg)前添加 remove_来命名该方法。编译器自动为该方法产生的代码总是调用 System.Delegate 的静态 Remove 方法，将委托实例从委托链表上移除，然后返回新的链表首部。

大家应该还会注意到 add 和 remove 方法上都应用了一个 MethodImplAttribute 特性。具体而言，这些方法被标识为同步方法，这使得它们得以实现线程安全：也就是说多个侦听者可以同时登记或者注销事件，而不损坏委托链表。

本例中，add 和 remove 方法都是公有方法，这是因为源代码中声明的事件为公有事件。如果源代码中事件被声明为受保护事件，那么编译器产生的 add 和 remove 方法也将为受保护方法。事件的访问限制决定了登记事件和注销事件的代码的访问限制，但是只有类型本身可以直接访问委托字段。

除了上述 3 个构造外，编译器还会在托管模块的元数据中产生一个事件定义条目。该条目中包含了一些标记和定义事件所使用的委托类型，并且有一个对 add 和 remove 访问器方法的引用。这些信息仅仅为事件的抽象概念和它的访问器方法之间提供了一层关联关系。编译器和其他一些工具可以使用这些元数据信息。当然，我们也可以通过 System.Reflection.EventInfo 类来获取它们。但 CLR 本身并不使用它们，它在运行时仅需要访问器方法即可。

11.2 侦听事件

到这里，剩下的就是一些较为简单的事情了。本节向大家展示怎样定义一个类型来使用另一个类型提供的事件。让我们从 Fax 类型的代码开始：

```
class Fax {
   // 将 MailManager 对象传递给构造器
   public Fax(MailManager mm) {

      // 构造一个指向 FaxMsg 回调方法的 MailMsgEventHandler
      // 委托实例。然后登记 MailManager 的 MailMsg 事件
      mm.MailMsg += new MailManager.MailMsgEventHandler(FaxMsg);
   }

   // MailManager 将调用该方法来通知
   // Fax 对象收到一个新的电子邮件 消息
   private void FaxMsg(
      Object sender, MailManager.MailMsgEventArgs e) {

      // 参数 'sender' 表示 MailManager 对象，如果
      // 期望和事件的触发者通信，将会用到该参数

      // 参数 'e' 表示 MailManager 对象希望提供的一些
      // 附加事件信息

      // 通常情况下，这里的代码应该传真电子邮件消息，
      // 这里的实现仅仅将消息显示到控制台上
      Console.WriteLine("Faxing mail message:");
      Console.WriteLine(
         " From: {0}\n To: {1}\n Subject: {2}\n Body: {3}\n",
         e.from, e.to, e.subject, e.body);
   }
   public void Unregister(MailManager mm) {

      // 构造一个指向 FaxMsg 回调方法的 MailMsgEventHandler
      // 委托实例
      MailManager.MailMsgEventHandler callback =
         new MailManager.MailMsgEventHandler(FaxMsg);
         // 注销 MailManager 的 MailMsg 事件
         mm.MailMsg -= callback;
   }
}
```

当电子邮件应用程序初始化时，它会首先构造一个MailManager对象，并将其引用保存在一个变量中。然后应用程序以该MaiManager对象引用作为参数构造一个Fax对象。在Fax构造器中，又构造一个新的MailManager.MailMsgEventHandler委托对象。这个新的委托对象是对Fax类型的FaxMsg方法的一个封装。我们应该注意到FaxMsg方法的返回值为void，并且接受的参数和MailManager.MailMsgEventHandler委托定义的相同——只有这样代码才能通过编译。

在构造了委托之后，Fax对象使用C#的 += 操作符来登记MailManager的MailMsg事件：

```
mm.MailMsg += new MailManager.MailMsgEventHandler(FaxMsg);
```

因为C#编译器对事件提供了内置的支持，所以它可以将上面一行转换为下面的代码：

```
mm.add_MailMsg(new MailManager.MailMsgEventHandler(FaxMsg));
```

如果我们使用的编程语言不直接支持事件，我们仍可以通过显式调用add访问器方法来登记事件。两者的效果是等同的—— 最后都是add访问器方法将委托实例添加到事件的委托链表上，只是后者的源代码看起来不够漂亮而已。

当MailManager触发事件时，Fax对象的FaxMsg方法将被调用。MailManager对象会被作为参数传递给FaxMsg方法。大多数情况下，该参数可以忽略。但是如果Fax对象希望在响应事件的时候能够访问MailManager对象中的字段或方法，该参数还是有用的。第2个参数是一个MailMsgEventArgs对象引用。该对象包含了MailManager认为对事件接受者有用的信息。

从MailMsgEventArgs对象中，FaxMsg方法可以很容易访问电子邮件消息的发件人、收件人、主题和正文。在一个实际的Fax对象中，该信息将被传真给某个地方。本例中，我们只是将它们简单地显示在控制台窗口中。

当一个对象不再希望接受事件通知时，它应该注销该事件。例如，如果用户不再希望将电子邮件传给一个传真机时，Fax对象就应该注销MailMsg 事件。注意，只要一个对象仍然登记有另一个对象的事件，该对象就不可能被执行垃圾收集。如果我们的类型实现了IDisposable接口的Dispose方法，我们应该在其内部注销其登记的所有事件。(关于IDisposable接口的更多信息可参见本书第19章。)

Fax 的 Unregister 方法演示了怎样注销一个事件。该方法和 Fax 构造器中的代码非常类似，二者惟一的差别在于前者用 -= 操作符替代了 += 操作符。当 C#编译器看到使用 -= 操作符注销事件的语句时，它会产生对事件的 remove 方法的一个调用。

```
mm.remove_MailMsg(callback);
```

和 += 操作符一样，即使我们使用的编程语言不直接支持事件，我们仍可以通过显式调用 remove 访问器方法来注销事件。remove 方法会扫描事件的委托链表，寻找传入的委托实例。如果找到匹配的委托实例，就将其从事件的委托链表上移除。如果没有找到，也不会出现任何错误，事件的委托链表也不会有任何改变。

顺便说一下，C# 要求我们的代码使用 += 操作符和 -= 操作符来在委托链表上添加和移除委托实例。如果我们试图显式调用 add 和 remove 方法，C#编译器将会报告“不能显式调用操作符或者访问器”错误。

MailManager 应用程序示例(可以从 http://www.wintellect.com 上下载)中包含了 MailManager 类型、Fax 类型和 Pager 类型的所有源代码。大家将会看到 Pager 类型和 Fax 类型的实现非常相似。

11.3 显式控制事件注册

有时我们会感到编译器自动产生的 add 和 remove 方法不够理想。例如，如果我们需要频繁地添加或者移除委托实例，同时我们又知道我们的应用程序是在单线程环境下运行，这时再对包含委托实例的对象进行同步访问的话就会损伤应用程序的性能。

另外，如果我们的类型定义了许多事件，那么我们也会对编译器自动产生的 add 和 remove 方法感到不够满意。11.4 一节将会集中讨论这个问题。

幸运的是，C#和许多其他的编译器都允许我们显式实现 add 和 remove 访问器方法。下面的示例对前面的 MailManager 类型进行了修改，为 add 和 remove 访问器方法提供了显式实现。

```
class MailManager {
    // 1. 传递给事件接受者的类型定义信息
    public class MailMsgEventArgs : EventArgs { ... }

    // 2. 下面的委托类型定义了接受者必须实现的回调方法原型
    public delegate void MailMsgEventHandler(
        Object sender, MailMsgEventArgs args);

    // 3a. 显式定义一个私有委托链表字段
    private MailMsgEventHandler mailMsgEventHandlerDelegate;
    // 3b. 显式定义事件及其访问器方法
    public event MailMsgEventHandler MailMsg {

        // 将传入的事件处理器(value)添加到委托链表上
        add {
            mailMsgEventHandlerDelegate = (MailMsgEventHandler)
            Delegate.Combine(mailMsgEventHandlerDelegate, value);
        }

        // 将传入的事件处理器(value)从委托链表上移除
        remove {
            mailMsgEventHandlerDelegate = (MailMsgEventHandler)
            Delegate.Remove(mailMsgEventHandlerDelegate, value);
        }
    }

    // 4. 下面的受保护虚方法负责通知事件的登记对象
    protected virtual void OnMailMsg(MailMsgEventArgs e) {

        // 有对象登记事件吗?
        if (mailMsgEventHandlerDelegate != null) {

            // 如果有，则通知委托链表上的所有对象
            mailMsgEventHandlerDelegate(this, e);
        }
    }

    // 5. 下面的方法将输入转化为期望的事件，该
    //    方法在新的电子邮件消息到达时被调用
    public void SimulateArrivingMsg(String from, String to,
        String subject, String body) { ... }
}
```

在新版的 MailManager 中，第 1、2、5 步和前面示例中的完全相同，没有做任何改变。第 3 步被分成了两步(3a 和 3b)，第 4 步有一些小的改动。下面我们来详细分析这些改动。

3a. 显式定义一个私有委托链表字段

在先前的示例中，C#编译器会自动为我们定义一个私有字段。在新版的示例中，我们使用了另外一种定义事件的语法，我们不仅要显式提供 add 和 remove 访问器方法的实现，我们还要显式提供一个保存委托链表的字段。

3b. 显式定义事件及其访问器方法

第 3a 步中只是定义了一个字段(一个 MailMsgEventHandler 委托实例)。该字段还不能成为一个事件成员。新的事件语法实际上就是代码中定义事件成员的内容，其中的 add 和 remove 块为事件的访问器方法提供了实现。注意，每个访问器方法都接受一个类型为 MailMsgEventHandler 的隐含参数(value)。这些方法在内部或者向委托链表上添加一个 MailMsgEventHandler 委托实例，或者从委托链表上移除一个 MailMsgEventHandler 委托实例。我们知道属性可以有一个 get 访问器方法，也可以有一个 set 访问器方法，还可以同时拥有 get 和 set 两个访问器方法。但是事件必须同时具有 add 访问器方法和 remove 访问器方法。

上面的示例中除了没有应用 MethodImplAttribute 特性外，其他地方和 C#编译器自动产生的代码的行为非常类似。实际上，我们仍然可以在源代码中使用 += 操作符和 -= 操作符，编译器遇到它们时会去自动调用我们显式实现的方法。MethodImplAttribute 特性的缺失使得上述示例实现的访问器方法不再是线程安全的方法，但是这也意味着代码的性能将有所提升。当然，只有在我们确信每次只有一个线程访问事件的委托链表时，我们才应该移除 MethodImplAttribute 特性。

4. 定义一个受保护的虚方法，负责通知事件的登记对象

从语义上来讲，这里的 OnMailMsg 方法和前一个版本的 OnMailMsg 方法是相同的。二者唯一的差别是我们将事件成员的名称替换成了委托字段的名称。

11.4 在一个类型中定义多个事件

在上一节中，我们讨论了如何为一个事件显式提供 add 和 remove 访问器方法。当我们显式提供这些访问器方法时，我们实际上可以在具体实现上进行一些创造。下面我们来看看如何使用显式提供的访问器方法来减少应用程序使用的内存资源。

如果我们查看 FCL 文档，我们会发现 System.Windows.Forms.Control 类型中定义了大约 60 个事件。如果 Control 类型在实现这些事件时让编译器来自动产生委托字段以及 add 和 remove 访问器方法，那么每个 Control 类型将仅仅因为事件就有将近 60 个委托字段！

由于我们大多数时候在对象上登记的事件都很少，因此每创建一个 Control 类型(以及继承自 Control 的类型)的实例都会浪费很多内存。顺便提一句，System.Web.UI.Control 类型也存在这样的问题，其内部也使用了下面介绍的技巧。

通过创造性地提供显式实现的 add 和 remove 访问器方法，我们可以极大地减少一个对象由于定义多个事件而造成的内存浪费。本节的目的就是向大家展示这一技巧。

我们的基本思想是让每个对象都保存一个事件/委托对的集合(通常为一个散列表)。当一个新的对象被构造时，该集合是空的。当有事件被登记时，我们将在集合中查找该事件。如果集合中存在该事件，那么新的委托实例将被组合到表示该事件的委托链表上；否则，该事件和新的委托实例将直接被添加到集合中。

当事件需要触发时，我们还是首先在集合中查找该事件。如果集合中不存在该事件，也就是说该事件没有被登记，那么将没有任何委托实例被调用。如果集合中存在该事件，那么其对应的委托链表将被调用。

下面的代码演示了在一个类型中定义多个事件时推荐的设计模式：

```
class TypeWithLotsOfEvents {
   // 1. 定义一个受保护的实例字段，该字段引用一个集合来管
   // 理一组事件/委托对。注意：类型 EventHandlerSet 并非位
   // 于 FCL 中，本章稍后将予以描述
   protected EventHandlerSet eventSet =
      EventHandlerSet.Synchronized(new EventHandlerSet());

   // 2. 为 Foo 事件定义必要的成员。
   // 2a. 构造一个静态只读对象来标识该事件。每个对象都有一
   // 个散列码用于在集合中查找事件对应的委托链表
   protected static readonly Object fooEventKey = new Object();

   // 2b. 为事件定义一个继承自 EventArgs 的类型
   public class FooEventArgs : EventArgs {}

   // 2c.为事件定义委托原型
   public delegate void FooEventHandler(Object sender, FooEventArgs e);
```

```
    // 2d. 为事件定义访问器方法用于在集合上添加/移除委托实例
    public event FooEventHandler Foo {
        add { eventSet.AddHandler(fooEventKey, value); }
        remove { eventSet.RemoveHandler(fooEventKey, value); }
    }

    // 2e、为事件定义一个受保护的虚方法(OnXXX)
    protected virtual void OnFoo(FooEventArgs e) {
        eventSet.Fire(fooEventKey, this, e);
    }

    // 2f、定义一个方法将输入转化为期望的事件
    public void SimulateFoo() {
        OnFoo(new FooEventArgs());
    }

    // 3. 为 Bar 事件定义必要的成员。
    // 3a. 构造一个静态只读对象来标识该事件。每个对象都有一
    // 个散列码用于在集合中查找事件对应的委托链表
    protected static readonly Object barEventKey = new Object();

    // 3b. 为事件定义一个继承自 EventArgs 的类型
    public class BarEventArgs : EventArgs {}

    // 3c. 为事件定义委托原型
    public delegate void BarEventHandler(Object sender, BarEventArgs e);

    // 3d. 为事件定义访问器方法用于在集合上添加/移除委托实例
    public event BarEventHandler Bar {
        add { eventSet.AddHandler(barEventKey, value); }
        remove { eventSet.RemoveHandler(barEventKey, value); }
    }

    // 3e. 为事件定义一个受保护的虚方法(OnXXX)
    protected virtual void OnBar(BarEventArgs e) {
        eventSet.Fire(barEventKey, this, e);
    }

    // 3f. 定义一个方法将输入转化为期望的事件
    public void SimulateBar() {
        OnBar(new BarEventArgs());
    }
}
```

实现上述代码展示的设计模式是定义事件的开发人员的职责。使用事件的开发人员则无需知道其中的内部实现。下面，我们来仔细分析上述代码中的每一个步骤。

1. 定义一个受保护的实例集合字段

TypeWithLotsOfEvents 类型的每个实例都会使用该集合字段来保存其上的事件侦听者集合。这里的集合通常使用散列表来实现，因为这可以提供比较快的事件查找速度。集合中的每个元素都有一个键(类型通常为 System.Object)用以唯一地标识事件类型，以及一个和键对应的值用以标识事件触发时要调用的委托链表。将该字段的访问限制设为 protected 的目的是让派生类型也可以使用该集合来为自己定义的事件服务。在本例中，我们使用了一个 EventHandlerSet 集合。该集合是我自己定义的一个类型，它在内部使用了一个 System.Collections.Hashtable 对象。EventHandlerSet 类型还有一些成员负责处理事件和委托实例。本章稍后会介绍 EventHandlerSet 类型。

2. 为类型希望提供的事件定义必要的成员

对于我们希望类型提供的每个事件，我们必须为其定义一组成员。第 2a 到第 2f 步中描述了这些成员。大家可能已经注意到了我们没有使用实例字段——它们正是浪费内存的根源。

2a. 构造一个静态只读对象来标识事件

我们必须采取一种方式来唯一地标识集合中的每个事件。因为我们使用的是一个散列表，所以我们需要一个唯一的键，这可以由一个对象的散列码产生。TypeWithLotsOfEvents 类型的所有实例将共享这个惟一的键，所以我们构造了一个 Object，并将其引用保存在一个静态只读字段中。只有在 TypeWithLotsOfEvents 类型的用户构造多个 TypeWithLotsOfEvents 类型实例时，这种技巧才会节省内存。如果用户只构造一个 TypeWithLotsOfEvents 类型实例，这种技巧实际上会耗费更多的内存。如果是这种情况，就不应该再使用这里演示的技巧。另外，本例中将这个静态字段定义为受保护字段，目的是为了让派生类型也能使用它。当然，大家也可以将其定义为私有字段。

2b. 为事件定义一个继承自 EventArgs 的类型，该类型用于保存任何需要传递给事件接受者的附加信息

本章前面曾经讨论过 EventArgs 类型和它的用法。如果事件没有什么特殊的信息需要传递，我们可以简单地使用 FCL 中的 System.EventArgs 类型就可以了。本例中定义该类型仅仅是处于演示目的，因此也就没有在其中定义任何字段。

2c．定义一个委托类型，指定事件触发时被调用的方法原型

本章前面曾经讨论过这样的委托类型和它的用法。本例中，由于 FooEventArgs 类型没有定义任何字段，所以实际上我们不需要定义一个特殊的委托类型。如果事件没有特殊的信息需要传递给侦听者，我们可以直接使用 FCL 中定义的 System.EventHandler 类型。

2d．显式定义事件及其访问器方法

这是整个技巧中比较有趣的部分。在该步骤中，我们为类型希望提供的每一个事件显式定义了 add 和 remove 访问器方法。add 和 remove 访问器方法分别调用了 EventHandlerSet 类型的 AddHandler 和 RemoveHandler 方法。EventHandlerSet 类型的 AddHandler 方法会扫描集合看其中是否存在 fooEventKey 键对象。如果不存在，它将被添加到散列表中，事件被触发时调用的也将是该键对象所标识的委托实例。如果存在，传入的值(新的委托实例)将被组合到已经存在的委托链表上。RemoveHandler 方法所做的操作正好相反。(再提醒大家一下，本章稍后会详细解释 EventHandlerSet 类型。)在本例中，所有对 EventHandlerSet 集合的访问都是线程安全的，所以我们不必再在事件的 add 和 remove 访问器方法上指定 MethodImplAttribute 特性。

2e．定义一个受保护的虚方法，负责通知事件的登记对象

不管何时出现“Foo 事件”，OnFoo 方法都会被调用。在本例中，OnFoo 方法调用了 EventHandlerSet 类型的 Fire 方法。Fire 方法会在集合中查找传入的键对象。如果找到，它将调用与其相关联的委托链表，其中传递给委托链表的参数为 this(触发事件的对象)和表示附加信息的继承自 EventArgs 的类型实例。大家稍后就会看到 EventHandlerSet 类型的 Fire 方法的详细实现。

2f．定义一个方法将输入转化为期望的事件

本章前面曾经讨论过该方法和它的用法。本例中，我们调用了 SimulateFoo 方法来模拟出现了一个“Foo 事件”。该方法的所有工作就是传递必要的附加信息，然后触发事件。

3. 为类型希望提供的事件定义必要的成员

这里仅仅是对第 2a 到第 2f 步的一个重复。本例中，我们定义了 Foo 和 Bar 两个事件。第 3a 到 3f 步为 Bar 事件提供了必要的成员。

11.5 设计 EventHandlerSet 类型

在上一节的演示代码中我们使用了一个 EventHandlerSet 类型。如前所述，该类型并不是一个 FCL 类型，它是我自己定义的一个类型。下面是该类型的实现代码：

```
public class EventHandlerSet : IDisposable {

    // 用于保存“事件键/委托值”对的私有散列表
    private Hashtable events = new Hashtable();

    // 一个索引器，用于获取或设置与传入的事件对象的
    // 散列键相关联的委托
    public virtual Delegate this[Object eventKey] {
        // 如果对象不在集合中，则返回 null
        get {
            return (Delegate) events[eventKey];
        }
        set {
            events[eventKey] = value;
        }
    }

    // 在指定的事件对象的散列键对应的委托链表上添加/组合一个委托实例
    public virtual void AddHandler(Object eventKey, Delegate handler) {
        events[eventKey] =
            Delegate.Combine((Delegate) events[eventKey], handler);
    }

    // 在指定的事件对象的散列键对应的委托链表上移除一个委托实例
    public virtual void RemoveHandler(
        Object eventKey, Delegate handler) {
```

```
        events[eventKey] =
            Delegate.Remove((Delegate) events[eventKey], handler);
    }

    // 在指定的事件对象的散列键对应的委托链表上触发事件
    public virtual void Fire(Object eventKey,
        Object sender, EventArgs e) {

        Delegate d = (Delegate) events[eventKey];
        if (d != null)
            d.DynamicInvoke(new Object[] { sender, e });
    }

    // 释放对象以使散列表占用的内存资源在下一次垃圾
    // 收集中被回收，从而阻止垃圾收集器提升其代龄
    public void Dispose() {
        events = null;
    }

    // 下面的静态方法返回一个对传入的EventHandlerSet
    // 对象的线程安全的封装
    public static EventHandlerSet Synchronized(
        EventHandlerSet eventHandlerSet) {

        if (eventHandlerSet == null)
            throw new ArgumentNullException("eventHandlerSet");

        return new SynchronizedEventHandlerSet(eventHandlerSet);
    }

    // 下面的类在EventHandlerSet类的基础上提供了
    // 一个线程安全的封装
    private class SynchronizedEventHandlerSet : EventHandlerSet {

        // 引用非线程安全的对象
        private EventHandlerSet eventHandlerSet;

        // 在非线程安全的对象上构造一个线程安全的封装
        public SynchronizedEventHandlerSet(
            EventHandlerSet eventHandlerSet) {
            this.eventHandlerSet = eventHandlerSet;
            Dispose();          // 释放基类中的散列表对象
        }
```

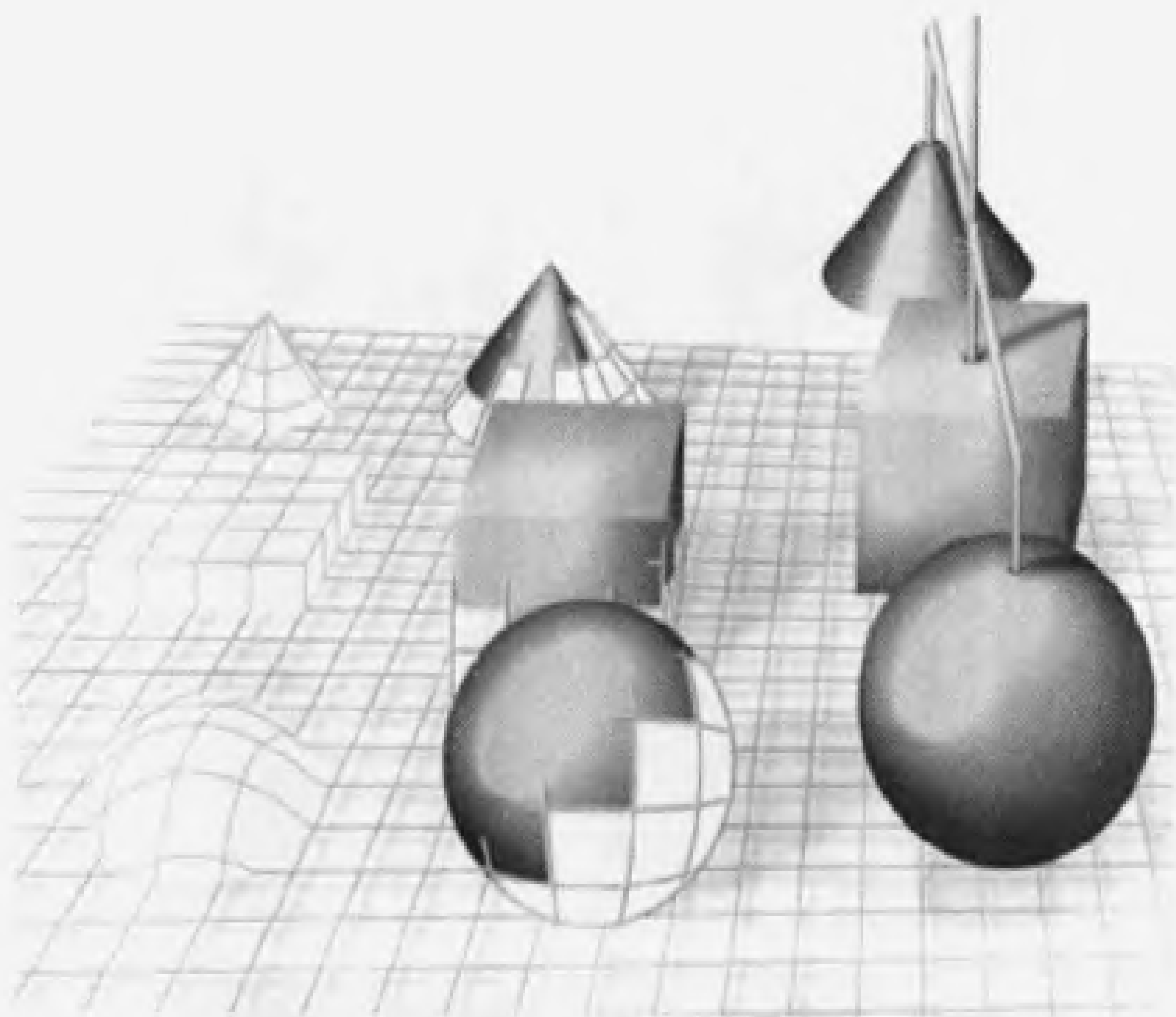
```
        // 线程安全的索引器
        public override Delegate this[Object eventKey] {
            [MethodImpl(MethodImplOptions.Synchronized)]
            get {
                return eventHandlerSet[eventKey];
            }

            [MethodImpl(MethodImplOptions.Synchronized)]
            set {
                eventHandlerSet[eventKey] = value;
            }
        }

        // 线程安全的 AddHandler 方法
        [MethodImpl(MethodImplOptions.Synchronized)]
        public override void AddHandler(
            Object eventKey, Delegate handler) {
            eventHandlerSet.AddHandler(eventKey, handler);
        }

        // 线程安全的 RemoveHandler 方法
        [MethodImpl(MethodImplOptions.Synchronized)]
        public override void RemoveHandler(
            Object eventKey, Delegate handler) {
            eventHandlerSet.RemoveHandler(eventKey, handler);
        }

        // 线程安全的 Fire 方法
        [MethodImpl(MethodImplOptions.Synchronized)]
        public override void Fire(
            Object eventKey, Object sender, EventArgs e) {
            eventHandlerSet.Fire(eventKey, sender, e);
        }
    }
}
```

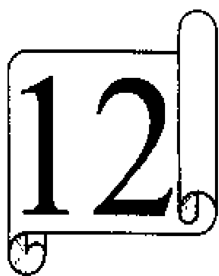
EventHandlerSet 类型的实现相当简单，其绝大多数代码都是在一个散列表上进行的操作。值得一提的是其中的 Fire 方法。该方法首先从 events 散列表中找到一个委托链表，然后调用该委托链表上封装的所有回调方法。因为散列表中可能包含有各种不同的委托类型，所以我们不可能在编译时进行类型安全的委托调用。因此，我们使用了 System.Delegate 的 DynamicInvoke 方法，并将回调方法的参数组合为一个对象数组传递给它。DynamicInvoke 方法在内部会检测回调方法期望的参数和我们传递的参数是否匹配。如果匹配，回调方法将被调用；否则，DynamicInvoke 方法将抛出一个异常。

注意 FCL中定义有一个名为System.ComponentModel.EventHandlerList的类型，该类型和上面展示的EventHandlerSet类型所做的事情基本上是一样的。System.Windows.Forms.Control和System.Web.UI.Control类型在维护它们各自的稀疏事件集合时，使用的就是EventHandlerList这个类型。大家当然可以选择使用FCL中的EventHandlerList类型。该类型和我们上面展示的EventHandlerSet类型的不同之处在于EventHandlerList类型在内部使用的是一个链表，而不是散列表。这意味着访问EventHandlerList类型中的元素要比访问EventHandlerSet类型中的元素慢一些。另外，EventHandlerList类型没有提供任何线程安全的访问方式。因此在某些情况下，大家还需要自己实现一个对EventHandlerList类型的线程安全的封装。

第IV部分

基本类型





文本处理

本章讨论.NET 框架中的字符与字符串处理机制。首先向大家介绍的是 System.Char 结构以及各种处理单个字符的操作，接着我们会深入讨论更为有用的 System.String 类—— 我们通常使用它来处理恒定字符串(恒定字符串一经创建便不能再做任何改变)。然后，我们将学习如何使用 System.Text.StringBuilder 类来执行一些高效的字符串动态创建操作。在解决了字符串的这些基本问题之后，本章最后将向大家介绍如何将对象格式化为字符串、以及如何使用各种编码来对字符串进行高效的持久化(persist)或传输操作。

12.1 字　　符

在.NET 框架中，字符采用 16 位 Unicode 编码，这种编码使得国际化应用程序的开发大为简化。从类型上来看，一个字符由一个 System.Char 结构(值类型)实例表示。System.Char 类型相当简单。它提供有两个常数字段：MinValue(定义为 0x0000)和 MaxValue(定义为 0xFFFF)。

给定一个 Char 实例作为参数，我们可以调用 Char 的静态方法 GetUnicodeCategory 返回一个 System.Globalization.UnicodeCategory 枚举值。根据该枚举值，我们可以判断传入的字符是否为一个控制字符、货币符号、小写字符、大写字符、标点符号、数学符号等(定义于 Unicode 3.0 标准)。

为了简化编程，Char 类型还提供有其他几个静态方法，例如 IsDigit、IsLetter、IsWhiteSpace、IsUpper、IsLower、IsPunctuation、IsLetterOrDigit、IsControl、IsNumber、IsSeparator、IsSurrogate 以及 IsSymbol。所有这些方法都在内部调用了 GetUnicodeCategory 方法，并简单地返回 true 或者 false。注意这些方法接受的参数或者为一个字符，或者为一个 String 和该 String 中的一个字符索引。

另外，我们也可以调用静态方法 ToLower 或 ToUpper 将一个字符转换为其对应的小写形式或者大写形式。这些方法在做字符转换时会用到与调用线程相关联的语言文化信息(方法内部会查询 System.Threading.Thread 的静态属性 CurrentCulture)。或者我们也可以传递一个 System.Globalization.CultureInfo 实例作为参数来指定一个特定的语言文化信息。ToLower 和 ToUpper 需要语言文化信息是因为字符转换本来就是一项赖于特定语言文化的操作。例如，土耳其语认为 U+0069(小写拉丁字母 I)的大写形式为 U+0130(大写拉丁字母 I 上面再加一点)，而其他语言文化则认为是 U+0049(大写拉丁字母 I)

除了这些静态方法外，Char 类型还提供了几种实例方法。如果两个 Char 实例表示相同的 16 位 Unicode 码值，Equals 方法将返回 true。CompareTo 方法(定义于 IComparable 接口中)返回两个字符码值的比较结果，这种比较和语言文化无关。本书第 15 章会对 IComparable 接口和 CompareTo 方法做详细解释。ToString 方法返回包含单个字符的字符串。和 ToString 方法相反，Parse 方法接受包含一个字符的 String 作为参数，然后返回其中的字符。

Char 类型的最后一个方法 GetNumericValue，返回一个字符对应的数值形式，看下面的代码：

```
using System;

class App {
    static void Main() {
        Double d;

        // '\u0033' 即为数字 '3'
        d = Char.GetNumericValue('\u0033');          // 用 '3' 也可以
        Console.WriteLine(d.ToString());              // 显示 "3"

        // '\u00bc' 即为普通分数('1/4')
        d = Char.GetNumericValue('\u00bc');
        Console.WriteLine(d.ToString());              // 显示 "0.25"
```

```
        ·
        // 'A' 即为大写拉丁字母 'A'
        d = Char.GetNumericValue('A');
        Console.WriteLine(d.ToString());          // 显示 "-1"
    }
}
```

有三种技巧允许我们在数值和 Char 实例之间进行转换。下面按推荐使用的优先顺序列出了它们:

- **转型** 将一个 Char 转换成一个数值(如 Int32)最容易的方法便是转型。在三种技巧中，这种技巧的效率最高，因为编译器会直接产生 IL 指令来执行转换，而不会有任何的方法调用。另外，某些语言(例如 C#)允许我们告诉编译器是否使用 checked 或 unchecked 代码(本书第 5 章对此有详细讨论)来执行转换。这使得我们可以自行决定当转换导致了数据丢失时是否抛出 System.OverflowException 异常。这种技巧惟一的缺点是它要求我们的编译器要将期望转换的数值类型看作是基元类型。例如 Visual Basic 就不认为 UInt16 是一个基元类型，所以在 Visual Basic 中就不能使用这种技巧在 Char 和 UInt16 之间进行转换。

- **使用 Convert 类型** System.Convert 类型提供了几种静态方法允许我们在 Char 和数值类型之间执行转换。所有这些方法执行的转换都为 checked 操作，即如果转换过程出现数据丢失，将会抛出 OverflowException 异常。

- **使用 IConvertible 接口** Char 类型和所有.NET 框架类库(FCL)中的数值类型都实现了 IConvertible 接口。该接口定义了如 ToUInt16、ToChar 这样的方法。这些方法执行起来的效率不如上面的好，因为在一个值类型上调用接口方法会导致该值类型实例被装箱——而 Char 和所有的数值类型都是值类型。如果类型之间不能进行转换(例如将一个 Char 转换为一个 Boolean)、或者转换会导致数据丢失，IConvertible 接口中的方法都会抛出异常。注意许多类型(包括 FCL 中的 Char 类型和数值类型)都是以显式接口成员实现的方式来实现 IConvertible 接口中的方法的(第 15 章对此有描述)。这意味着我们在调用任何 IConvertible 接口方法前，都必须首先将实例转型为 IConvertible 接口。

下面的代码演示了上述3种技巧：

```
using System;

class App {
    static void Main() {
        Char c;
        Int32 n;

        // 使用C#的转型在数值和字符之间执行转换
        c = (Char) 65;
        Console.WriteLine(c);                   // 显示 "A"

        n = (Int32) c;
        Console.WriteLine(n);                   // 显示 "65"

        c = unchecked((Char) (65536 + 65));
        Console.WriteLine(c);                   // 显示 "A"

        // 使用 Convert 中的方法在数值和字符之间执行转换
        c = Convert.ToChar(65);
        Console.WriteLine(c);                   // 显示 "A"

        n = Convert.ToInt32(c);
        Console.WriteLine(n);                   // 显示 "65"

        // 这里演示了 Convert 类的范围检查
        try {
            c = Convert.ToChar(70000);          // 70000 超出了 16 位
            Console.WriteLine(c);               // 这里由于上面抛出异常不会执行
        }
        catch (OverflowException) {
            Console.WriteLine("Can't convert 70000 to a Char.");
        }

        // 使用 IConvertible 接口在数值和字符之间执行转换
        c = ((IConvertible) 65).ToChar(null);
        Console.WriteLine(c);                   // 显示 "A"

        n = ((IConvertible) c).ToInt32(null);
        Console.WriteLine(n);                   // 显示 "65"
    }
}
```

12.2 System.String 类型

任何应用程序中使用最频繁的类型无疑是 System.String 类型。一个 String 表示一个恒定不变的字符序列集合。String 类型直接继承自 Object，这使得它成为一个引用类型，也就是说线程的堆栈上不会驻留有任何字符串。(译注：注意这里的“直接继承”。直接继承自 Object 的类型一定是引用类型，因为所有的值类型都继承自 System.ValueType。值得指出的是 System.ValueType 却是一个引用类型。)另外，String 类型还实现了几个接口(IComparable、ICloneable、IConvertible 和 IEnumerable)。

12.2.1 创建字符串

许多编程语言(包括 C#)都将 String 认为是一个基元类型——也就是说编译器允许我们在源代码中用文本常量(literal)来直接表达字符串。编译器会将这些文本常量字符串存放在托管模块的元数据中，然后在运行时使用一种称作**字符串驻留**(string interning)的机制来访问它们(本章稍后探讨)。

在 C#中，我们不能使用 new 操作符来创建 String 对象：

```
class App {
   static void Main() {
      String s = new String("Hi there.");     // 编译错误
   }
}
```

相反，我们必须使用下面特殊的简化语法：

```
class App {
   static void Main() {
      String s = "Hi there.";
   }
}
```

如果我们编译上面这段代码，并用 ILDasm.exe 查看生成的 IL 代码，我们将会看到以下内容：

```
.method private hidebysig static void Main() cil managed
{
   .entrypoint
   // Code size  7 (0x7)
   .maxstack 1
   .locals (string V_0)
   IL_0000:  ldstr   "Hi there."
   IL_0005:  stloc.0
   IL_0006:  ret
} // end of method App::Main
```

我们知道，构造一个对象实例的 IL 指令为 newobj。但是在上面的例子中我们并没有看到 newobj 指令。相反，我们看到了一个特殊的 IL 指令 ldstr(即 load string，加载字符串)，该指令通过从元数据中获得的文本常量来构造 String 对象。这表明 CLR 实际上有一个更为特殊、高效的构造字符串对象的方式。

在极少数的情况下，我们可能需要一个不是从元数据中的文本常量构造而得的 String 对象。这时，我们需要使用 C#的 new 操作符来调用 String 类型提供的一些构造器。其中，接受 Char*或者 SByte*参数的构造器主要供托管扩展 C++编写的代码调用。这些构造器会根据 Char 数组或者 SByte 数组(译注：这里的 Char 数组、SByte 数组是从 C/C++的观点来讲的，就是指上面的 Char*或 SByte*，它们和 C#中的托管数组不同)来创建、并初始化 String 对象。String 类型还提供了其他一些构造器，它们没有任何的指针参数，因此可以在任何一门托管编程语言中调用。

C#提供了专门的语法来帮助我们在源代码中输入文本常量字符串。对于一些特殊的字符(例如换行、回车、退格等)，C#允许我们使用 C/C++开发人员所熟悉的转义机制：

```
// 包含回车、换行的字符串
String s = "Hi\r\nthere.";
```

重要 在上面的例子中，我们是将回车、换行符硬编码到字符串去中的，但我个人并不推荐这种做法。相反，我们应该使用 System.Enviroment 类型提供的一个名为 NewLine 的只读属性，当我们的应用程序在 Windows 系统上运行时，该属性返回的字符串就包括上述两个字符。但是，NewLine 属性是依赖于特定平台的，根据底层平台的不同，它的返回值也有所不同。例如，如果我们将 CLR 移植到 UNIX 系统上，NewLine 属性将返回一个仅包含字符 '\n' 的字符串。所以用下面的方式来定义上面的字符串将使得它可以在任何平台上都能正确运行：

```
String s = "Hi" + Environment.NewLine + "there.";
```

我们可以用 C#中的+操作符来将几个字符串连接为一个字符串：

```
// 将三个文本常量字符串连接为一个字符串
String s = "Hi" + " " + "there.";
```

在上面的代码中，由于所有的字符串都是文本常量字符串，所以编译器会在编译时就将它们连接为一个字符串，并只将连接后的字符串("Hi there.")放在生成模块的元数据中。对那些不是文本常量的字符串使用+操作符将使连接操作在运行时执行。要在运行时连接几个字符串，建议大家不要使用+操作符，因为它会在要执行垃圾收集的托管堆上创建多个字符串对象。相反，我们应该使用 System.Text.StringBuilder 类型(本章稍后予以解释)。

最后，C#还为我们提供了一种声明字符串的特殊方式，这种特殊方式允许编译器将所有在引号之间的字符都认为是字符串的一部分。由这种特殊的方式所声明的字符串被称为字面字符串(verbatim string)，它典型地用于文件或目录的路径、以及正则表达式。看下面的例子：

```
// 指定应用程序的路径名
String file = "C:\\Windows\\System32\\Notepad.exe";

// 使用字面字符串指定应用程序的路径名
String file = @"C:\Windows\System32\Notepad.exe";
```

上面两行代码会产生相同的结果。其中第2行代码中字符串前的@符号告诉编译器该字符串为一个字面字符串。实际上，这会告诉编译器将反斜线字符看作为反斜线字符，而不是转义字符，显然这会使得路径名更具可读性。

既然我们已经学会了如何创建一个字符串，下面让我们来讨论一下可以在 String 对象上执行的一些操作。

12.2.2 字符串的恒定性

String 对象最重要的特性便是其恒定性。也就是说，一个字符串一旦被创建，我们就不可能再将其变长、变短、或者改变其中任何的字符。字符串的恒定性有几个好处。首先它总是允许我们在一个字符串上进行各种操作，而不改变该字符串：

```
if (s.ToLower().Substring(10, 20).EndsWith("exe")) {
...
}
```

这里，ToLower 返回的是一个新的字符串。它并不会改变字符串 s 中的字符。在 ToLower 返回的字符串上进行的 Substring 操作也返回的是一个新的字符串。最后我们又使用 EndsWith 对其末尾的字符串进行检查。由于 ToLower 和 Substring 返回的两个临时字符串不再被应用程序中的代码所引用，因此垃圾收集器将在下一次执行时回收它们的内存。由于这些生存期较短的对象很快就会被回收，所以通常不值得我们在算法上花费时间来提高性能。

字符串的恒定性还意味着当我们操作字符串的时候不会出现线程同步的问题。另外，如果两个字符串的值相等的话，我们可以使两个字符串引用指向同一个字符串对象，而不必非要指向两个不同的字符串对象。这可以减少系统中字符串的数量，从而可以节省内存，这也是字符串驻留(本章稍后讨论)的机理。

出于性能考虑，String类型和CLR被紧密地集成在一起。CLR知道String类型中字段的内存布局，它可以直接访问这些字段。但是，这种性能提升和直接访问特性伴随有一点小小的开发代价：String必须为密封类型。如果我们能够使用String作为基类型来定义自己的类型，我们将可以添加自己的字段，这将会破坏CLR对String所做的假设。另外，这样做也可能会破坏CLR对String对象的恒定性所做的假设。

12.2.3 字符串比较

字符串比较可能是字符串处理中最常用的操作了。幸运的是，String类型提供了好几个静态方法和实例方法，它们允许我们以几种非常有用的方式来对字符串进行比较。表12.1总结了这些方法。

表12.1 有关字符串比较的方法

成 员	成员类型	描 述
Compare	静态方法	返回两个字符串之间的排序情况。不像CompareTo方法，该方法允许我们控制语言文化信息(通过一个CultureInfo对象)，以及指定是否考虑大小写。如果我们需要更多高级的字符串比较功能，可以参见本节后面对CompareInfo类的讨论
CompareTo	实例方法	返回两个字符串之间的排序情况。注意，该方法总是使用和当前调用线程相关联的CultureInfo对象

续表

成 员	成员类型	描 述
StartsWith/EndsWith	实例方法	如果字符串以指定的字符串开头或者结尾，方法将返回true。两个比较都是大小写敏感的，并且使用和当前调用线程相关联的CultureInfo对象。如果所要进行的操作不考虑大小写，可以使用CompareInfo类的IsPrefix和IsSuffix方法
CompareOrdinal	静态方法	如果两个字符串有相同的字符集，方法将返回true(译注：这里的说法有错误，CompareOrdinal方法返回的类型为Int32类型。结果为0表示相等，负数表示第一个字符串比第二个小，正数表示第一个比第二个大。所以这里的true应该是0)。不像其他一些方法，该方法仅比较字符串中字符的码值，它不考虑语言文化信息，并且比较总是大小写敏感的。该方法比其他方法都要快，但是我们不应该使用它来排序那些显示给用户的字符串。因为比较排序的结果可能会与用户期待的有所不同。例如，CompareOrdinal会把“a”排在“A”的后面，因为“a”的码值比“A”大
Equals	静态方法和实例方法	如果两个字符串有相同的字符集，方法将返回true。Equals方法在内部会调用到CompareOrdinal。静态Equals方法会首先检查两个引用是否指向同一个对象，如果是，则返回true，这样就不用再比较字符串中的各个字符了。在使用字符串驻留机制(马上就会谈到)时，比较引用会极大地提高性能
GetHashCode	实例方法	返回字符串的散列码

除了上述列出的这些方法外，String 类型还提供了判等(==)和判异(!=)重载操作符。这些操作符方法在内部都是通过调用 String 的静态 Equals 方法来实现的。

比较字符串一般有两个原因。第一个是判断两个字符串是否表示同样的字符串。第二个是对字符串进行排序，通常是面向用户的一个逻辑表示。要判断两个字符串是否相等，我们可以使用 CompareOrdinal 方法。该方法比较快，因为它仅仅比较字符串中字符的码值。但是，有些字符串即使其中的字符码值不同，它们在逻辑上仍然被认为是相等的。

要在逻辑上判断两个字符串是否相等，我们应该使用 Compare 方法。Compare 方法在内部使用了特定语言文化的排序表(定义于 Unicode 3.1 标准)，该排序表也是.NET 框架的一部分。因为这些表已经包含在.NET 框架中了，所以所有版本的.NET 框架(与底层的操作系统无关)会以同样的方式来比较和排序字符串。当比较两个字符串是否相等时，根据应用程序的需要，Compare 方法在比较时可以选择是否考虑大小写敏感问题。

> **重要** 当比较字符串或者对字符串进行大小写转换时，我们应该使用 InvariantCulture。例如，在比较或者转换路径名、文件名、注册表键值、反射字符串、XML 标记、XML 属性名，以及其他一些与编程相关的字符串时，我们就应该使用 InvariantCulture。实际上，只有在我们比较和转换的字符串要显示给一个特定语言下的用户时，我们才应该使用一个非固定(noninvariant)的语言文化。当然，我们使用的语言文化要和用户选择的语言以及用户所处的国家(可选)相匹配。

下面的代码演示了 CompareOrdinal 方法和 Compare 方法的一些差别：

```
using System;
using System.Globalization;

class App {
    static void Main() {
        String s1 = "Strass", s2 = "Straß";
        Int32 x;
```

```
        // CompareOrdinal 返回非 0 值表示两个字符串的值不同
        x = String.CompareOrdinal(s1, s2);
        Console.WriteLine("CompareOrdinal: '{0}' {2} '{1}'", s1, s2,
            (x == 0) ? "equals" : "does not equal");

        // Compare 返回 0 表示两个字符串的值相同
        x = new CultureInfo("de-DE").CompareInfo.Compare(s1, s2);
        Console.WriteLine("Compare: '{0}' {2} '{1}'", s1, s2,
            (x == 0) ? "equals" : "does not equal");
    }
}
```

编译并运行上面的代码将会产生以下输出：

```
CompareOrdinal: 'Strass' does not equal 'Straß'
Compare: 'Strass' equals 'Straß'
```

另外，我们还应该使用 Compare 方法来对字符串进行排序。在用 Compare 方法对字符串进行排序时，我们应该使 Compare 方法进行大小写敏感的比较。因为如果两个仅因为大小写不同的字符串被认为是相等的，那么我们在一个列表中排序字符串时，用户每一次运行应用程序时可能会发现"相等"的字符串却有不同的顺序，这将会搞乱用户。

Compare 方法在内部首先获取与调用线程相关联的 CurrentCulture，然后读取 CurrentCulture 的 CompareInfo 属性。该属性返回一个 System.Globalization.CompareInfo 对象引用。因为 CompareInfo 对象封装了一个和语言文化相关的字符比较表，所以每一种语言文化仅有一个 CompareInfo 对象。

String 的 Compare 方法在获得一个 CompareInfo 对象之后，便调用该对象的 Compare 方法。String 的 Compare 方法允许我们指定比较操作是否考虑其中字符的大小写，而 CompareInfo 的 Compare 方法提供的比较操作允许我们有更多的控制，这在某些应用程序中是必要的。这些应用程序必须获得一个所期望语言文化的 CompareInfo 对象，然后直接在上面调用 Compare 方法。

具体而言，CompareInfo 的 Compare 方法的几个重载版本接受一个 CompareOptions 枚举类型作为参数，该枚举类型中定义了以下几种符号：IgnoreCase、IgnoreKanaType、IgnoreNonSpace、IgnoreSymbols、IgnoreWidth、None、Ordinal 和 StringSort。关于这些符号完整的描述，请参见.NET 框架文档。

下面的代码演示了语言文化在字符串排序中所起的重要角色：

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Globalization;
using System.Threading;
```

```
class App {

   static void Main() {

      StringBuilder sb = new StringBuilder();
      String[] sign = new String[] { "<", "=", ">" };
      Int32 x;

      // 下面的代码演示了怎样比较不同语言文化的字符串
      String s1 = "coté", s2 = "côte";

      // 以法国法语对字符串进行排序
      x = new CultureInfo("fr-FR").CompareInfo.Compare(s1, s2);
      sb.AppendFormat("fr-FR Compare: {0} {2} {1}", s1, s2, sign[x + 1]);
      sb.Append(Environment.NewLine);

      // 以日本日语对字符串进行排序
      x = new CultureInfo("ja-JP").CompareInfo.Compare(s1, s2);
      sb.AppendFormat("ja-JP Compare: {0} {2} {1}", s1, s2, sign[x + 1]);
      sb.Append(Environment.NewLine);

      // 以当前线程的语言文化对字符串进行排序
      x = Thread.CurrentThread.CurrentCulture.CompareInfo.Compare(s1, s2);
      sb.AppendFormat("{0} Compare: {1} {3} {2}",
         Thread.CurrentThread.CurrentCulture.Name, s1, s2, sign[x + 1]);
      sb.Append(Environment.NewLine);
      sb.Append(Environment.NewLine);

      // 下面的代码演示了怎样用 CompareInfo.Compare 的高
      // 级选项来比较两个日文字符串。这些字符串用平假名(hiragana)
      // 和片假名(katakana)来表示 "shinkansen"(新干线)
      s1 = "しんかんせん";  // ("\u3057\u3093\u304B\u3093\u305b\u3093")
      s2 = "";   // ("\uff7c\uff9d\uff76\uff9d\uff7e\uff9d")

      // 下面是默认比较的结果
      x = String.Compare(s1, s2, true, new CultureInfo("ja-JP"));
      sb.AppendFormat("Simple ja-JP Compare: {0} {2} {1}", s1, s2, sign[x + 1]);
      sb.Append(Environment.NewLine);

      // 下面是忽略假名(Kana)类型的比较结果
      CompareInfo ci = CompareInfo.GetCompareInfo("ja-JP");
      x = ci.Compare(s1, s2, CompareOptions.IgnoreKanaType);
      sb.AppendFormat("Advanced ja-JP Compare: {0} {2} {1}", s1, s2, sign[x + 1]);

      MessageBox.Show(sb.ToString(), "StringSorting Results");
   }
}
```

编译并运行这段代码将产生如图 12.1 所示的输出：

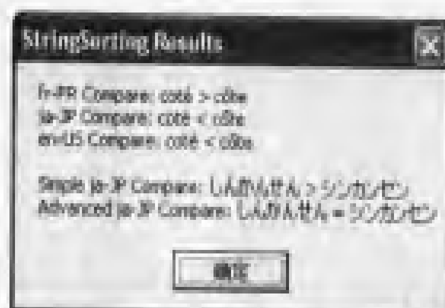


图 12.1 StringSorting 的运行结果

日文字符

要看对话框和源代码中的日文字符，Windows 必须安装东亚语言文件(占用近 230 MB 磁盘空间)。要安装这些文件，首先打开【控制面板】中的【区域和语言选项】对话框(如图 12.2 所示)，选择【语言】选项卡，选中【为东亚语言安装文件】，然后单击【确定】。这将使 Windows 安装东亚语言字体和输入法编辑器(IME，即 Input Method Editor)文件。

图 12.2 使用【区域和语言选项】控制面板对话框安装东亚语言文件

另外，源代码文件不能用 ANSI 编码保存。本例中使用的是 UTF-8 编码，Visual Studio .NET 编辑器和微软的 C#编译器可以很好地处理它们。

除了 Compare 方法外，CompareInfo 类还提供有 IndexOf、IsLastIndexOf、IsPrefix 和 IsSuffix 方法。所有这些方法都提供了接受 CompareOptions 枚举类型参数的重载版本，它们给予了我们比 String 类中相应的方法更多的控制。

12.2.4 字符串驻留

如前一节所述，字符串比较是许多应用程序中常用的操作——它也是一个可以极大地损伤系统性能的操作。损伤性能的原因是字符串比较要求检查字符串中的每一个字符，直到有两个字符被认为不同为止。例如，要比较一个字符串看其中是否包含“Hello”，一个循环必须比较两个字符五次。另外，如果我们在内存中有多个“Hello”字符串的实例，实际上是对内存的一种浪费，因为字符串是恒定不变的。如果在内存中只存储一个“Hello”字符串，并且所有对“Hello”字符串的引用都指向该字符串对象，内存的利用将更为有效。

如果我们的应用程序频繁地对字符串进行判等操作，或者如果我们希望许多字符串对象都有同样的值，我们可以通过利用 CLR 中的字符串驻留(string interning)机制来提高性能。下面的代码可以帮助我们理解字符串的驻留机制：

```
String s = "Hello";
Console.WriteLine(Object.ReferenceEquals("Hello", s));
```

大家猜猜这段代码显示的是“True”还是“False”？许多人可能认为是“False”。毕竟，我们有两个“Hello”字符串，并且 ReferenceEquals 只有在两个引用指向同一个对象时才会返回 true。然而编译并运行这段代码，我们将看到程序输出“True”。下面来解释其中的原因。

当 CLR 初始化时，它会创建一个内部的散列表，其中的键为字符串，值为指向托管堆中字符串对象的引用。刚开始，该表为空(当然)。当 JIT 编译器编译方法时，它会在散列表中查找每一个文本常量字符串。对于上面的代码，编译器首先会查找第一个“Hello”字符串，并且因为它没有找到，它便会在托管堆中构造一个新的 String 对象(指向该字符串)，然后将“Hello”字符串和指向该对象的引用添加到散列表中。接着，JIT 编译器在散列表中查找第二个“Hello”字符串，这一次由于会找到该字符串，所以不会执行任何操作。因为代码中再也没有其他的文本常量字符串，所以 JIT 编译将告完成，代码也开始执行。

当代码执行时，它会在第一行发现需要一个“Hello”字符串引用。于是，CLR 便在其内部的散列表中查找“Hello”，并且会找到它，这样指向先前创建的 String 对象的引用就被保存在变量 s 中。当执行到第二行代码时，CLR 会再一次在其内部的散列表中查找“Hello”，并且仍然会找到它。于是，指向同一个 String 对象的引用会被传递给 Object 的静态方法 ReferenceEquals 作为第一个参数，自然该操作将返回 true。

当一个引用字符串的方法被 JIT 编译时，所有嵌入在源代码中的文本常量字符串总会被添加到 CLR 内部的散列表中。但是，运行时动态创建的字符串呢？看看下面代码输出的是什么？

```
String s1 = "Hello";
String s2 = "Hel";
String s3 = s2 + "lo";
Console.WriteLine(Object.ReferenceEquals(s1, s3));
Console.WriteLine(s1.Equals(s3));
```

在上面的代码中，s2 引用的字符串("Hel")和一个文本常量字符串("lo")被连接在一起。结果是一个新构造的、位于托管堆中由 s3 引用的字符串对象。

这个动态创建的字符串包含的是一个"Hello"，但是它并没有被添加到 CLR 内部的散列表中。因此，ReferenceEquals 将返回 false，因为两个引用指向的是不同的字符串对象。但是调用 Equals 产生的结果将为 true，因为两个字符串实际上仍表示着同样的字符集。(译注：如果将 s3 赋值为"Hel" + "lo"，ReferenceEquals 将返回 true，这是因为 C#编译器在将代码编译成 IL 指令时会直接将两个文本常量字符串连接为一个文本常量字符串"Hello"，这可以用 ILDasm.exe 看到。)很明显，ReferenceEquals 执行的效率要高于 Equals。如果一个应用程序中所有的字符串比较都仅仅是比较引用而非字符集的话，那么系统的性能将会有很大的提升。另外，如果有一种方法可以将含有相同字符集的动态字符串变为托管堆中的一个字符串对象的话，应用程序需要的对象也将更少，从而也可以提升系统性能。幸运的是，String 类型提供的两个静态方法允许我们做到这一点：

```
public static String Intern(String str);
public static String IsInterned(String str);
```

第一个方法为 Intern，它接受一个 String 参数，然后在 CLR 内部的散列表中查找它。如果能够找到该字符串，Intern 将返回已经存在的 String 对象的引用。(译注：如果找不到该字符串，该字符串将被添加到 CLR 内部的散列表中，Intern 最后也会返回它的引用。)如果应用程序不再保存有原来 String 对象(译注：指作为参数传递给 Intern 方法的那个 String 对象)的引用，垃圾收集器将可以回收其所占用的内存。下面对前面的代码用 Intern 方法进行了重新改写：

```
String s1 = "Hello";
String s2 = "Hel";
String s3 = s2 + "lo";
s3 = String.Intern(s3);
Console.WriteLine(Object.ReferenceEquals(s1, s3));
Console.WriteLine(s1.Equals(s3));
```

现在 ReferenceEquals 将返回一个 true 值，并且比较操作的执行效率也更高。另外，s3 原来引用的 String 对象现在也可以接受垃圾收集。注意，因为 String 的 Intern 方法所做的工作，这段代码执行起来实际上要比前面的慢一些。只有当我们需要在应用程序中多次比较同一个字符串时，我们才应该

运用字符串驻留技术。否则，系统的性能会因此而损伤，而不是提高。

注意垃圾收集器不会释放 CLR 内部的散列表中引用的字符串对象，这些 String 对象的引用一直被散列表保存着。只有当进程中所有的应用程序域(AppDomain)都不再引用这些字符串对象时，它们才会被释放。这是因为字符串驻留是按进程为单位进行的，也就是说一个字符串对象可以被同一个进程中的多个应用程序域所访问，这也会为我们的应用程序节省不少的内存。这种从多个应用程序域中访问同一个字符串的能力也对性能的提高有所帮助，因为字符串在同一个进程中跨越应用程序域时将没有必要再进行封送处理(marshal)，需要做封送处理的仅仅是它们的引用。

如上所述，String 类型还提供了一个静态方法 IsInterned。和 Intern 方法一样，IsInterned 方法也接受一个 String 作为参数，并会在 CLR 内部的散列表中查找它。如果 CLR 内部的散列表中含有该字符串，IsInterned 将返回散列表中保存的字符串对象引用。如果散列表中不含该字符串，IsInterned 将返回 null，它并不会将该字符串添加到散列表中。

实际上，C#编译器就使用了 IsInterned 方法对 switch/case 语句进行了性能优化。看下面的代码：

```
using System;

class App {
    static void Main() {
        Lookup("Jeff", "Richter");
        Lookup("Fred", "Flintstone");
    }

    static void Lookup(String firstName, String lastName) {
        switch (firstName + " " + lastName) {
            case "Jeff Richter":
                Console.WriteLine("Jeff");
                break;
            default:
                Console.WriteLine("Unknown");
                break;
        }
    }
}
```

编译上面的代码，然后使用 ILDasm.exe 查看其生成的 IL 代码，我们将会看到以下内容。

```
.method private hidebysig static void Lookup( string firstName,
    string lastName) cil managed
{
    // Code size  53 (0x35)
    .maxstack 3
    .locals (object V_0)

    // 将firstName、" "、和lastName连接为一个字符串
    IL_0000: ldarg.0
    IL_0001: ldstr     " "
    IL_0006: ldarg.1
    IL_0007: call          string [mscorlib]System.String::Concat(string,
                                                                  string,
                                                                  string)
    // 复制连接生成的字符串
    IL_000c: dup

    // 将其中的一个字符串引用存储在临时堆栈变量中
    IL_000d: stloc.0

    // 如果Concat返回null，则跳转到IL_002a
    IL_000e: brfalse.s IL_002a

    // 判断连接生成的字符串是否在CLR内部的散列表中
    IL_0010: ldloc.0
    IL_0011: call string [mscorlib]System.String::IsInterned(string)

    // 将临时变量修改为IsInterned返回的字符串
    // 注意：null表示连接字符串不在CLR内部的散列表中
    IL_0016: stloc.0

    // 将switch后被修改的字符串引用和"Jeff Richter"字符串
    // 引用(两个引用都驻留在CLR内部的散列表中)相比较
    IL_0017: ldloc.0
    IL_0018: ldstr "Jeff Richter"

    // 如果两个引用指向不同的对象，则跳转到IL_002a
    IL_001d: bne.un.s IL_002a

    // 两个引用匹配：将"Jeff"显示到控制台上，然后返回
    IL_001f: ldstr "Jeff"
    IL_0024: call void [mscorlib]System.Console::WriteLine(string)
    IL_0029: ret
```

```
    // 将"Unknown"显示到控制台上，然后返回
    IL_002a: ldstr "Unknown"
    IL_002f: call void [mscorlib]System.Console::WriteLine(string)
    IL_0034: ret
} // end of method App::Lookup
```

这段代码中最重要的一点就是对IsInterned方法的调用，注意IsInterned方法接受的参数为switch语句中指定的字符串。如果IsInterned方法返回null，那么它不可能匹配任何case语句中的字符串，从而导致default处的代码执行："Unknown"将被显示到控制台上。但是，如果IsInterned方法发现switch语句中指定的字符串位于CLR内部的散列表中，它将返回散列表中保存的字符串引用。在这之后，系统将IsInterned方法返回的字符串引用和每个case语句指定的字符串引用(这些引用都驻留在CLR内部的散列表中)相比较。显然，比较引用较之于比较字符串中所有的字符要快得多，因此case语句的执行会很快。

12.2.5 字符串池技术

当编译源代码时，我们的编译器必须处理每一个文本常量字符串，并将其放入生成模块的元数据中。如果同样的文本常量字符串在我们的源代码中出现多次，那么将所有这些字符串都放到元数据中将导致生成文件的大小急剧膨胀。

为了消除这种膨胀效应，许多编译器(包括C#编译器)都在生成模块的元数据中只写入一次这样的字符串。然后将所有引用该字符串的代码改变为引用元数据中的一个字符串。这种将一个多次出现的字符串合并为一个实例的能力可以极大地减少生成模块的大小。实际上，这种处理方法并不新鲜——C/C++编译器已经使用很多年了。(微软的C/C++编译器称之为字符串池，string pooling。)字符串池技术是另一种提高字符串处理性能的方法，我们应该对此有所了解。

12.2.6 查看字符串中的字符

虽然字符串比较对于排序和判等非常有用，但有时候我们还需要查看一个字符串中的字符。String类型提供了几种方法帮助我们来实现这一操作。表12.2总结了这些方法。

表12.2 查看字符串中字符的一些方法

成 员	成员类型	描 述
Length	实例只读属性	返回字符串中字符的数量
Chars	实例只读索引器属性	返回字符串中指定索引位置的字符

续表

成 员	成员类型	描 述
GetEnumerator	实例方法	返回一个 IEnumerator 用于遍历字符串中所有的字符
ToCharArray	实例方法	返回一个包含字符串中一部分字符的 Char[ ]
IndexOf LastIndexOf	实例方法	返回第一个/最后一个与指定的值相匹配的字符/字符串的索引
IndexOfAny LastIndexOfAny	实例方法	返回第一个/最后一个与指定的字符数组相匹配的字符的索引

实际上，System.Char 表示的 16 位 Unicode 码值并不必然等同一个抽象的 Unicode 字符。例如，某些抽象 Unicode 字符是两个码值的组合。U+0625(下面带有闭锁音符的阿拉伯字母 Alef)和 U+0650(阿拉伯字母 Kasra)就可以组合形成一个抽象字符。

另外，16 位值对于某些 Unicode 抽象字符来说是不够的。这些字符需要使用两个 16 位值来表示。其中第一个码值称为高位代理项(high surrogate)，第二个码值称为低位代理项(low surrogate)。高位代理项位于 U+D800 和 U+DBFF 之间，低位代理项位于 U+DC00 和 U+DFFF 之间。使用代理项使得 Unicode 可以表示的不同字符多于 1 百万个。

代理字符在美国和欧洲很少用，但是在东亚使用的却很频繁。要正确地处理抽象 Unicode 字符，我们应该使用 System.Globalization.StringInfo 类型。我们可以调用该类型的静态方法 GetTextElementEnumerator 来获取一个 System.Globalization.TextElementEnumerator 对象，该对象允许我们枚举字符串中包括的所有抽象 Unicode 字符。

另外，我们还可以调用 StringInfo 的静态方法 ParseCombiningCharacters 来获得一个 Int32 数组。数组的长度表示字符串中包含的抽象 Unicode 字符的个数。数组的每一个元素表示字符串中每一个抽象 Unicode 字符的第一个码值出现的索引。

下面的代码演示了怎样正确使用 StringInfo 的 GetTextElementEnumerator 方法和 ParseCombiningCharacters 方法来操作一个字符串中的抽象 Unicode 字符：

```
using System;
using System.Text;
using System.Windows.Forms;
using System.Globalization;

class App {
    static void Main() {
        // 下面的字符串中包括由多个码值组合而成的字符
        String s = "a\u0304\u0308bc\u0327";
        EnumTextElements(s);
        EnumTextElementIndexes(s);
    }

    static void EnumTextElements(String s) {
        StringBuilder sb = new StringBuilder();

        TextElementEnumerator charEnum =
            StringInfo.GetTextElementEnumerator(s);
        while (charEnum.MoveNext()) {
            sb.AppendFormat(
                "Character at index {0} is '{1}'{2}",
                charEnum.ElementIndex, charEnum.GetTextElement(),
                Environment.NewLine);
        }
        MessageBox.Show(sb.ToString(),
            "Result of GetTextElementEnumerator");
    }
    static void EnumTextElementIndexes(String s) {
        StringBuilder sb = new StringBuilder();

        Int32[] textElemIndex = StringInfo.ParseCombiningCharacters(s);
        for (Int32 i = 0; i < textElemIndex.Length; i++) {
            sb.AppendFormat(
                "Character {0} starts at index {1}{2}",
                i, textElemIndex[i], Environment.NewLine);
        }
        MessageBox.Show(sb.ToString(),
            "Result of ParseCombiningCharacters");
    }
}
```

编译并运行上面的代码，将产生如图 12.3 和图 12.4 所示的消息对话框。

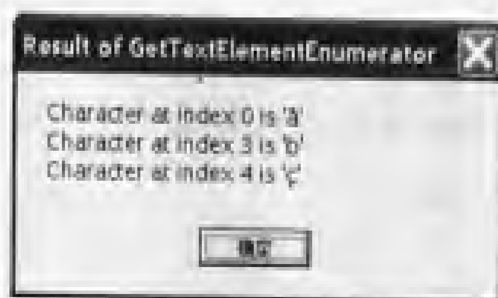


图 12.3 GetTextElementEnumerator 方法的执行结果

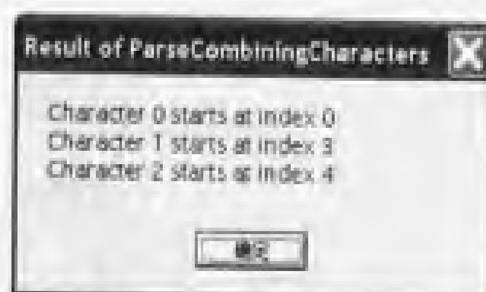


图 12.4 ParseCombiningCharacters 方法的执行结果

> **注意** 要看到图 12.3 中消息对话框上的文字，必须打开 Windows 的【显示属性】对话框，将消息对话框文本的字体改为 Lucida Sans Unicode，因为该字体包含了这些组合字符的字形。这也是为什么代码中没有将结果显示到控制台上的原因。

本例中，我们首先调用了 StringInfo 的 GetTextElementEnumerator 方法，该方法接受一个 String，返回一个 TextElementEnumerator 对象。TextElementEnumerator 对象和其他枚举器对象的使用方式类似。TextElementEnumerator 对象有一个只读属性 ElementIndex，其返回原字符串中当前字符开始处的码值的索引。TextElementEnumerator 对象还有一个 GetTextElement 方法，其返回的字符串中包含所有组成当前字符的码值。

另外，StringInfo 还提供有一个静态方法 ParseCombiningCharacters，该方法对一个字符串进行分析并返回一个 Int32 数组。返回数组中的每一个元素是一个抽象字符起始处的码点(code-point)单元的索引。.NET 框架 SDK 文档中有一个例子演示了如何调用该方法。注意 StringInfo 还定义了一个公有构造器，但这是一个 bug，因为没有理由构造一个 StringInfo 实例。

12.2.7 其他字符串操作

String 类型还提供了其他几种方法允许我们拷贝一个字符串或其一部分。表 12.3 总结了这些方法。

表 12.3 拷贝字符串的一些方法

成 员	方法类型	描 述
Clone	实例方法	返回同当前对象(this)相同的引用。这种做法可行是因为 String 对象具有恒定不变性。该方法实现了 String 的 ICloneable 接口
Copy	静态方法	返回一个新的字符串，新字符串为指定字符串的一个副本。该方法很少用，只是为了帮助应用程序将字符串看作为一些标记。通常情况下，有着相同字符集的字符串被驻留为同一个字符串。但该方法会创建一个新的字符串对象，这样即使两个字符串包含同样的字符集，它们的引用也不同
CopyTo	实例方法	将字符串中的一部分字符拷贝到一个字符数组中
Substring	实例方法	返回一个新构造的字符串，新字符串表示原来字符串的一部分
ToString	实例方法	返回当前对象的一个引用(this)

除了上述这些方法外，String 还提供了许多静态方法和实例方法，例如 Insert、Remove、PadLeft、Replace、Split、Join、ToLower、ToUpper、Trim、Concat、Format，等等。再强调一遍，这些方法返回的都是新创建的字符串对象：因为字符串是恒定不变的，一经创建，它们便不能再被做任何改变。

12.3 高效地动态创建字符串

由于 String 类型表示的是一个恒定不变的字符串，所以 FCL 提供了另外一种类型 System.Text.StringBuilder，它允许我们通过对字符串和字符执行动态操作来创建 String 对象。我们可以将 StringBuilder 看作是一个特殊的构造器，然后将它和 FCL 中的其他类一起使用来创建字符串对象。通常情况下，我们应该将方法设计为接受 String 类型的参数，而不是 StringBuilder 类型的参数，除非我们定义的方法返回的是一个由方法自身动态创建的字符串。

StringBuilder 对象在内部有一个指向 Char 结构数组的字段。StringBuilder 的成员允许我们操作该字符数组，从而高效地压缩字符串或者改变字符串中的字符。如果字符串的增长超出了原来分配的字符数组，StringBuilder 会自动分配一个新的、更大的数组，并将原来字符数组中的字符拷贝到新的数组中，然后便开始使用这个新的数组。先前的数组将成为可被垃圾收集器回收的对象。

在用 StringBuilder 完成字符串的构造之后，我们可以调用 StringBuilder 的 ToString 方法来将 StringBuilder 中的字符数组“转换”为一个 String。该方法返回一个 StringBuilder 内部维护的字符串字段。这使得 StringBuilder 的 ToString 方法执行很快，因为不必拷贝字符数组。

StringBuilder 的 ToString 方法返回的 String 必须是恒定不变的。如果我们调用的 StringBuilder 上的方法试图改变 StringBuilder 中维护的字符串字段时，StringBuilder 可以判断出它的 ToString 方法先前是否被调用过，如果是，它会在内部创建并使用一个新的字符数组，这使得我们执行的操作不会影响先前调用 ToString 所返回的字符串。

12.3.1 构造 StringBuilder 对象

和 String 不同，CLR 没有关于 StringBuilder 的任何特殊知识。大多数语言(包括 C#)也都不把 StringBuilder 看作是基元类型。构造 StringBuilder 对象的方式和构造其他非基元类型对象的方式没有什么不同：

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

StringBuilder 类型提供了许多构造器。每个构造器的工作就是分配和初始化 StringBuilder 对象维护的三个内部字段：

- **最大容量** 这是一个 Int32 字段，它表示字符串中可以容纳的字符的最大个数。默认值为 Int32.MaxValue(大约 20 亿)。通常情况下不要改变该值。但是，有时候我们可能需要指定一个更小的最大容量来确保创建的字符串不会超过某个长度。StringBuilder 的最大容量字段一旦被创建，就不可以再做任何改变。

- **容量** 这是一个 Int32 字段，它表示 StringBuilder 中维护的字符数组字段的长度。其默认值为 16。如果我们知道 StringBuilder 中维护的字符的个数，我们应该在构造 StringBuilder 对象时将容量设为该值。

当我们向 StringBuilder 的字符数组追加字符时，StringBuilder 会检测数组的增长是否超出了其容量。如果超出，StringBuilder 会自动将容量字段加倍，并分配一个新的数组(其大小为更新后的容量)，然后将原数组中的字符拷贝到新分配的数组中。原数组则成为可被垃圾收集的对象。数组的动态增长会损伤系统的性能，我们应该通过设置一个合理的初始容量来避免这一点。

- **字符数组** 这是一个 Char 结构数组，它维护着 StringBuilder 所表示的字符串中的字符集合。该数组中字符的数量总是小于或等于 StringBuilder 的容量、以及最大容量字段。我们可以用 StringBuilder 的 Length 属性来获得该数组中字符的数量。当构造一个 StringBuilder 时，我们可以传递一个 String 来初始化该字符数组。如果我们没有指定字符串，该数组初始的时候将不包括任何字符——也就是说，Length 属性将返回 0 值。

12.3.2 StringBuilder 的成员

和 String 不同的是，一个 StringBuilder 表示一个可变的字符串。这意味着大多数 StringBuilder 的成员都会改变 StringBuilder 中维护的字符数组的内容，并且不会导致在托管堆中分配新的对象。只有在两种情况下，一个 StringBuilder 才会分配新的对象：

- 动态构造一个长度超过预设容量的字符串。
- 在调用 StringBuilder 的 ToString 方法之后，试图改变字符数组。

需要清楚的是，为了获得更高的性能，StringBuilder 的方法并不保证线程安全。这在一般情况下没什么，因为用多个线程来访问一个 StringBuilder 对象的情况并不常见。如果我们的应用程序需要对 StringBuilder 对象做线程安全的操作，那么我们必须显式添加线程同步代码。

表 12.4 总结了 StringBuilder 的成员。

表12.4 StringBuilder的成员

成 员	成员类型	描 述
MaxCapacity	只读属性	返回字符串中可以容纳的字符的最大个数
Capacity	读写属性	读取或者设置字符数组的大小。试图将Capacity设置为比字符串长度更小的容量将抛出ArgumentOutOfRangeException异常
EnsureCapacity	方法	确保字符数组的大小至少为该方法指定的值。如果指定的值大于StringBuilder目前的容量，目前的容量将会被扩大。如果StringBuilder目前的容量已经大于方法指定的值，将不发生任何改变
Length	读写属性	读取或者设置字符串中字符的个数。该数值一般比字符数组目前的容量要小
ToString	方法	该方法无参的那个版本返回一个表示StringBuilder中字符数组字段的String。该方法的效率很高，因为它不用创建新的String对象。在ToString被调用之后，任何试图改变StringBuilder中字符数组的操作都将导致StringBuilder重新分配并使用一个新的数组(根据旧数组进行初始化)。接受startIndex和length参数的那个版本的ToString则会创建一个新的String对象来表示StringBuilder中字符串的一部分
Chars	读写索引器属性	读取或者设置字符数组中指定索引位置的字符。在C#中，这是一个索引器(含参属性)，这使得我们可以用类似访问数组的语法([ ])来访问它
AppendInsert	方法*	向字符数组中追加或者插入一个对象，并在必要的时候自动增长数组。方法首先会使用常规格式，以及和调用线程相关联的语言文化将传入的对象转换为一个字符串

续表

成　员	成员类型	描　述
AppendFormat	方法	向字符数组追加指定的对象，并在必要的时候自动增长数组。方法首先会使用调用者指定的格式和语言文化信息将对象转换为一个字符串。AppendFormat 是 StringBuilder 对象最常用的方法之一
Replace	方法	将字符数组中的一个字符或字符串替换为另一个字符或字符串
Remove	方法	从字符数组中删除指定范围的字符
Equals	方法	如果两个 StringBuilder 对象有相同的最大容量、容量、以及字符数组，方法将返回 true

StringBuilder 的大多数方法都返回指向同一个 StringBuilder 对象的引用。这使得我们可以使用一种方便的语法来将几个操作放在一起执行：

```
StringBuilder sb = new StringBuilder();
String s = sb.AppendFormat("{0} {1}", "Jeffrey", "Richter");
Replace(' ', '-').Remove(4, 3).ToString();
Console.WriteLine(s);                    // "Jeff-Richter"
```

大家可能已经注意到了 String 和 StringBuilder 并没有完全对等的方法。例如，String 为我们提供了 ToLower、ToUpper、EndsWiths、PadLeft、Trim 等方法。而 StringBuilder 却没有提供任何这些方法。另一方面，StringBuilder 为我们提供了一个更为丰富的 Replace 方法，它允许我们在字符串的一部分(而不是整个字符串)中进行字符和字符串替换操作。这很不幸，因为为了完成某些操作，我们可能必须在 String 和 StringBuilder 之间来回转换。例如，要完成“创建一个字符串，然后将其所有的字符都转换为大写，最后再插入一个字符串”这样的操作，我们的代码可能就要像下面这样编写：

```
// 构造一个 StringBuilder 进行字符串操作
StringBuilder sb = new StringBuilder();

// 使用 StringBuilder 执行字符串操作
sb.AppendFormat("{0} {1}", "Jeffrey", "Richter").Replace(" ", "-");
```

```
// 为了将所有的字符都变为大写，我们需要首先
// 将 StringBuilder 转换为一个 String
String s = sb.ToString().ToUpper();

// 清除 StringBuilder (分配一个新的 Char 数组)
sb.Length = 0;

// 将大写的 String 追加到 StringBuilder 上，
// 然后执行更多的操作
sb.Append(s).Insert(8, "Marc-");

// 将 StringBuilder 再转换为一个 String
s = sb.ToString();

// 将 String 显示给用户
Console.WriteLine(s); // "JEFFREY-Marc-RICHTER"
```

仅仅因为 StringBuilder 没有提供 String 所具有的一些操作，我们就必须像上面这样“迂回”地编写代码，显然这很不方便。我希望微软在.NET 框架后续的版本中能够为 StringBuilder 类添加更多的字符串操作方法，使其成为一个更为完整的类。

12.4 获取对象的字符串表达形式

在应用程序的开发过程中，我们经常需要获取一个对象的字符串表达形式。当希望将一个数值类型(例如 Byte、Int32、Single 等等)或者一个 DateTime 对象显示给用户时，这种操作就显得很有必要。因为.NET 框架是一个面向对象的平台，所以每一个类型需要自己负责将其实例的“值”转换为对应的字符串形式。在设计如何使类型实现这一点时，FCL 的设计者们决定构造出一种能够被广泛使用的模式。本节将向大家描述这种模式。

我们知道，System.Object 中定义了一个公有的无参 ToString 方法，由于所有的类型都继承自 System.Object，因此我们可以在任何类型的实例上通过调用 ToString 来获得该实例的字符串表达形式。从语义上来讲，ToString 返回表示对象当前值的一个 String，该字符串应该使用和调用线程相关联的语言文化进行格式化。比如，一个数值的字符串表达形式就应该使用正确的十进制分隔符、数字分组符、以及其他与调用线程相关联的语言文化。

但是 System.Object 实现的 ToString 方法仅仅只是返回对象类型的全名。这样的返回值并不是很有用，但对于许多不能提供有意义的字符串表达的类型来讲还是比较合理的。例如，一个 FileStream 或 Hashtable 这样的对象的字符串表达形式该是什么样子呢？

如果一个类型希望为它的使用者提供一个合理的方式来获取表示其实例当前值的一个字符串，那么它就应该重写ToString方法。对于所有FCL中内置的基本类型(如Byte、Int32、Int64、Double等)，微软已经重写了它们的ToString方法，其实现为返回一个面向特定语言文化的字符串。

12.4.1 特定格式与语言文化

然而，无参的ToString方法有两个问题。首先，调用者不能对字符串的格式有任何控制。例如，一个应用程序可能希望将一个数值格式化为一个货币形式的字符串、或者一个十进制形式的字符串、或者一个百分比形式的字符串、或者一个十六进制的字符串。其次，调用者不能选择一种特定的语言文化来格式化字符串。第二个问题对于服务器端代码来说比客户端代码更为麻烦。在极少数情况下，一个应用程序格式化字符串时需要使用的语言文化有别于和调用线程相关联的语言文化。为了对字符串格式化有更多的控制，我们需要一个允许我们指定格式和语言文化信息的ToString方法。

如果我们希望自己的类型能为调用者提供格式和语言文化选择的话，我们就应该使其实现System.IFormattable接口：

```
public interface IFormattable {
   String ToString(String format, IFormatProvider formatProvider);
}
```

在FCL中，所有的基本类型(Byte、SByte、Int16/UInt16、Int32/UInt32、Int64/UInt64、Single、Double、Decimal以及DateTime)都实现了该接口。一些其他的类型，如GUID，也实现了该接口。另外，每个枚举类型也会自动实现该接口，这使得我们可以根据存储在枚举实例中的数值来获得一个有意义的字符串符号。

IFormattable接口的ToString方法接受两个参数。第一个参数format是一个字符串，它告诉方法应该怎样来格式化对象。第二个参数formatProvider是一个实现了System.IFormatProvider接口的类型实例，该类型为ToString方法提供了特定的语言文化信息。本章稍后会谈到其中的实现原理。

实现了IFormattable接口的类型要确定它应该识别哪些表示特定格式的字符串。如果我们传递了一个它不能识别的format字符串，那么它应该抛出一个System.FormatException异常。

微软在FCL中定义的许多类型都能识别多种格式。例如DateTime类型支持用“d”来表示短日期格式，用“D”来表示长日期格式，用“g”来表示常规日期/时间格式，用“M”来表示“月/日”格式，用“s”来表示可排序的日期/时间格式，用“T”来表示时间格式，用“u”来表示ISO 8601格式标准定义的通用时间格式，用“U”来表示长日期格式中的通用时间格式，用“Y”来表示“年/月”格式，等等。而所有的枚举类型都支持用“G”来表示常规格式，用“F”来表示位标记格式，用“D”

来表示十进制格式，用“X”来表示十六进制格式。本书第13章将对枚举类型的格式化有更详细的讨论。

另外，所有.NET框架内置的数值类型都支持用“C”来表示货币格式，用“D”来表示十进制格式，用“E”来表示科学计数法格式(指数)，用“F”来表示定点格式，用“G”来表示常规格式，用“N”来表示数字格式，用“P”来表示百分比格式，用“R”来表示回程(round-trip)格式(译注：回程格式可以保证一个被转换为字符串的数值被再次分析为数值后，所得的数值和原来的数值相同)，用“X”来表示十六进制格式。实际上，如果这些简单的格式字符串不能满足我们的需求时，我们还可以使用图片格式字符串(picture format string)。图片格式字符串包含的特殊字符可以告诉ToString方法应该显示多少个数字、将十进制分隔符放在哪里，以及在十进制分隔符后放置多少个数字等。关于格式字符串的更多信息，可参见.NET框架SDK中的“格式字符串”。

调用ToString方法时为格式字符串参数传递null值和传递“G”的效果等同。换句话说，对象默认情况下使用“常规格式”来格式化自己。当我们实现一个类型时，应该为其选择一个我们认为最常用的格式，也就是“常规格式”。顺便提一句，不带参数的ToString方法假定调用者期望的是常规格式。

格式字符串就谈到这里，下面我们来谈谈语言文化信息。默认情况下，系统使用和调用线程相关联的语言文化信息来格式化字符串。不带参数的ToString方法就是这样实现的。如果为参数formatProvider传递的值为null，那么IFormattable接口的ToString方法也将使用和调用线程相关联的语言文化信息来格式化字符串。

当我们对数值(包括货币、整数、浮点数、百分数)、日期、时间进行格式化时，语言文化信息将会发挥作用。但是GUID类型的ToString方法返回的只是一个表示GUID值的字符串，在生成GUID字符串时没有必要考虑调用线程的当前语言文化。

当格式化一个数值时，ToString方法会首先查看我们传递给它的formatProvider参数。如果formatProvider参数的值为null，那么ToString方法会通过读取属性System.Threading.Thread.CurrentThread.CurrentCulture的值来确定和调用线程相关联的语言文化。该属性返回一个System.Globalization.CultureInfo类型的实例。

利用CultureInfo类型实例，ToString方法会根据格式化的是一个数值、还是一个日期/时间来读取它的NumberFormat或者DateTimeFormat属性。两个属性分别返回一个System.Globalization.NumberFormatInfo或者System.Globalization.DateTimeFormatInfo实例。NumberFormatInfo类型定义有许多属性，如CurrencyDecimalSeparator、CurrencySymbol、NegativeSign、NumberGroupSeparator以及PercentSymbol。类似地，DateTimeFormatInfo类型也定义有一些属性，如Calendar、DateSeparator、DayNames、LongDatePattern、ShortTimePattern和TimeSeparator。ToString方法在构造和格式化字符串时会读取这些属性。

当调用 IFormattable 接口的 ToString 方法时，我们可以传递给其一个实现了 IFormatProvider 接口的类型实例来代替 null 值：

```
public interface IFormatProvider {
    Object GetFormat(Type formatType);
}
```

下面是 IFormatProvider 接口背后的基本思想：当一个类型实现该接口时，它的意思是该类型的实例知道如何提供特定语言文化的格式化信息，而和调用线程相关联的语言文化信息应该被忽略。

System.Globalization.CultureInfo 类型是 FCL 中很少的几个实现了 IFormatProvider 接口的类型之一。如果我们希望为某一种语言文化，例如越南语(Vietnam)，格式化一个字符串，我们应该构造一个 CultureInfo 对象，并将该对象作为 formatProvider 参数传递给 ToString 方法。下面的代码获取一个 Decimal 数值的字符串表示，其中的字符串会被格式化为越南货币格式。

```
Decimal price = 123.54M;
String s = price.ToString("C", new CultureInfo("vi-VN"));
System.Windows.Forms.MessageBox.Show(s);
```

编译并运行上面的代码，我们将会看到如图 12.5 所示的消息框。

图 12.5　一个被格式化为越南货币的数值

Decimal 的 ToString 方法在内部会发现 formatProvider 参数不为 null，于是它便调用其上的 GetFormat 方法：

```
NumberFormatInfo nfi = (NumberFormatInfo)
    formatProvider.GetFormat(typeof(NumberFormatInfo));
```

ToString 方法正是利用这种技巧通过 CultureInfo 对象来查询数值格式化信息的。数值类型(例如 Decimal)只查询数值格式化信息。其他类型(例如 DateTime)则可以像下面这样通过调用 GetFormat 来查询针对它们的格式化信息：

```
DateTimeFormatInfo dtfi = (DateTimeFormatInfo)
    formatProvider.GetFormat(typeof(DateTimeFormatInfo));
```

实际上，因为 GetFormat 的参数可以识别任何类型，所以我们可以使用该方法来查询任何类型的格式化信息。然而，.NET 框架版本 1 中的类型调用 GetFormat 只能查询有关数值、或日期/时间的格式化信息。但是在将来的版本中，我们有可能查询其他一些格式化信息。

顺便提一句，如果我们希望获取一个对象的字符串、而又不需要针对任何特殊的语言文化进行格式化，那么我们应该调用 System.Globalization.CultureInfo 的静态属性 InvariantCulture，并将其传递给 ToString 方法的 formatProvider 参数：

```
Decimal price = 123.54M;
String s = price.ToString("C", CultureInfo.InvariantCulture);
System.Windows.Forms.MessageBox.Show(s);
```

编译并运行上面的代码，我们将会看到如图 12.6 所示的消息框。注意显示的字符串中的第一个字符‘¤’，这是货币的国际符号(U+00A4)。

图 12.6 一个被格式化为语言文化中性的货币数值

一般情况下，我们不会将一个用固定语言文化(invariant culture)(译注：固定语言文化即不区分语言文化，又称语言文化中性)所格式化的字符串显示给用户。我们通常只将这样的字符串保存在一个数据文件中以便以后进行分析。基本上而言，固定语言文化允许我们在不同的语言文化之间传递相互可理解的字符串。

在 FCL 中，总共只有三个类型实现了 IFormatProvider 接口。第一个是 CultureInfo，前面已经解释过了，另外两个是 NumberFormatInfo 和 DateTimeFormatInfo。当我们在一个 NumberFormatInfo 对象上调用 GetFormat 时，该方法会检查请求的类型是否为一个 NumberFormatInfo。如果是 NumberFormatInfo，方法将返回 this；否则返回 null。类似地，在一个 DateTimeFormatInfo 对象上调用 GetFormat 时，如果请求的类型为一个 DateTimeFormatInfo，那么返回 this；否则返回 null。这两个类型实现 IFormatProvider 接口的目的仅仅是为了方便编程。

在试图获取一个对象的字符串表示时，我们通常会指定一种格式、并使用和调用线程相关联的语言文化。出于这种原因，我们经常在调用 ToString 方法时为其 format 参数传递一个 String，而为其 formatProvider 参数传递一个 null。为了使 ToString 的调用更加方便，许多类型都为 ToString 方法提供了几个重载的版本。例如 Decimal 类型就提供了如下四个不同的 ToString 方法：

```
// 该版本调用 ToString(null, null)
// 含义：常规格式，调用线程的语言文化信息
String ToString();

// 该版本是 ToString 真正的实现之处
// 该版本是对 IFormattable.ToString 方法的实现
// 含义：调用者指定的格式和语言文化信息
String ToString(String format, IFormatProvider formatProvider);

// 该版本调用 ToString(format, null)
// 含义：调用者指定的格式，调用线程的语言文化信息
String ToString(String format);

// 该版本调用 ToString(null, formatProvider)
// 含义：常规格式，调用者指定的语言文化信息
String ToString(IFormatProvider formatProvider);
```

12.4.2 将多个对象格式化为一个字符串

到目前为止，本章已经向大家解释了一个类型怎样格式化它自己的对象。但是，有些时候，我们希望构造的字符串能够包含多个经过格式化的对象。例如，下面的字符串就包含一个日期、一个人名和一个年龄：

```
String s = String.Format("On {0}, {1} is {2} years old.",
   DateTime.Now, "Wallace", 35);
Console.WriteLine(s);
```

如果我们在 2002 年 1 月 23 日下午 4:37 编译并运行该段代码，我们将会得到以下输出：

```
On 1/23/2002 4:37:37 PM, Wallace is 35 years old.
```

String 的静态方法 Format 接受一个格式字符串，该格式字符串使用大括号中的数字来标识可替换的参数。上面例子中的格式字符串告诉 Format 方法用格式字符串后的第一个参数(即当前的日期/时间)来代替“{0}”，用格式字符串后的第二个参数(即“Wallace”)来代替“{1}”，用格式字符串后的第三个参数(即 35)来代替“{2}”。

Format 方法在内部会首先调用每个对象的 ToString 方法来获取对象的字符串表示，然后它将这些字符串连接在一起并返回。这种做法对于大多数情况已经够用了，但是这意味着所有的对象都只能用它们的常规格式和调用线程的语言文化信息来进行格式化。

如果希望能对格式化对象有更多的控制，我们可以在大括号中指定格式信息。例如，下面的代码在前面例子的基础上向可替换的参数 0 和 2 中添加了一些格式信息：

```
String s = String.Format("On {0:D}, {1} is {2:E} years old.",
    DateTime.Now, "Wallace", 35);
Console.WriteLine(s);
```

如果我们在 2002 年 1 月 23 日下午 4:37 编译并运行该段代码，我们将会得到以下输出：

```
On Wednesday, January 23, 2002, Wallace is 3.500000E+001 years old.
```

当 Format 方法分析格式字符串时，它会调用替换参数 0 实现的 IFormattable 接口中的 ToString 方法，并为其传递 "D" 和 null 两个参数。类似地，Format 也会调用替换参数 2 实现的 IFormattable 接口中的 ToString 方法，并为其传递 "E" 和 null 两个参数。如果替换参数的类型没有实现 IFormattable 接口，那么 Format 将调用它的无参 ToString 方法，最后向结果中添加的将为常规格式的字符串。

String 类为静态方法 Format 提供了好几个重载的版本。其中一个版本除了接受上面的参数外，还接受一个实现了 IFormatProvider 接口的对象，这样我们就可以使用自己指定的语言文化信息来格式化所有的可替换参数了。显然，这时的 Format 会调用每一个对象的 ToString 方法，并传递给它 Format 接受的那个 IFormatProvider 对象。

如果我们使用 StringBuilder、而不是 String 来构造字符串的话，我们可以调用 StringBuilder 的 AppendFormat 方法。该方法除了格式化一个字符串后会将其添加到 StringBuidler 的字符数组中外，其他地方和 String 的 Format 方法工作原理类似。和 String 的 Format 方法一样，AppendString 方法(译注：这里应为 AppendFormat 方法)也接受一个格式字符串，并且也有一个版本还另外接受一个 IFormatProvider 参数。

System.Console 提供的 Write 和 WriteLine 方法也接受格式字符串和可替换参数。但是，它们没有提供重载版本来允许我们传递一个 IFormatProvider。如果我们希望使用某一特定的语言文化来格式化字符串，那么我们必须调用 String 的 Format 方法，并传递给它一个期望的 IFormatProvider 对象，然后再将得到的字符串传递给 Console 类的 Write 或 WriteLine 方法。这不应该成为什么大问题，因为前面曾经说过，客户端代码很少使用一个与调用线程不相关的语言文化来格式化字符串。

12.4.3 提供自定义格式化器

到目前为止，我们应该很清楚.NET 框架中为我们提供的灵活的格式化能力了。但是，事情还没有完。我们还可以定义一个方法使得 StringBuilder 的 AppendFormat 方法在将对象格式化为字符串时能够调用它。换句话说，AppendFormat 不会调用每个对象的 ToString 方法，而是调用我们自己定义的方法，这使得我们能够以任何我们期望的方式来格式化任何一个对象。注意，这里描述的机制只适用于 StringBuilder 的 AppendFormat 方法，String 的 Format 方法不支持这种机制。(译注：这里的说法是不正确的，String 的 Format 方法也支持这里描述的机制。实际上 String 的 Format 方法在内部就是通过调用 StringBuilder 的 AppendFormat 方法来实现的。)

下面通过一个例子来解释这种机制。假设我们正在格式化一段用户可在浏览器上查看的 HTML 文本，我们希望能将其中所有的 Int32 值用粗体显示。为了实现这一点，每当将一个 Int32 值格式化为一个 String 时，我们都希望能在字符串两边加上 HTML 的粗体标记：<B>和</B>。下面的代码演示了一种相当容易的做法：

```
using System;
using System.Text;
using System.Globalization;
using System.Threading;

class BoldInt32s : IFormatProvider, ICustomFormatter {
   public Object GetFormat(Type formatType) {
      if (formatType == typeof(ICustomFormatter)) return this;
      return Thread.CurrentThread.CurrentCulture.GetFormat(formatType);
   }

   public String Format(String format, Object arg,
      IFormatProvider formatProvider) {

      String s;
      IFormattable formattable = arg as IFormattable;

      if (formattable == null) s = arg.ToString();
      else s = formattable.ToString(format, formatProvider);

      if (arg.GetType() == typeof(Int32))
         return "<B>" + s + "</B>";
      return s;
   }
}
```

```
class App {
    static void Main() {

        StringBuilder sb = new StringBuilder();
        sb.AppendFormat(new BoldInt32s(), "{0} {1} {2:M}",
            "Jeff", 123, DateTime.Now);
        Console.WriteLine(sb);
    }
}
```

编译并运行上面的代码，我们将得到以下输出：

```
Jeff <B>123</B> January 23
```

在上面的代码中，Main首先构造了一个空的StringBuilder，然后又在其后面追加了一个经过格式化的字符串。当AppendFormat方法被调用时，其接受的第一个参数为一个BoldInt32s实例。BoldInt32s类实现了前面讨论过的IFormatProvider接口。另外该类还实现了ICustomFormatter接口：

```
public interface ICustomFormatter {
    String Format(String format, Object arg,
        IFormatProvider formatProvider);
}
```

当StringBuilder的AppendFormat需要获取一个对象的字符串表示时，ICustomFormatter接口的Format方法将会被调用。我们可以在该方法中做一些相当巧妙的操作来对字符串格式化进行更多的控制。我们来看一下AppendFormat方法的内部实现。下面的伪码演示了AppendFormat方法的工作原理：

```
public StringBuilder AppendFormat(IFormatProvider formatProvider,
    String format, params Object[] args) {

    // 如果传递了一个IFormatProvider，则找出它提
    // 供的ICustomFormatter对象
    ICustomFormatter cf = null;
    if (formatProvider != null)
        cf = (ICustomFormatter)
            formatProvider.GetFormat(typeof(ICustomFormatter));

    // 将添加的文本常量字符(这段伪码中没有显示)和可
    // 替换参数保存在StringBuilder的字符数组中
    while (MoreReplaceableArgumentsToAppend) {
        // argFormat指向由format参数获得的可替换
        // 格式字符串
        String argFormat =…;

        // argObj指向args数组参数中的对应元素
        Object argObj =…;
```

```
        // argStr 将指向最终添加到结果字符串中的经过
        // 格式化的字符串
        String argStr = null;

        // 如果自定义格式化器可用，那么让它格式化传入
        // 的参数
        if (cf != null)
            argStr = cf.Format(argFormat, argObj, formatProvider);

        // 如果没有自定义格式化器，或者它没有格式化
        // 传入的参数，那么试用别的做法
        if (argStr == null) {

            // 参数的类型是否支持 IFormattable 接口?
            IFormattable formattable = argObj 入as IFormattable;
            if (formattable != null) {
                // 支持：传递格式字符串和提供者给类
                // 型的 IFormattable.ToString 方法
                argStr = formattable.ToString(argFormat, formatProvider);
            }else {
                // 不支持：使用调用线程的语言文化信
                // 息来获得字符串的常规格式
                if (argObj!= null) argStr = argObj.ToString();
                else argStr = String.Empty;
            }
        }
        // 将 argStr 中的字符添加到 StringBuilder 的字符数组上
        ...
    }
    return this;
}
```

当Main调用AppendFormat时，它会调用格式化提供者的GetFormat方法，并传入ICustomFormatter类型。BoldInt32s中定义的GetFormat方法发现传入的为ICustomFormatter类型，它便会返回一个自身对象的引用。如果传递给 GetFormat 方法的是其他类型，那么它就使用和调用线程相关联的CultureInfo对象来调用GetFormat方法。

不管AppendFormat何时需要格式化一个可替换参数，它都会调用ICustomFormatter的Format方法。在上面的例子中，它就会调用BoldInt32s中定义的Format方法。该Format方法首先检查被格式化的对象是否支持IFormattable接口。如果不支持，那么调用最简单的无参ToString方法来格式化对象；如果支持，那么调用具有两个参数的ToString方法、并为其传入格式字符串和格式提供者。

在得到了经过格式化的字符串后，Format接着便检查被格式化的对象是否为Int32类型，如果是，它便将字符串封装在HTML标记<B>和</B>中，然后返回；如果不是，那么它将不做任何处理而直接返回。

12.5 通过解析字符串获取对象

上一节向大家解释了怎样由一个对象获取它的字符串表达形式。本节讨论相反的话题：怎样通过解析一个字符串来获取一个对象。虽然这样的操作并不常见，但是偶尔也会派上用场。微软认为有必要将这种机制正式化。

任何能够解析一个字符串的类型都提供有一个名为 Parse 的公有静态方法。该方法接受一个 String，并返回一个类型的实例。从某种角度来看，Parse 的行为有点像一个构造器。在 FCL 中，Parse 方法存在于所有的数值类型、DateTime、TimeSpan 以及其他几个类型(如 SQL 数据类型)中。

我们来看一下怎样将一个字符串解析为一个数值类型。所有的数值类型(Byte、SByte、Int16/UInt16、Int32/UInt32、Int64/UInt64、Single、Double、和 Decimal)都提供有至少一个 Parse 方法。下面向大家展示的是 Int32 类型中定义的 Parse 方法。(其他数值类型的 Parse 方法与此是一样的。)

```
public static Int32 Parse(String s, NumberStyles style,
    IFormatProvider provider);
```

单从方法原型上来看，我们应该能够猜出该方法的工作原理。String 参数(s)表示我们要解析的字符串。System.Globalization.NumberStyles 参数(style)是一个位标记枚举类型，它表示可能在字符串中出现的字符形式。IFormatProvider 参数(provider)表示一个 Parse 方法可以用来获取特定语言文化信息的对象。

下面的代码将导致 Parse 方法抛出一个 System.FormatException 异常，因为被解析的字符串前面包含一个空格：

```
Int32 x = Int32.Parse(" 123", NumberStyles.None, null);
```

要使 Parse 方法忽略前面的空格，我们应该改变 style 参数：

```
Int32 x = Int32.Parse(" 123", NumberStyles.AllowLeadingWhite, null);
```

表 12.5 显示了 NumberStyles 类型定义的位符号。

表 12.5 NumberStyles 类型定义的位符号

符号	值	描述
None	0x00000000	字符串中不允许出现表中其余各行的任何位表示的特殊字符
AllowLeadingWhite AllowTrailingWhite	0x00000001 0x00000002	可以在字符串的前导/结尾处包含空白字符(由下面的 Unicode 码点标识：0x0009、0x000A、0x000B、0x000C、0x000D 和 0x0020)
AllowLeadingSign AllowTrailingSign	0x00000004 0x00000008	字符串中可以包含一个有效的前导/结尾符号字符。NumberFormatInfo 的 PositiveSign 和 NegativeSign 属性决定了有效的前导符号字符
AllowParentheses	0x00000010	字符串中可以包含小括号
AllowDecimalPoint	0x00000020	字符串中可以包含一个有效的十进制分隔符。NumberFormatInfo 的 NumberDecimalSeparator 和 CurrencyDecimalSeparator 属性决定了有效的十进制分隔符
AllowThousands	0x00000040	字符串中可以包含一个有效的分组符。NumberFormatInfo 的 NumberGroupSeparator 和 CurrencyGroupSeparator 属性决定了有效的分组符。NumberFormatInfo 的 NumberGroupSize 和 CurrencyGroupSize 属性决定了分组中的数字个数
AllowExponent	0x00000080	字符串中可以包含一个用指数格式表达的数： {e|E} [{+|-}] n 其中 n 为一个数字

续表

符　号	值	描　述
AllowCurrencySymbol	0x00000100	字符串中可以包含一个有效的货币符号。NumberFormatInfo 的 CurrencySymbol 属性决定了有效的货币符号
AllowHexSpecifier	0x00000200	字符串中可以包含十六进制数字(0~9、A~F)，并且字符串将被认为是一个十六进制的数值

除了表 12.5 中列出的位符号外，NumberStyles 枚举类型还定义了几种符号来表示一些常用的单个位的组合。表 12.6 列出了它们。

表 12.6　NumberStyles 类型定义的位组合符号

符　号	位集合
Integer	AllowLeadingWhite、AllowTrailingWhite、AllowLeadingSign 三者之间取或
Number	AllowLeadingWhite、AllowTrailingWhite、AllowLeadingSign、AllowTrailingSign、AllowDecimalPoint、AllowThousands 六者之间取或
Float	AllowLeadingWhite、AllowTrailingWhite、AllowLeadingSign、AllowDecimalPoint、AllowExponent 五者之间取或
Currency	AllowLeadingWhite、AllowTrailingWhite、AllowLeadingSign、AllowTrailingSign、AllowParentheses、AllowDecimalPoint、AllowThousands、AllowCurrencySymbol 八者之间取或
HexNumber	AllowLeadingWhite、AllowTrailingWhite、AllowHexSpecifier 三者之间取或
Any	AllowLeadingWhite、AllowTrailingWhite、AllowLeadingSign、AllowTrailingSign、AllowParentheses、AllowDecimalPoint、AllowThousands、AllowCurrencySymbol、AllowExponent 九者之间取或

下面一段代码展示了如何解析一个十六进制的数：

```
Int32 x = Int32.Parse("1A", NumberStyles.HexNumber, null);
Console.WriteLine(x);     // 显示 "26"
```

上面的 Parse 方法总共接受三个参数。为方便起见，很多类型都提供了几个 Parse 的重载版本以减少我们传递不必要的参数。例如 Int32 就提供了以下四个重载的 Parse 方法：

```
// 传递 NumberStyles.Integer 给 style 参
// 数，传递 null 给 provider 参数
public static Int32 Parse(String s);

// 传递 null 给 provider 参数
public static Int32 Parse(String s, NumberStyles style);

// 传递 NumberStyles.Integer 给 style 参数
public static Int32 Parse(String s, IFormatProvider provider)

// 该方法前面已经讨论过
public static int Parse(String s, NumberStyles style,
    IFormatProvider provider);
```

另外，DateTime 类型也提供了一个类似的 Parse 方法：

```
public static DateTime Parse(String s,
    IFormatProvider provider, DateTimeStyles styles);
```

该方法除了接受 DateTimeStyles 枚举类型、而不是 NumberStyles 定义的位标记集合外，其工作原理和数值类型定义的 Parse 方法是一样的。表 12.7 展示了 DateTimeStyles 类型定义的位符号。

表 12.7 DateTimeStyles 类型定义的位符号

符 号	值	描 述
None	0x00000000	字符串中不允许出现表中其余各行的任何位表示的特殊字符
AllowLeadingWhite AllowTrailingWhite AllowInnerWhite	0x00000001 0x00000002 0x00000004	可以在字符串的前导/中间/结尾处包含空白字符(由下面的 Unicode 码点标识：0x0009、0x000A、0x000B、0x000C、0x000D 和 0x0020)
NoCurrentDateDefault	0x00000008	在解析一个包含时间(不带日期)的字符串时，将日期设为 0001 年 1 月 1 日，而不是当前的日期
AdjustToUniversal	0x00000010	在解析一个包含时区指示符("GMT"、"Z"、"+xxxx"、"-xxxx")的字符串时，将解析得到的时间调整为格林尼治标准时间(Greenwich Mean Time)

除了上述这些位符号外，DateTimeStyles 枚举类型还定义了一个 AllowWhiteSpaces 符号，它等同于所有的空白符号取或(AllowLeadingWhite | AllowInnerWhite | AllowTrailingWhite)。

同样处于方便起见，DateTime 类型也定义了三个重载的 Parse 方法：

```
// 传递null给format，传递DateTimeStyles.None给styles
public static DateTime Parse(String s);

// 传递DateTimeStyles.None给styles
public static DateTime Parse(String s, IFormatProvider provider);

// 该方法前面已经讨论过
public static DateTime Parse(String s,
   IFormatProvider provider, DateTimeStyles styles);
```

日期和时间的解析通常是比较复杂的。许多开发人员都发现 DateTime.Parse 的解析工作做的太过宽松，以致有时候它可以解析不含日期或时间的字符串。出于这种原因，DateTime 类型又提供了一个 ParseExact 方法，该方法接受一个精确表示日期/时间字符串怎样被格式化以及应该怎样被解析的图片格式字符串。关于图片格式字符串的更多信息，请参见.NET 框架 SDK 中的 DateTimeFormatInfo 类。

12.6　编码：字符与字节之间的转换

在 Win32 环境下，开发人员通常需要自己手动编程来将 Unicode 编码的字符和字符串转换为多字节字符集(Multibyte Character Set，简称 MBCS)编码的字符和字符串。我个人也编写过这样的代码，这种工作非常单调，而且很容易出错。在 CLR 中，所有的字符都被表示为 16 位的 Unicode 码值，并且所有的字符串都由 16 位的 Unicode 码值组成。这使得我们在运行时操作字符和字符串变得非常容易。

然而有时候我们可能希望将字符串保存到一个文件中、或者把它们放在网络上进行传输。如果字符串中包含的大多数字符都是英文字符，那么这些 16 位值的保存或者传输的效率将不会很高，因为近一半的字节包含的都是 0 值。但是，如果我们能将这些 16 位的值编码到一个压缩的字节数组中，然后再将它们解码到一个 16 位值的数组中，那么整个工作的效率将会有很大的提升。

编码还可以使得一个托管应用程序创建的字符串与非 Unicode 系统创建的字符串进行交互。例如，如果我们希望产生一个可被 Windows 95 日文版上的应用程序读取的文件，我们就必须使用 Shift-JIS(代码页 932)编码来保存 Unicode 文本。(译注：代码页是一个包括字符、数字、标点、符号和特殊字符组成的字符集合。不同的区域、不同的语言使用不同的代码页，代码页后的数字表示代码页的 ID 号。如中文 GB 内码的代码页 ID 号为 936，繁体中文 Big5 的代码页 ID 为 950。)类似地，我们也应该使用 Shift-JIS 编码来将一个由 Windows 95 日文版系统上产生的文本文件读取到 CLR 中。

当我们使用 System.IO.BinaryWriter 或者 System.IO.StreamWriter 将一个字符串发送到一个文件或者网络流中时，我们往往需要进行编码操作。类似地，当我们使用 System.IO.BinaryReader 或者 System.IO.StreamReader 来从一个文件或者网络流中读取一个字符串时，我们一般需要进行解码操作。如果我们没有显式选择一种编码方式，那么所有这些类型将默认使用 UTF-8 编码(UTF 表示 Unicode 转换格式，即 Unicode Transformation Format)。但是，有时候我们可能需要自己对一个字符串进行编码或解码。

幸运的是，FCL 提供了一些类型允许我们方便地对字符进行编码和解码。两种最常用的编码方式为 UTF-16 和 UTF-8。

- UTF-16 编码将 16 位的字符编码为 2 个字节。这根本不会影响字符，也不存在任何压缩——这种编码的性能自然很好。UTF-16 编码又称 Unicode 编码。另外，UTF-16 编码还可以用于在低位优先(little endian)的码值和高位优先(big endian)的码值之间进行转换。(译注：低位优先是指将低位字节放在存储区的低地址，将高位字节放在存储区的高地址；而高位优先是指将高位字节放在存储区的低地址，将低位字节放在存储区的高地址。)
- 一个字符经 UTF-8 编码后有四种可能的结果，分别为 1 个字节、2 个字节、3 个字节和 4 个字节。具体来说，码值低于 0x0080 的字符会被压缩为 1 个字节，这对于美国地区使用的字符来说非常合适。码值在 0x0080 和 0x07FF 之间的字符会被转换为 2 个字节，这适用于欧洲和中东地区的语言。码值在 0x0800 及其以上的字符会被转换为 3 个字节，这适用于东亚地区的语言。最后，代理字符对(surrogate character pairs)会被转换为 4 个字节。UTF-8 是一种相当流行的编码方式，但是如果我们希望对许多码值在 0x0800 及其以上的字符进行编码的话，UTF-16 要比它更有用一些。

虽然 UTF-16 和 UTF-8 编码是目前最为通用的编码方式，但 FCL 还支持一些不常用的编码：

- UTF-7 编码典型地用于处理一些用 7 位码值来表示字符的老式系统。我们应该避免使用这种编码方式，因为它通常会导致数据的膨胀、而不是压缩。Unicode 联盟从 Unicode 3.0 标准开始就已经废弃了这种编码方式。
- ASCII 编码将 16 位字符编码为 ASCII 字符。也就是说，任何码值小于 0x0080 的 16 位字符都将被转换为一个字节。这种编码不能转换任何码值大于 0x007F 的字符，否则字符的值将会出现丢失。对于包含 ASCII 字符(从 0x00 到 0x7F)的字符串来说，这种编码会将数据压缩一半，并且效率也很高(因为可以直接去掉高位字节)。但如果我们的字符超出了 ASCII 字符的码值范围，这种编码将不再适用，因为它会丢失一些字符的码值。

另外，FCL还允许我们将16位的字符编码到任意一个代码页中。和ASCII编码类似，将16位的字符编码到一个代码页中有一定的危险性，因为那些码值不能在指定的代码页中表示的字符会被丢失。除非必须处理某些使用其他编码的遗留文件或者应用程序，一般情况下我们应该尽量使用UTF-16或者UTF-8编码。

当我们需要对一组字符进行编码或者解码时，我们应该获取一个类型继承自System.Text.Encoding的实例。Encoding是一个抽象类，它提供有几个静态属性，每个属性都返回一个类型继承自Encoding的实例。(每个Encoding类基本上都是对大家熟悉的两个Win32函数WideCharToMultiByte和MultiByteToWideChar的一种封装。)

下面的例子使用了UTF-8来对字符进行编码和解码：

```
using System;
using System.Text;

class App {
   static void Main() {
      // 下面为将要进行编码的字符串
      String s = "Hi there.";

      // 获得一个类型继承自Encoding的对象，该对
      // 象知道怎样使用UTF-8进行编码和解码
      Encoding encodingUTF8 = System.Text.Encoding.UTF8;

      // 将字符串编码为一个字节数组
      Byte[] encodedBytes = encodingUTF8.GetBytes(s);

      // 显示编码后的字节值
      Console.WriteLine("Encoded bytes: " +
         BitConverter.ToString(encodedBytes));

      // 将字节数组解码为一个字符串
      String decodedString = encodingUTF8.GetString(encodedBytes);

      // 显示解码后的字符串
      Console.WriteLine("Decoded string: " + decodedString);
   }
}
```

编译并运行上面的代码，我们将得到以下输出：

```
Encoded bytes: 48-69-20-74-68-65-72-65-2E
Decoded string: Hi there.
```

除了UTF8静态属性外，Encoding类还提供有以下几个静态属性：Unicode、BigEndianUnicode、UTF7、ASCII和Default。

其中 Default 属性返回的对象知道怎样使用用户在 Windows 控制面板【区域和语言选项】中指定的代码页来进行编码或解码。(可参见 Win32 函数 GetACP 来获得更多的信息。)但是，一般情况下建议大家不要使用 Default 属性。

除了上述这些属性外，Encoding 类还提供有一个静态方法 GetEncoding，该方法允许我们指定一个代码页(通过一个整数或者字符串)，并返回一个可以使用该代码页来进行编码或解码的对象。例如，我们可以传递"Shift-JIS"或者 932 来调用 GetEncoding 方法。

当我们第一次请求一个编码对象时，Encoding 类的属性或 GetEncoding 方法会构造一个所请求的编码对象，然后返回该对象。如果以后再次请求已经请求过的对象，那么 Encoding 类将简单地返回先前已构造好的对象，而不会为每一次请求都构造新的对象。这既减少了系统中对象的数量，也减轻了垃圾收集器的压力。

除了调用 Encoding 类的静态属性或 GetEncoding 方法外，我们还可以构造下面几个类的实例：System.Text.UnicodeEncoding、System.Text.UTF8Encoding、System.Text.UTF7Encoding、或者 System.Text.ASCIIEncoding。但是，我们要清楚每一次构造这些类的实例都会在托管堆中创建新的对象，这会对系统的性能有负面影响。

UnicodeEncoding、UTF8Encoding 和 UTF7Encoding 三个类都提供有多个构造器，这使得我们可以对编码和字节顺序标记(即 byte order marks，简称 BOMs)有更多的控制。在使用 BinaryWriter 或者 StreamWriter 时，我们可能希望显式构造这些编码类型的实例。而 ASCIIEncoding 类只有一个构造器，因此我们不能用它来对编码进行更多的控制。如果我们需要一个 ASCIIEncoding 对象，我们应该通过查询 Encoding 类的 ASCII 属性来获得，而不要自己构造 ASCIIEncoding 类的实例。

一旦我们有了一个类型继承自 Encoding 的对象，我们就可以通过调用 GetBytes 方法来将一个字符数组转换为一个字节数组。(该方法有几个重载版本。)我们也可以通过调用 GetChars 方法、或者更为有用的 GetString 方法来将一个字节数组转换为一个字符数组。(两个方法也都有几个重载版本。)前面的例子曾经向大家演示过 GetBytes 和 GetString 这两个方法。

所有继承自 Encoding 的类型都提供了一个 GetByteCount 方法，该方法不用执行实际的编码操作便可以返回对一组字符进行编码所需要的字节个数，该方法的用处不是很大。我们可以使用该方法来分配字节数组。同样，也有一个 GetCharCount 方法，它不用执行实际的解码操作便可以返回对一组字节进行解码所需要的字符个数。如果我们希望能节省内存、并重用数组的话，那么这些方法还是有些用处的。

GetByteCount 方法和 GetCharCount 方法的效率不是很高，因为它们必须通过分析字符或字节数组来得到精确的结果。如果我们更在意的是速度、而不是一个精确的结果，我们可以调用 GetMaxByteCount 和 GetMaxCharCount 两个方法。这两个方法都接受一个整数(字符或者字节的数量)，并返回一个最坏情况下的数值。

每个类型继承自 Encoding 的对象都提供了一组公有只读属性，我们可以通过查询它们来获得有关某种编码的详细信息。表 12.8 简要描述了这些属性。

表 12.8 继承自 Encoding 类的属性

属 性	类 型	描 述
EncodingName	String	返回此编码的可读名称
CodePage	Int32	返回此编码的代码页
WindowsCodePage	Int32	返回与此编码最贴近的 Windows 代码页
WebName	String	返回此编码在 Internet 编号分配管理机构(Internet Assigned Numbers Authority，简称 IANA)注册的名称。更多的信息可流览 http://www.iana.org
HeaderName	String	返回邮件代理头部标记
BodyName	String	返回邮件代理正文标记
IsBrowserDisplay	Boolean	如果流览器客户端可以用此编码来进行显示，那么返回 true
IsBrowserSave	Boolean	如果流览器客户端可以用此编码来进行保存，那么返回 true
IsMailNewsDisplay	Boolean	如果邮件和新闻客户端可以用此编码来进行显示，那么返回 true
IsMailNewsSave	Boolean	如果邮件和新闻客户端可以用此编码来进行保存，那么返回 true

下面的程序显示了几种不同编码的属性，以及它们的含义：

```
using System;
using System.Text;

class App {
    static void Main() {
        Show(Encoding.Unicode);
        Show(Encoding.BigEndianUnicode);
```

```
        Show(Encoding.UTF8);
        Show(Encoding.UTF7);
        Show(Encoding.ASCII);
        Show(Encoding.Default);
        Show(Encoding.GetEncoding(0)); // 与Default相同
        Console.WriteLine();
        Console.WriteLine("Below are some specific code pages:");
        Show(Encoding.GetEncoding(437));
        Show(Encoding.GetEncoding(28595));
        Show(Encoding.GetEncoding(57008));
        Show(Encoding.GetEncoding(54936));
        Show(Encoding.GetEncoding(874));
    }
    static void Show(Encoding e) {
        Console.WriteLine("{1}{0}" +
            "\tCodePage={2}, WindowsCodePage={3}{0}" +
            "\tWebName={4}, HeaderName={5}, BodyName={6}{0}" +
            "\tIsBrowserDisplay={7}, IsBrowserSave={8}{0}" +
            "\tIsMailNewsDisplay={9}, IsMailNewsSave={10}{0}",
            Environment.NewLine, e.EncodingName, e.CodePage,
            e.WindowsCodePage, e.WebName, e.HeaderName,
            e.BodyName, e.IsBrowserDisplay, e.IsBrowserSave,
            e.IsMailNewsDisplay, e.IsMailNewsSave);
    }
}
```

编译并运行上面的代码，我们将得到以下输出(译注：根据读者操作系统的配置，执行该段代码的输出可能会与下面的内容有所不同，甚至有可能因为不支持某些代码页而抛出异常)：

```
Unicode
    CodePage=1200, WindowsCodePage=1200
    WebName=utf-16, HeaderName=utf-16, BodyName=utf-16
    IsBrowserDisplay=False, IsBrowserSave=True
    IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode (Big-Endian)
    CodePage=1201, WindowsCodePage=1200
    WebName=unicodeFFFE, HeaderName=unicodeFFFE, BodyName=unicodeFFFE
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=False, IsMailNewsSave=False

Unicode (UTF-8)
    CodePage=65001, WindowsCodePage=1200
    WebName=utf-8, HeaderName=utf-8, BodyName=utf-8
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True
```

```
Unicode (UTF-7)
    CodePage=65000, WindowsCodePage=1200
    WebName=utf-7, HeaderName=utf-7, BodyName=utf-7
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=True, IsMailNewsSave=True

US-ASCII
    CodePage=20127, WindowsCodePage=1252
    WebName=us-ascii, HeaderName=us-ascii, BodyName=us-ascii
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=True, IsMailNewsSave=True

Western European (Windows)
    CodePage=1252, WindowsCodePage=1252
    WebName=Windows-1252, HeaderName=Windows-1252, BodyName=iso-8859-1
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True

Western European (Windows)
    CodePage=1252, WindowsCodePage=1252
    WebName=Windows-1252, HeaderName=Windows-1252, BodyName=iso-8859-1
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True

Below are some specific code pages:
OEM United States
    CodePage=437, WindowsCodePage=1252
    WebName=IBM437, HeaderName=IBM437, BodyName=IBM437
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=False, IsMailNewsSave=False

Cyrillic (ISO)
    CodePage=28595, WindowsCodePage=1251
    WebName=iso-8859-5, HeaderName=iso-8859-5, BodyName=iso-8859-5
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True

ISCII Kannada
    CodePage=57008, WindowsCodePage=57008
    WebName=x-iscii-ka, HeaderName=x-iscii-ka, BodyName=x-iscii-ka
    IsBrowserDisplay=False, IsBrowserSave=False
    IsMailNewsDisplay=False, IsMailNewsSave=False

Chinese Simplified (GB18030)
    CodePage=54936, WindowsCodePage=936
    WebName=GB18030, HeaderName=GB18030, BodyName=GB18030
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True
```

```
Thai (Windows)
    CodePage=874, WindowsCodePage=874
    WebName=windows-874, HeaderName=windows-874, BodyName=windows-874
    IsBrowserDisplay=True, IsBrowserSave=True
    IsMailNewsDisplay=True, IsMailNewsSave=True
```

表 12.9 对所有继承自 Encoding 的类的方法做了一个总结。

表 12.9 继承自 Encoding 的类的方法

方 法	描 述
GetPreamble	返回一个字节数组，该数组表示在写入任何编码字节之前应该先写入一个流的内容。这些字节通常被称为字节顺序标记(BOM)字节。当我们开始从一个流中读取字节时，BOM 字节自动帮助我们检测在流被写入时使用的是何种编码，这样在读取字节时才能使用正确的解码器。对于大多数继承自 Encoding 的类来说，该方法返回一个 0 长的字节数组——也就是说没有前导字节(preamble bytes)。显式构造一个 UTF8Encoding 对象并调用该方法将返回一个元素为 0xEF、0xBB、0xBF 的 3 字节数组。显式构造一个 UnicodeEncoding 对象并调用该方法，如果是高位优先的编码，方法将返回一个元素为 0xFE、0xFF 的 2 字节数组；如果是低位优先的编码，方法将返回一个元素为 0xFF、0xFE 的 2 字节数组
Convert	将一个源编码指定的字节数组转换为一个目的编码指定的字节数组。该静态方法在内部会调用源编码对象的 GetChars 方法，并将结果传递给目的编码对象的 GetBytes 方法，最后将得到的字节数组返回
Equals	如果两个类型继承自 Encoding 的对象表示同样的代码页和前导字节设置，那么该方法将返回 true
GetHashCode	返回编码对象的代码页(译注：这是大多数编码类的 GetHashCode 方法目前内部的实现，即返回 CodePage 属性。但个别类——如 UTF8Encoding——的 GetHashCode 返回的值会对 CodePage 属性进行一个修正)

12.6.1 字符与字节的编码/解码流

假设我们正在通过一个 System.Net.Sockets.NetworkStream 对象读取一个 UTF-16 编码的字符串，其中的字节将很有可能以成块的数据流进来。换句话说，我们可能首先从流中读取 5 个字节，然后再读取 7 个字节。但是，在 UTF-16 中，每个字符包含的是 2 个字节。

所以如果调用 Encoding 的 GetString 方法、并传递第一个 5 字节的数组，我们将得到一个只包含 2 个字符的字符串。如果再调用 GetString 方法、并传递第二个 7 字节的数组，我们将得到一个包含 3 个被错误编码的字符的字符串。

出现这种数据损坏的问题是因为继承自 Encoding 的类不会保存方法调用之间的状态。如果我们正在对成块的字符/字节进行编码或解码，我们必须做一些额外的工作以保存方法调用之间的状态，从而防止数据丢失。

要对成块的字节进行解码，我们应该获得一个类型继承自 Encoding 的对象(就像前一节描述的那样)，然后调用它的 GetDecoder 方法。该方法返回一个新构造的、类型继承自 System.Text.Decoder 的对象。和 Encoding 类相似，Decoder 类也是一个抽象基类。如果我们查看.NET 框架 SDK 文档，我们将看不到任何继承自 Decoder 的具体实现类。但是，FCL 确实定义了许多继承自 Decoder 的类，例如 UTF8Decoder。虽然这些类都是 FCL 的内部类，但是 GetDecoder 方法可以构造这些类的实例，并将它们返回给我们的应用程序代码。

所有继承自 Decoder 的类都提供了两个方法：GetChars 和 GetCharCount。很明显，这些方法用于解码工作，并且其原理和前面讨论过的 Encoding 的 GetChars 和 GetCharCount 方法类似。当我们调用其中任何一个方法时，它会对字节数组中尽可能多的字节进行解码。如果字节数组中剩余的字节没能包含足够填充一个字符的字节数据，那么剩余的字节将被保存在解码器对象中。在下一次调用这些方法时，解码器对象会使用剩余的字节、以及新传递给它的字节数组——这就确保了成块的数据可以被正确地解码。当我们从一个流中读取字节时，Decoder 对象非常有用。

一个继承自 Encoding 的类型可以同时用于无状态的编码和解码。但是一个继承自 Decoder 的类型则只能用于解码。如果我们希望对字符串进行成块的编码，那么我们应该调用 Encoding 对象的 GetEncoder(而不是 GetDecoder)方法。GetEncoder 方法也会返回一个新构造的、类型继承自 System.Text.Encoder(也是一个抽象基类)的对象。同样，.NET 框架 SDK 文档中也不包含任何继承自 Encoder 的具体实现类。但是 FCL 确实也定义了一些继承自 Encoder 的类，例如 UTF8Encoder。和继承自 Decoder 的类一样，这些类也都是 FCL 的内部类，但是 GetEncoder 方法可以构造这些类的实例，并将它们返回给我们的应用程序代码。类似地，所有继承自 Encoder 的类都提供了两个方法：GetBytes 和 GetByteCount。每当我们调用它们的时候，编码器对象都会保存着任何剩余的状态信息，从而使得我们可以对数据进行成块的编码。

12.6.2 Base-64 字符串编码与解码

目前，UTF-16 和 UTF-8 编码正在变得非常流行。而将一个字节序列编码到一个 base-64 字符串中也正在获得关注。FCL 确实提供了一些方法来允许我们进行 base-64 的编码和解码。大家可能会认为这可以通过继承自 Encoding 的类型来实现。但是，处于某种原因，base-64 的编码和解码是通过 System.Convert 类型提供的一些静态方法来实现的。

要将一个 base-64 的字符串编码为一个字节数组，我们可以调用 Convert 的静态方法 FromBase64String 或者 FromBase64CharArray。类似地，要将一个字节数组解码为一个 base-64 的字符串，我们可以调用 Convert 的静态方法 ToBase64String 或者 ToBase64CharArray。下面的代码演示了它们的用法：

```
using System;

class App {
    static void Main() {
        // 获得一组 10 个随机产生的字节
        Byte[] bytes = new Byte[10];
        new Random().NextBytes(bytes);

        // 显示产生的字节
        Console.WriteLine(BitConverter.ToString(bytes));

        // 将字节数组解码为一个 base-64 的字符串，
        // 并将其显示在控制台上
        String s = Convert.ToBase64String(bytes);
        Console.WriteLine(s);

        // 将 base-64 的字符串编码为一个字节数组，
        // 并将其显示在控制台上
        bytes = Convert.FromBase64String(s);
        Console.WriteLine(BitConverter.ToString(bytes));
    }
}
```

编译并运行上面的代码，我们将得到以下输出。(因为随机产生字节的原因，大家的输出结果可能会与此不同。)

```
34-24-99-7A-66-BF-D1-5F-41-1C
NCSZema/0V9BHA==
34-24-99-7A-66-BF-D1-5F-41-1C
```

13

枚举类型与位标记

本章讨论枚举类型与位标记。这些构造在 Windows 中的应用已经有很多年了，相信很多人对它们都比较熟悉。但是直到 CLR 和 FCL 的出现，枚举类型与位标记才成为真正的面向对象的类型，这其中蕴涵了很多很酷的特性(feature)，估计大多数开发人员对它们还都不太熟悉。我个人也惊讶于这些特性竟能使应用程序的开发工作变得如此方便，这些特性将是本章讨论的重点。

13.1 枚举类型

枚举类型定义了一组符号名称和数值对。例如，下面演示的 Color 类型就定义了一组符号，其中每一个符号标识一种颜色：

```
enum Color {
    Red,          // 赋为 0 值
    Green,        // 赋为 1 值
    Blue,         // 赋为 2 值
    Orange        // 赋为 3 值
}
```

当然，我们在编写程序时总可以用 0 来表示 Red，1 来表示 Green，等等。但是，我们不应该将这些数字硬编码(hard-code)到程序代码中，相反我们应该采用枚举类型，这至少有以下两个原因：

- 枚举类型使得程序更容易编写、阅读和维护。使用枚举类型，我们可以在整个代码中采用符号名称，而不必时刻提醒自己0和“红色”的对应关系。另外，如果符号的数值需要改变，源代码将无需做任何改变，只需简单地重新编译即可。其次，我们可以使用一些文档工具和实用程序(例如调试器)来显示一些有意义的符号名称。

- 枚举类型是强类型的。例如，如果我们试图将 Color.Orange 传递给一个接受 Fruit 枚举类型参数的方法，编译器将会报告错误。

在 CLR 中，枚举类型不仅只是编译器所关心的一些符号。枚举类型在整个类型系统中被认为是“第一等的公民”，它允许我们进行一些非常强大的操作，而这在其他一些环境中是不可能的(例如非托管 C++)。

每个枚举类型都直接继承自 System.Enum，System.Enum 又继承自 System.ValueType，System.ValueType 最后继承自 System.Object。因此，枚举类型属于值类型(第 5 章中对值类型有详细的描述)，并且可以用未装箱和已装箱两种形式表示。但是和其他值类型不同的是，枚举类型不能定义任何的方法、属性或事件。

当编译一个枚举类型时，C#编译器会将其中的每个符号转变为类型的一个常数字段。例如，编译器对待前面的 Color 枚举类型就像我们编写了下面的代码一样：

```
struct Color : System.Enum {
    public const Color Red = (Color) 0;
    public const Color Green = (Color) 1;
    public const Color Blue = (Color) 2;
    public const Color Orange = (Color) 3;
}
```

C#编译器并不会编译上面的代码，但是这个例子却使得我们对枚举类型的内部机理有了一个认识。基本上来讲，枚举类型就是一个定义了一组常数字段的结构而已。这些字段编译后会被存放在生成模块的元数据中，并且可以通过反射来访问。这意味着我们可以在运行时通过反射来得到和枚举类型相关联的所有符号以及它们的数值。这还意味着我们可以将一个字符串符号转换为它所对应的数值。

重要 枚举类型定义的符号是一些常数值。这意味着在编译时，编译器会将代码中引用的枚举类型符号转换为一个数值。一旦出现这种情况，代码编译后生成的元数据中将没有枚举类型的引用、定义该枚举类型的程序集在运行时也不必存在。如果我们的代码中引用了枚举类型本身，而非仅仅引用它们所定义的一些符号，那么在运行时仍然需要定义该枚举类型的程序集。由于枚举类型符号是一些常数，而非只读数值，所以它们会带来一些版本问题，本书第 8 章对此有详细的讨论。

为了简化这些操作，并使我们不必熟悉反射就可以进行这些操作，System.Enum 类型提供了一些静态方法和实例方法。这些方法的操作对象一般为一些枚举类型的实例。下面我们来讨论其中的一些方法。

例如，Enum 类型有一个名为 GetUnderlyingType 的静态方法：

```
static Type GetUnderlyingType(Type enumType);
```

该方法返回用于保存枚举类型实例值的基础类型(underlying type)。每个枚举类型都有一个基础类型，它们可以是 byte、sbyte、short、ushort、int(最常用的类型，也是 C#选用的默认值)、uint、long 或者 ulong。当然，这些 C#基元类型和它们对应的 FCL 类型是等同的。但是，当我们为枚举类型指定基础类型时，C#编译器要求我们只能使用基元类型，使用 FCL 类型(例如 Int32)将会导致编译器报错。下面的代码演示了怎样使用 C#声明一个基础类型为 byte(System.Byte)的枚举类型：

```
enum Color : byte {
    Red,
    Green,
    Blue,
    Orange
}
```

使用上面定义的 Color 枚举类型，下面的代码演示了 GetUnderlyingType 返回的结果：

```
// 下面的代码将显示 "System.Byte"
Console.WriteLine(Enum.GetUnderlyingType(typeof(Color)));
```

给定一个枚举类型实例，我们可以用 System.Enum.ToString 方法将其数值映射为 4 种字符串表达形式：

```
Color c = Color.Blue;
Console.WriteLine(c.ToString());           // "Blue" (常规格式)
Console.WriteLine(c.ToString("G"));        // "Blue" (常规格式)
Console.WriteLine(c.ToString("D"));        // "2" (十进制格式)
Console.WriteLine(c.ToString("X"));        // "02" (十六进制格式)
```

ToString 方法在内部调用了 System.Enum 的静态方法 Format：

```
public static String Format(Type enumType,
    Object value, String format);
```

一般情况下，ToString 方法更好一些，因为它需要的代码较少，并且更容易调用。但是使用 Format 方法有一个好处是，它允许我们传递枚举值的数值形式，而不必非要使用枚举类型的实例。例如，下面的代码将显示 "Blue"(译注：要运行下面的代码，读者应该使用基础类型为 int 的那个 Color 枚举定义，否则代码会抛出 System.ArgumentException 异常，这是因为代码中的常数 2 被认为是 int 类型)：

```
// 下面一行将显示 "Blue"
Console.WriteLine(Enum.Format(typeof(Color), 2, "G"));
```

> **注意** 我们可以在定义一个枚举类型的时候，让其中的多个符号有着相同的数值。但当将一个数值转换为枚举类型符号时，Enum 的方法只返回其中的一个符号，具体返回哪一个并不确定。另外，如果不存在与传入的数值对应的符号，方法将返回一个包含数值的字符串。

下面再来看一个 System.Enum 类型的静态方法 GetValues，该方法允许我们获取一个枚举类型中定义的所有符号。

```
static Array GetValues(Type enumType);
```

将该方法和 ToString 一起使用，我们可以显示一个枚举类型中定义的所有符号名称以及它们对应的数值：

```
Color[] colors = (Color[]) Enum.GetValues(typeof(Color));
Console.WriteLine("Number of symbols defined: " + colors.Length);
Console.WriteLine("Value\tSymbol\n-----\t------");
foreach (Color c in colors) {
    // 以十进制格式和常规格式显示每一个符号
    Console.WriteLine("{0,5:D}\t{0:G}", c);
}
```

上面的代码将产生以下输出：

```
Number of symbols defined: 4
Value    Symbol
-----    ------
    0    Red
    1    Green
    2    Blue
    3    Orange
```

上面的讨论演示了一些可以在枚举类型上执行的很酷的操作。其中常规格式的 ToString 方法经常在程序的用户界面中用来显示枚举实例的符号名称，前提是这些符号字符串不需要被本地化(枚举类型没有对本地化提供支持)。除了 GetValues 方法外，Enum 类型还提供了下面两个静态方法用于返回一个枚举类型的符号：

```
// 返回数值的字符串表达形式
public static String GetName(Type enumType, Object value);

// 返回一个字符串数组：枚举类型中的每个符号对应一个数组元素
public static String[] GetNames(Type enumType);
```

上面讨论了很多用来查询一个枚举类型符号的方法。但是我们还需要一个方法来查询一个符号对应的值，例如，它可以转换用户输入到文本框中的符号。Enum 类型的静态方法 Parse 就是这样一个方法，它可以很容易将一个文本符号转换为一个枚举类型的实例：

```
public static Object Parse(Type enumType,
    String value, Boolean ignoreCase);
```

下面的代码演示了 Enum.Parse 的一些用法：

```
// 因为 Orange 被定义为 3，所以 'c' 被初始化为 3
Color c = (Color) Enum.Parse(typeof(Color), "orange", true);

// 因为没有定义 Brown ，所以将抛出一个 ArgumentException 异常
Color c = (Color) Enum.Parse(typeof(Color), "Brown", false);

// 创建一个值为 1 的 Color 枚举实例
Color c = (Color) Enum.Parse(typeof(Color), "1", false);

// 创建一个值为 23 的 Color 枚举实例
Color c = (Color) Enum.Parse(typeof(Color), "23", false);
```

另外，利用 Enum 的静态方法 IsDefined，我们还可以判定一个数值对于某个枚举类型是否合法：

```
// 显示 "True" 因为 Color 中将 Green 定义为 1
Console.WriteLine(Enum.IsDefined(typeof(Color), 1));

// 显示 "True" 因为 Color 中将 Red 定义为 0
Console.WriteLine(Enum.IsDefined(typeof(Color), "Red"));

// 显示 "False" 因为要区分大小写
Console.WriteLine(Enum.IsDefined(typeof(Color), "red"));

// 显示 "False" 因为 Color 中没有定义值为 10 的符号
Console.WriteLine(Enum.IsDefined(typeof(Color), 10));
```

IsDefined 方法通常用来做参数验证，下面是一个例子：

```
public void SetColor(Color c) {
    if (!Enum.IsDefined(typeof(Color), c)) {
        throw(new ArgumentOutOfRangeException("c", "Not a valid Color"));
    }
    ...
}
```

对枚举类型的参数进行验证是必要的，因为可能会有人像下面这样调用 SetColor：

```
SetColor((Color) 547);
```

因为 547 在 Color 中没有对应的符号，所以 SetColor 方法将抛出一个 ArgumentOutOfRangeException 异常，表明哪个参数是无效的，以及参数无效的原因。

最后，System.Enum 类型还提供了一组静态的 ToObject 方法，可以将一个 Byte、SByte、Int16、UInt16、Int32、UInt32、Int64 或者 UInt64 转换为一个枚举类型实例。

枚举类型总是要和某些其他类型联合起来使用。它们通常用于一个类型的字段、属性以及方法参数。这时就会出现一个常见的问题：是将枚举类型嵌套定义在需要它的类型中，还是将枚举类型与需要它的类型定义在同一层次上？如果我们查看 FCL，我们会发现其中的枚举类型通常和需要它的类型定义在同一层次上。原因很简单，因为这样可以减轻开发人员的代码录入工作。所以我们也应该将枚举类型和需要它的类型定义在同一层次上，除非我们担心这样做会导致命名冲突。

13.2 位标记

开发人员通常要用到位标记(bit flag)集合。当我们调用 System.IO.File 类型的 GetAttributes 方法时，它会返回一个 FileAttribute 类型的实例。FileAttribute 类型是一个基础类型为 Int32 的枚举类型，其每个位反映了文件的一个属性(attribute)。FCL 中的 FileAttribute 类型定义如下：

```
[Flags, Serializable]
public enum FileAttributes {
    ReadOnly            = 0x0001,
    Hidden              = 0x0002,
    System              = 0x0004,
    Directory           = 0x0010,
    Archive             = 0x0020,
    Device              = 0x0040,
    Normal              = 0x0080,
    Temporary           = 0x0100,
    SparseFile          = 0x0200,
    ReparsePoint        = 0x0400,
    Compressed          = 0x0800,
    Offline             = 0x1000,
    NotContentIndexed = 0x2000,
    Encrypted           = 0x4000
}
```

要判定一个文件是否为隐藏文件，我们可以执行如下代码：

```
String file = @"C:\Boot.ini";
FileAttributes attributes = File.GetAttributes(file);
Console.WriteLine("Is {0} hidden? {1}", file,
    (attributes & FileAttributes.Hidden) != 0);
```

下面的代码演示了怎样将一个文件的属性改变为只读和隐藏：

```
File.SetAttributes(@"C:\Boot.ini",
    FileAttributes.ReadOnly | FileAttributes.Hidden);
```

如 FileAttribute 类型所示，使用枚举类型来表达可以组合的位标记集合十分常见。虽然枚举类型和位标记有些类似，但是它们具有不同的语义。例如，枚举类型表示单个的数值，而位标记表示一组标记，其中有些位为真、有些位为假。

当定义用于标识位标记的枚举类型时，我们应该显式地为每一个符号赋予映射到单个位的数值。同时强烈建议大家在这样的枚举类型上应用 System.FlagsAttribute 定制特性，看下面的例子：

```
[Flags]  // C#编译器允许我们使用 "Flags" 或者 "FlagsAttribute"
enum Actions {
   Read      = 0x0001,
   Write     = 0x0002,
   Delete    = 0x0004,
   Query     = 0x0008,
   Sync      = 0x0010
}
```

由于 Action 仍是一个枚举类型，所以我们仍然可以使用前一节中描述的所有方法。当然，如果某些方法的行为稍微有一些不同的话就更好了，例如，假设我们有以下代码：

```
Actions actions = Actions.Read | Actions.Write;    // 0x0003
Console.WriteLine(actions.ToString());             // "Read, Write"
```

当 ToString 方法被调用时，它会试图将 actions 表示的数值翻译为枚举类型中对应的符号。现在枚举实例 actions 的数值为 0x0003，因此它没有对应的符号。但是 ToString 方法会检测到 Actions 类型上应用有 [Flags] 特性，它会据此把枚举数值看作是一组位标记、而非一个单独的数值。因为 0x0003 是由 0x0001 和 0x0002 组合而成的，Actions 又定义了 0x0001 和 0x0002，所以 ToString 方法将产生下面的字符串："Read, Write"。如果删除在 Actions 类型上应用的 [Flags] 特性，ToString 将会简单地返回字符串 "3"。

前面一节曾经讨论过 ToString 方法，并且向大家演示过它提供的 3 种格式化输出的方式："G"(常规)，"D"(十进制)，和 "X"(十六进制)。当我们用常规格式来格式化一个枚举类型实例时，ToString 方法会首先检查该类型上是否应用了 [Flags] 特性。如果没有应用该特性，ToString 将查找和枚举数值相匹配的符号，并将其返回。如果应用了 [Flags] 特性，ToString 将查找枚举数值中每一个位的匹配符号，并将它们连接为一个字符串，其中各个符号之间用逗号分隔。

如果大家喜欢，也可以在定义 Actions 类型时不使用 [Flags] 特性，并且仍然可以通过使用"F"格式来得到正确的字符串：

```
// [Flags]        // 这里现在被注掉了
enum Actions {
   Read      = 0x0001,
   Write     = 0x0002,
   Delete    = 0x0004,
   Query     = 0x0008,
   Sync      = 0x0010
}

Actions actions = Actions.Read | Actions.Write;    // 0x0003
Console.WriteLine(actions.ToString("F"));          // "Read, Write"
```

如上一节所述，ToString 方法实际上在内部调用了 System.Enum 的静态方法 Format。也就是说我们同样可以使用“F”格式来调用 Format 静态方法。在调用这两个方法时，如果枚举实例数值包括一个没有匹配符号的位，那么返回的字符串将只包括一个十进制数值，而不会有任何符号。

注意，我们在位标记枚举类型中定义的符号不必非要为 2 的整数次方。例如，Actions 类型可以定义一个值为 0x001F 的符号 All。如果有一个值为 0x001F 的 Actions 类型实例，那么格式化该实例将产生一个包括 “All” 的字符串。其他的符号字符串不会出现。

目前，我们已经讨论了怎样将数值转换为一个标记字符串。利用 Enum 的 Parse 方法，我们也可以将一个以逗号分隔的符号字符串转换为一个数值。看下面的代码：

```
// 因为 Query 被定义为 8，所以 'a' 被初始化为 8
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query", true);

// 因为定义了 Query 和 Read，所以 'a' 被初始化为 9
Actions a = (Actions) Enum.Parse(typeof(Actions), "Query,Read", false);

// 创建一个值为 28 的 Actions 枚举实例
Actions a = (Actions) Enum.Parse(typeof(Actions), "28", false);
Console.WriteLine(a.ToString());          // "Delete, Query, Sync"

// 创建一个值为 333 的 Actions 枚举实例
Actions a = (Actions) Enum.Parse(typeof(Actions), "333", false);
Console.WriteLine(a.ToString());          // "333"
```

当 Parse 方法被调用时，它同样会检查枚举类型上是否应用了 [Flags] 定制特性。如果应用了该定制特性，Parse 将会把字符串拆分成单个的符号，然后查询每一个符号的值，并对相应的值执行位或操作，最后得到一个枚举类型的实例。关于定制特性的更多信息，请参见第 16 章。

[Flags] 特性会影响 ToString、Format 和 Parse 方法的行为。.NET 框架鼓励编译器通过查找该特性来确保枚举类型被当作一组位标记来操作。例如，一个编译器可能只允许在位标记枚举类型上进行位操作，而禁止其他算术操作(如乘法和除法)。C#编译器在这方面完全忽略 [Flags] 特性，我们可以对一个普通枚举类型进行的任何操作都适用于位标记枚举类型。

当使用 Visual Studio .NET 的窗体设计器时，我们可以在设计时利用属性窗口来做各种各样的设置。如果这些设置中的一些为枚举类型，那么窗体设计器会检查其上是否应用了 [Flags] 特性，并据此显示可能的值。

14

数　组

数组允许我们将多个数据项视作一个单独的集合。通用语言运行时(CLR)支持一维数组、多维数组以及交错数组(jagged array，也就是数组的数组)共3种。所有的数组类型都隐含继承自System.Array，System.Array本身又继承自System.Object。这意味着数组总是分配在托管堆上的引用类型，并且应用程序中的数组变量包含的是一个指向数组的引用，而非数组本身。看下面的代码：

```
Int32[] myIntegers;              // 声明一个数组引用
myIntegers = new Int32[100];     // 创建一个含有 100 个 Int32 的数组
```

在上面第1行代码中，myIntegers被声明为一个指向元素类型为Int32的一维数组变量。这时的myIntegers被设为null，因为我们并没有为其分配一个数组。第2行代码为myIntegers分配了一个含有100个Int32的数组，所有这些Int32元素值都被初始化为0。虽然Int32属于值类型，但保存这些Int32值的内存块却是从托管堆中分配的。该内存块中包含着100个未装箱形式的Int32值。最后，代码返回保存这些Int32值的内存块的地址，并将其保存在变量myIntegers中。

我们也可以创建元素为引用类型的数组：

```
Control[] myControls;            // 声明一个数组
myControls = new Control[50];    // 创建一个含有 50 个 Control 引用的数组
```

在上面第1行代码中，myControls被声明为一个指向元素为Control引用(注意是引用)的一维数组变量。这时的myControls被设为null，因为我们并没有为其分配一个数组。第2行代码为myControls分配了一个含有50个Control引用的数组，所有这些Control引用都被初始化为null。因为Control属于引用类型，所以我们创建的仅仅是引用，而不是实际的对象。最后，代码返回保存这些引用的内存块的地址，并将其保存在变量myControls中。

图 14.1 为我们形象地描述了元素为值类型的数组和元素为引用类型的数组在托管堆中的情况：

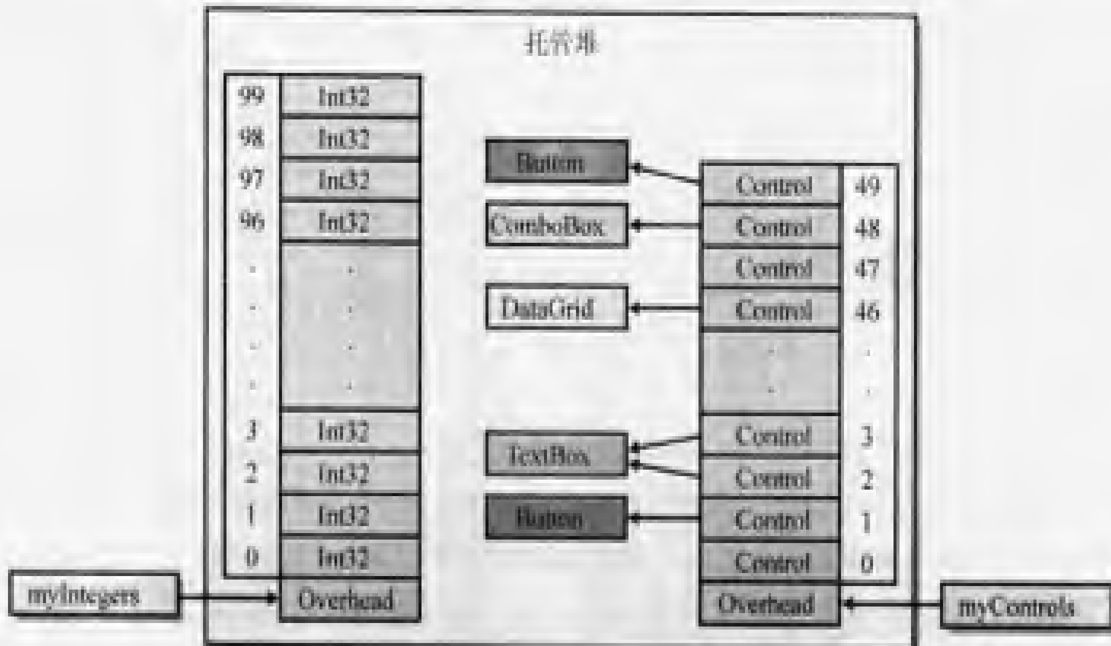


图 14.1 托管堆中元素为值类型的数组和元素为引用类型的数组

图中数组 Controls(译注：应为 myControls)展示的是以下代码执行后的结果：

```
myControls[1] = new Button();
myControls[2] = new TextBox();
myControls[3] = myControls[2];          // 两个元素指向同一个对象

myControls[46] = new DataGrid();
myControls[48] = new ComboBox();
myControls[49] = new Button();
```

通用语言规范(CLS)要求所有的数组都是 0 基数组(译注：即最小索引为 0)。这使得我们可以将用 C#代码创建的数组引用传递给另一门语言(如 Visual Basic)编写的代码。另外，0 基数组也是目前最常用的数组，微软在优化它们的性能方面做了大量的工作。但是，CLR 也支持非 0 基的数组，虽然不鼓励使用它们。对于那些不关心性能或者跨语言移植性的开发人员，本章后面会演示怎样创建和使用非 0 基的数组。

注意在图 14.1 中，每个数组都有一些额外的负载信息。这些信息包括数组的秩(即维数)、每一维的最低下限(大多数时候总为 0)、每一维的长度，以及数组中每一个元素的类型。本章稍后会向大家介绍一些允许我们查询这些负载信息的方法。

到目前为止，大家已经了解了怎样创建一个一维数组。我们应该尽可能地坚持使用一维 0 基数组。一维 0 基数组有时又被称作 SZ 数组(single-dimension、zero-based 数组)，或者向量(vector)。在各种类型的数组中，一维 0 基数组的性能最好，因为我们可以使用一些特殊的 IL 指令(例如 newarr、ldelem、ldelema、ldlen，以及 stelem)来操作它们。但是，如果大家喜欢，也可以使用多维数组。下面是一些使用多维数组的例子：

```
// 创建一个 Double 类型的二维数组
Double[,] myDoubles = new Double[10, 20];

// 创建一个 String 类型的三维数组
String[,,] myStrings = new String[5, 3, 10];
```

CLR 还支持交错数组。一维 0 基交错数组和通常的一维 0 基数组有着同样的性能。但是访问交错数组的元素意味着必须进行两次以上的数组访问。注意交错数组与 CLS 是不兼容的——因为 CLS 不允许一个数组的元素类型为 System.Array——所以它不能在不同的编程语言中传递(译注：这里更准确的说法应该是不能在仅支持 CLS 最小集的不同语言中进行传递。但是仍然可以在两个都支持交错数组之间的语言中传递，比如 Visual Basic .NET 仍然可以使用 C#返回的交错数组)。幸运的是，C#支持交错数组。下面的例子演示了怎样创建一个多边形数组，其中每个多边形包含一个 Point 实例数组：

```
// 创建一个元素为 Point 数组的一维数组
Point[][] myPolygons = new Point[3][];

// myPolygons[0] 指向一个含有 10 个 Point 实例的数组
myPolygons[0] = new Point[10];

// myPolygons[1] 指向一个含有 20 个 Point 实例的数组
myPolygons[1] = new Point[20];

// myPolygons[2] 指向一个含有 30 个 Point 实例的数组
myPolygons[2] = new Point[30];

// 显示第一个多边形中的 Point
for (Int32 x = 0, l = myPolygons[0].Length; x < l; x++)
    Console.WriteLine(myPolygons[0][x]);
```

注意 CLR会验证数组索引的有效性。换句话说,我们不可以创建一个含有100个元素的数组(索引0~99),然后又试图去访问索引为100或者-5的元素。这样的操作会导致一个System.IndexOutOfRangeException异常抛出。允许访问一个数组索引范围以外的内存将破坏类型安全,并造成潜在的安全漏洞。CLR不允许可验证代码中出现这样的行为。通常情况下,索引检查带来的性能损失是微不足道的,因为JIT编译器通常会在一次循环执行之前只检查一次数组界限,而不是每一次循环都做这样的检查。(译注:JIT编译器的这种优化是针对特定的数组类型和代码结构进行的,比如目前微软的JIT编译器只有当我们在一个for循环中访问一维0基数组,并且索引介于0和数组的Length属性——不包括Length——之间所有的元素时才会进行这样的优化。)如果大家仍旧担心CLR索引检测带来的性能损失,则可以考虑使用C#中的非安全代码来访问数组。本章14.5节将演示这样的技巧。

14.1 所有数组的基类:System.Array

System.Array类型中包含着好几种静态成员和实例成员。因为所有的数组都隐含继承自System.Array,所以我们可以使用这些成员来操作各种类型的数组(包括元素为值类型的数组和元素为引用类型的数组)。另外Array还实现了几个接口:ICloneable、IEnumerable、ICollection,以及IList。这些接口使得我们可以在许多情况下方便地使用数组。

表14.1总结了System.Array提供的几种方法以及实现的几种接口。

表14.1 System.Array的成员

成员	成员类型	描述
Rank	只读实例属性	返回数组的维数
GetLength	实例方法	返回数组中指定维数的元素个数
Length	只读实例属性	返回数组中所有元素的总个数
GetLowerBound	实例方法	返回数组中指定维数的下限。大多数时候总为0

续表

成 员	成员类型	描 述
GetUpperBound	实例方法	返回数组中指定维数的上限。大多数时候总为指定维数的元素个数减 1
IsReadOnly	只读实例属性	指示数组是否为只读数组。对于数组来讲，该值总为 false
IsSynchronized	只读实例属性	指示数组的访问是否线程安全。对于数组来讲，该值总为 false
SyncRoot	只读实例属性	返回一个可以用来同步访问数组的对象。对于数组来讲，该值总是指向数组本身
IsFixedSize	只读实例属性	指示数组的大小是否固定。对于数组来讲，该值总是 true
GetValue	实例方法	返回一个数组中指定位置的元素的引用。如果数组中包含的是值类型，返回值将指向对应元素的一个装箱形式的拷贝。这个方法很少用，只有当我们在设计时不知道数组的维数时才使用该方法
SetValue	实例方法	设置数组中指定位置的元素值。这个方法也很少用，只有当我们在设计时不知道数组的维数时才使用该方法
GetEnumerator	实例方法	为数组返回一个 IEnumerator。这使得我们可以使用 C# 的 foreach 语句(或者其他语言中类似的构造)来操作数组。对于多维数组，枚举器在遍历所有元素的时候，从最右边的维数改变的最快
Sort	静态方法	对一个数组、两个数组、或一个数组的一部分中的元素进行排序。数组元素类型必须实现 IComparer 接口(译注：这里的 IComparer 应为 IComparable)，或者必须传递一个实现了 IComparer 接口的对象
BinarySearch	静态方法	使用二分搜索法(binary search algorithm)来搜索指定数组的指定元素。该方法假定数组元素本身已经排好了序。数组元素类型必须实现 IComparer 接口(译注：这里的 IComparer 同样应为 IComparable)。我们通常在调用 BinarySearch 之前应首先调用 Sort 方法

续表

成　员	成员类型	描　述
IndexOf	静态方法	返回在一维数组或者它的一部分中某个值第一次出现的索引值
LastIndexOf	静态方法	返回在一维数组或者它的一部分中某个值最后一次出现的索引值
Reverse	静态方法	反转一维数组或者它的一部分中的元素的顺序
Clone	实例方法	返回一个新创建的数组，该数组是源数组的一个浅拷贝
CopyTo	实例方法	将一个数组中的元素拷贝到另一个数组中
Copy	静态方法	将一个数组中的部分元素拷贝到另一个数组中，并根据需要执行适当的转型
Clear	静态方法	将数组中部分元素设为 0 或者 null 空引用
CreateInstance	静态方法	创建一个数组实例。该方法很少用，它允许我们动态(在运行时)定义拥有任何类型、任何维数、任何上下限的数组
Initialize	实例方法	该方法会为值类型数组中的每一个元素调用默认的构造器。如果数组中的元素为引用类型，该方法将不执行任何操作。C#不允许我们为值类型定义默认的构造器，因此该方法对于 C#中的结构数组来讲没有任何用处。该方法主要是为编译器厂商服务

14.2 数组的转型

对于元素为引用类型的数组，CLR允许我们隐式地将源数组中的元素由一种类型转换为另一种类型。要使这样的转型成功，两个数组类型都必须有同样的维数，并且在源数组元素类型和目标数组元素类型之间必须存在隐式或者显式的转换(译注：如果源数组元素类型和目标数组元素类型之间只存在显式转换，那么要将源数组转换为目标数组，则必须进行显式的转型)。CLR不允许将元素为值类型的数组转型为任何其他类型。(但是，通过使用Array.Copy，我们可以创建一个新的数组来达到期望的效果。)下面的代码演示了怎样执行数组转型：

```
// 创建一个二维 FileStream 数组
FileStream[,] fs2dim = new FileStream[5, 10];

// 隐式转型为一个二维 Object 数组
Object[,] o2dim = fs2dim;

// 不能将一个二维数组转型为一个一维数组，
// 编译器报错 error CS0030：无法将类型
// "object[*,*]" 转换为 "System.IO.Stream[]"
Stream[] s1dim = (Stream[]) o2dim;

// 显式转型为一个二维 Stream 数组
Stream[,] s2dim = (Stream[,]) o2dim;

// 显式转型为一个二维 Type 数组：能够通过编译
// 但是会在运行时抛出 InvalidCastException 异常
Type[,] t2dim = (Type[,]) o2dim;

// 创建一个一维 Int32 数组(值类型)
Int32[] i1dim = new Int32[5];

// 不能将一个值类型数组转型为任何其他的数组，
// 编译器报错 error CS0030: 无法将类型 "int[]"
// 转换为 "object[]"
Object[] o1dim = (Object[]) i1dim;

// Array.Copy 创建一个新的数组，并强行将源数组
// 中的每一个元素转型为目标数组中期望的类型。以
// 下代码创建的数组元素为一组已装箱的 Int32 引用
Object[] o1dim = new Object[i1dim.Length];
Array.Copy(i1dim, o1dim, i1dim.Length);
```

Array.Copy方法不仅能使我们可以快速地将一个数组中的元素拷贝到另一个数组中。如果需要的话，Copy方法还能够在拷贝每一个元素时进行相应的类型转换。Copy方法能够执行以下几种转换。

- 将值类型元素装箱为引用类型元素，例如在将一个 Int32[ ]拷贝到一个 Object[ ]中时。
- 将引用类型元素拆箱为值类型元素，例如在将一个 Object[ ]拷贝到一个 Int32[ ]中时。
- 拓宽(widen)CLR 基元类型，例如在将一个 Int32[ ]拷贝到一个 Double[ ]中时。

下面的例子演示了 Copy 的一些用途：

```
// 定义一个实现了 ICloneable 接口的值类型
struct MyValueType : ICloneable {
    ...
}

class App {
   static void Main() {

       // 创建一个含有 100 个值类型元素的数组
       MyValueType[] src = new MyValueType[100];

       // 创建一个元素为 ICloneable 引用的数组
       ICloneable[] dest = new ICloneable[src.Length];

       // 初始化数组中的 ICloneable 元素，使
       // 其指向源数组中元素的已装箱版本
       Array.Copy(src, dest, src.Length);
   }
}
```

可以想见，.NET 框架类库(FCL)会非常频繁地用到 Array 的 Copy 方法。

14.3 数组的传递与返回

数组总是以引用的方式传递给方法的。因为 CLR 不支持常数参数，所以方法可以改变数组中的元素。如果不希望发生这样的行为，我们必须做一个数组的拷贝，然后将该拷贝传递给方法。注意 Array.Copy 方法执行的是浅拷贝，因此如果源数组中的元素是引用类型，该方法返回的新数组中的元素将指向源数组中元素所引用的对象。

为了获得一个深拷贝，我们可能希望克隆数组中的每一个元素，但是这要求每个元素的类型都要实现 ICloneable 接口。

作为一种可选的做法，我们也可以将每个对象序列化到一个 System.IO.MemoryStream 中，然后再立即对该内存流执行反序列化，从而得到一个新的对象。根据对象的类型，这些操作的性能代价可能非常高昂，而且并不是所有的类型都是可以被序列化的。

类似地，一些方法返回的也是一个数组的引用。如果是方法自己构造并初始化的数组，那么返回该数组的引用一般没有什么问题。但是如果方法返回的是一个由类型内部的字段维护的数组引用的话，我们必须确定是否允许方法的调用者直接访问该数组。如果答案是确定的，那么返回这样的数组引用也没什么。但是大多数情况下，我们并不希望方法的调用者有这样的访问能力。所以，我们的方法应该创建一个新的数组，然后调用 Array.Copy，最后返回这个新创建的数组引用。同样，我们还可能希望在返回数组引用之前克隆其中的每个对象。

如果我们定义了一个返回数组引用的方法，而在某些情况下数组又不含任何元素，那么这时我们的方法既可以返回一个 null，也可以返回一个长度为 0 的数组引用。当我们实现这样的方法时，微软强烈建议我们让该方法返回一个 0 长的数组，因为这样会简化调用该方法的开发人员的编码工作。例如在下面的代码中，即使没有 appointments 可以遍历，代码仍会正常运行：

```
// 下面的代码很容易编写和理解
Appointment[] appointments = GetAppointmentsForToday();
for (Int32 a = 0, n = appointments.Length; a < n; a++) {
    ...
}
```

如果没有 appointments 可以遍历，下面的代码也可以正常运行，但是代码编写和理解起来稍微有些困难：

```
// 下面的代码难以编写和理解
Appointment[] appointments = GetAppointmentsForToday();
if (appointments != null) {
    for (Int32 a = 0, n = appointments.Length; a < n; a++) {
        ...
    }
}
```

如果我们将方法设计为返回 0 长的数组、而不是 null 的话，方法的调用者在使用它们的时候就会轻松一些。顺便提一句，我们应该以同样的方式来处理字段。如果我们的类型中有一个字段为一个数组引用，那么即使数组中没有元素，我们也应该使该字段指向一个 0 长的数组。将字段设为 null 将使得我们的类型难以使用。

14.4 创建下限非 0 的数组

本章前面曾经提过 CLR 允许我们创建和使用下限非 0 的数组。我们可以通过调用 Array 的静态方法 CreateInstance 来动态地创建这样的数组。该方法有几个重载版本，它们允许我们指定数组元素的类型、数组的维数、每一维的下限，以及每一维的元素个数。CreateInstance 方法会为数组分配内存，并将这些参数信息保存在数组内存块中的负载信息部分，然后返回数组引用。我们可以将 CreateInstance 返回的数组引用转型为一个便于我们访问其元素的数组变量。

下面的代码演示了怎样动态创建一个元素为 System.Decimal 的二维数组。其中第一维表示年份，其范围为 1995~2004(包括 2004)。第二维表示季度，其范围为 1~4(包括 4)。

```
// 创建一个二维数组 [1995..2004][1..4]
Int32[] lowerBounds = { 1995, 1 };
Int32[] lengths = { 10, 4 };
Decimal[,] quarterlyRevenue = (Decimal[,])
   Array.CreateInstance(typeof(Decimal), lengths, lowerBounds);
```

下面的代码遍历动态数组中的所有元素。我们可以将数组的上下限硬编码到代码中，这可以获得较好的性能，但是这里处于演示目的使用了 System.Array 的 GetLowerBound 和 GetUpperBound 方法：

```
Int32 firstYear = quarterlyRevenue.GetLowerBound(0);
Int32 lastYear = quarterlyRevenue.GetUpperBound(0);
Console.WriteLine("{0,4} {1,9} {2,9} {3,9} {4,9}",
    "Year", "Q1", "Q2", "Q3", "Q4");

for (Int32 year = firstYear; year <= lastYear; year++) {
   Console.Write(year + " ");

   for (Int32 quarter = quarterlyRevenue.GetLowerBound(1);
      quarter <= quarterlyRevenue.GetUpperBound(1); quarter++) {

      Console.Write("{0,9:C} ", quarterlyRevenue[year, quarter]);
   }
   Console.WriteLine();
}
```

14.5　快速数组访问

当我们每次访问一个数组中的元素时，CLR 都会确保索引不会超出数组的上下限。这可以防止我们访问数组以外的内存，因为那样的行为会潜在地损坏其他对象。如果使用无效的索引来访问数组元素，CLR 会抛出一个 System.IndexOutOfRangeException 异常。

可以想见，CLR 的索引检查会有一些性能方面的代价。如果大家对自己的代码有足够的信心，并且不介意使用不可验证(非安全)的代码，则可以在访问一个数组时不让 CLR 执行索引检查。下面的 C#代码演示了这种技巧：

```
using System;

class App {
    unsafe static void Main() {

        // 创建一个包含 5 个 Int32 元素的数组
        Int32[] arr = new Int32[] { 1, 2, 3, 4, 5};

        // 获取一个指向数组第 0 个元素的指针
        fixed (Int32* element = &arr[0]) {

            // 遍历数组中每一个元素
            // 注意：下面的代码中有一个 bug!
            for (Int32 x = 0, n = arr.Length; x <= n; x++) {
                Console.WriteLine(element[x]);
            }
        }
    }
}
```

在控制台上键入以下命令编译上面的代码(假设源代码保存在一个名为 UnsafeArrayAccess.cs 的文件中)：

```
csc.exe /unsafe UnsafeArrayAccess.cs
```

运行编译后得到的可执行文件，我们将会看到以下输出：

```
1
2
3
4
5
0
```

虽然我们期望上面的代码输出的是5个元素，但实际上它输出的却是6个，因为代码中有一个bug：在for循环中，测试表达式应该为x<n，而非x<=n。在使用非安全代码时，我们必须非常小心！

如果我们使用ILDasm.exe来查看上面的Main方法编译后的IL代码，我们将会看到以下内容(代码经过了注释)：

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      58 (0x3a)
    .maxstack 3
    .locals ( [0] int32[] arr,
              [1] int32& pinned element,
              [2] int32 x,
              [3] int32 n)
     // 创建一个包含5个Int32元素的数组
     IL_0000:      ldc.i4.5
     IL_0001:      newarr   [mscorlib]System.Int32
     // 用元数据中存储的值初始化数组元素
     IL_0006:      dup
     IL_0007:      ldtoken  field valuetype
         '<PrivateImplementationDetails>'/'$$struct0x6000001-1'
         '<PrivateImplementationDetails>'::'$$method0x6000001-1'
     IL_000c:      call
         void [mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::
             InitializeArray(class [mscorlib]System.Array,
             valuetype [mscorlib]System.RuntimeFieldHandle)
     // 将数组引用保存在arr变量中
     IL_0011:      stloc.0
     // 获取arr中第0个元素的地址，然后将其保存在element中
     IL_0012:      ldloc.0
     IL_0013:      ldc.i4.0
     IL_0014:      ldelema  [mscorlib]System.Int32
     IL_0019:      stloc.1
     // 将x初始化为0
     IL_001a:      ldc.i4.0
     IL_001b:      stloc.2
     // 将n初始化为arr的长度
     IL_001c:      ldloc.0
     IL_001d:      ldlen
     IL_001e:      conv.i4
     IL_001f:      stloc.3
```

```
    // 跳转到 for 循环测试上
    IL_0020:  br.s        IL_0032

    // 计算 element + (4 * x) -- 4 为一个 Int32 所含的字节个数
    IL_0022:  ldloc.1
    IL_0023:  conv.i
    IL_0024:  ldc.i4.4
    IL_0025:  ldloc.2
    IL_0026:  mul
    IL_0027:  add

    // 将该地址上的值传递给 Console.WriteLine 方法
    IL_0028:  ldind.i4
    IL_0029:  call        void [mscorlib]System.Console::WriteLine(int32)

    // 给 x 加 1
    IL_002e:  ldloc.2
    IL_002f:  ldc.i4.1
    IL_0030:  add
    IL_0031:  stloc.2

    // for 循环测试：如果 x <= n 便继续循环
    IL_0032:  ldloc.2
    IL_0033:  ldloc.3
    IL_0034:  ble.s   IL_0022

    // 循环结束：将 element 设为 null(处于安全考虑)
    IL_0036:  ldc.i4.0
    IL_0037:  conv.u
    IL_0038:  stloc.1

    // 从 Main 中返回
    IL_0039:  ret
} // end of method App::Main
```

作为对比，下面是没有使用非安全代码的版本：

```
using System;
class App {
    static void Main() {
      Int32[] arr = new Int32[] { 1, 2, 3, 4, 5};
      for (Int32 x = 0, n = arr.Length; x <= n; x++) {
         Console.WriteLine(arr[x]);
      }
    }
}
```

编译上面的代码，并使用ILDasm.exe查看其生成的IL代码，我们将会看到以下内容：

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size 43 (0x2b)
    .maxstack 3
    .locals init (    [0] int32[] arr,
                [1] int32 x,
                [2] int32 n)

    // 创建一个包含5个Int32元素的数组
    IL_0000:  ldc.i4.5
    IL_0001:  newarr   [mscorlib]System.Int32

    // 用元数据中存储的值初始化数组元素
    IL_0006:  dup
    IL_0007:  ldtoken  field valuetype
         '<PrivateImplementationDetails>'/'$$struct0x6000001-1'
         '<PrivateImplementationDetails>'::'$$method0x6000001-1'
    IL_000c:  call
       void [mscorlib]System.Runtime.CompilerServices.RuntimeHelpers::
       InitializeArray(class [mscorlib]System.Array,
       valuetype [mscorlib]System.RuntimeFieldHandle)

    // 将数组引用保存在arr变量中
    IL_0011:  stloc.0
    // 将x初始化为0
    IL_0012:  ldc.i4.0
    IL_0013:  stloc.1

    // 将n初始化为arr的长度
    IL_0014:  ldloc.0
    IL_0015:  ldlen
    IL_0016:  conv.i4
    IL_0017:  stloc.2

    // 跳转到for循环测试上
    IL_0018:  br.s         IL_0026

    // 将arr[x]传递给Console.WriteLine方法
    IL_001a:  ldloc.0
    IL_001b:  ldloc.1
    IL_001c:  ldelem.i4
    IL_001d:  call         void [mscorlib]System.Console::WriteLine(int32)
```

```
    // 给 x 加 1
    IL_0022:  ldloc.1
    IL_0023:  ldc.i4.1
    IL_0024:  add
    IL_0025:  stloc.1

    // for 循环测试：如果 x < n(译注：应为 x<=n)便继续循环
    IL_0026:  ldloc.1
    IL_0027:  ldloc.2
    IL_0028:  ble.s    IL_001a

    // 从 Main 中返回
    IL_002a:  ret
} // end of method App::Main
```

显然，类型安全的那个版本的 IL 代码要少一些。正是类型安全的那个版本中的 ldelem 指令导致了 CLR 执行索引检查。而非安全版本使用的是 ldind.i4 指令，该指令直接从一个内存地址中获取 4 字节的数值。注意，非安全数组操作只可以用于元素为以下类型的数组：SByte、Byte、Int16、UInt16、Int32、UInt32、Int64、UInt64、Char、Single、Double、Decimal、Boolean、枚举类型，或者字段为上面所述类型的结构类型(译注：可用于非安全数组操作的结构类型的字段除了可以是上述类型外，还可以为符合非安全数组操作的结构类型，这里有一个递规定义的概念)。

14.6　重新调整数组长度

当我们在编译时不知道数组元素的类型时，Array 的静态方法 CreateInstance 可以帮助我们动态地创建数组。还有一些情况是我们在编译时不知道数组的维数和每个维数的上下限，这时 CreateInstance 方法也显得十分有用。前面 14.4 一节向大家演示了怎样动态创建一个任意上下限的数组。我们还可以用 CreateInstance 方法来重新调整一个数组的长度，看下面的代码：

```
using System;
class App {
    static void Main() {
        // 创建一个含有 3 个元素的数组
        Int32[] arr = new Int32[] { 1 , 2, 3 };

        // 显示数组中所有的元素
        foreach (Int32 x in arr)
              Console.Write(x + " ");
        Console.WriteLine();
```

```
        // 重新调整数组长度使其包含 5 个元素
        arr = (Int32[]) Redim(arr, 5);

        // 显示数组中所有的元素
        foreach (Int32 x in arr)
            Console.Write(x + " ");
        Console.WriteLine();

        // 重新调整数组长度使其包含 2 个元素
        arr = (Int32[]) Redim(arr, 2);

        // 显示数组中所有的元素
        foreach (Int32 x in arr)
            Console.Write(x + " ");
    }

    public static Array Redim(Array origArray, Int32 desiredSize) {

        // 确定每个元素的类型
        Type t = origArray.GetType().GetElementType();

        // 创建一个含有期望元素个数的新数组
        // 新数组的类型必须匹配原数组的类型
        Array newArray = Array.CreateInstance(t, desiredSize);

        // 将原数组中的元素拷贝到新数组中
        Array.Copy(origArray, 0,
            newArray, 0, Math.Min(origArray.Length, desiredSize));

        // 返回新数组
        return newArray;
    }
}
```

编译并运行上面的代码，我们将会看到以下输出：

```
1 2 3
1 2 3 0 0
1 2
```

15

接　口

本章首先向大家解释如何使用接口来描述一个类型的功能。然后，我们将详细讨论一个类型如何通过实现接口来提供这种定义良好的功能，从而使得类型在各种场合都能被方便地使用。最后，我们还会学习几种有用的技巧，借助这些技巧我们可以避免一些接口使用过程中出现的问题。具体而言，就是方法名称的重复和编译时类型安全两个问题。

15.1　接口与继承

在程序设计过程中，如果能将一个对象看作多个类型将是非常有用的，因为对象的类型描述了它的能力和行为。比如，我们可能会设计一个 SortedList 类型来保存一个有序对象集合。只要对象的类型支持将它和其他类型的对象比较的能力，我们便可将任何继承自 System.Object 类型的对象添加到 SortedList 中。

从某种角度来讲，我们可能期望 SortedList 接受的类型继承自某个假想的 System.Comparable。但是，许多现存的类型并没有继承 System.Comparable。这样，我们便不能将这些类型的对象添加到一个 SortedList 中，SortedList 类型也将变得不是很有用。

理想情况下，我们可能希望 SortedList 类型能够接受现存的继承自 System.Object 的类型，然后在某些时候又可以将它看作是从 System.Comparable 继承的类型。这种将一个对象看成多个类型的能力通常称作**多继承**(multiple inheritance)。通用语言运行时(CLR)支持单实现继承和多接口继承。

CLR规定一个类型只能有一个基类型，System.Object是所有类型的最终基类型。这种继承称作实**现继承**(implementation inheritance)，因为派生类型继承了基类型所有的行为和能力：派生类型可以有像基类型一样的行为。但是，在继承基类型的同时，派生类型也可以重写(override)基类型的某些行为。这种重写基类型的行为(也可称作实现)使得新的派生类型成为一个独特的类型。

接口继承(interface inheritance)意味着一个类型继承的是接口中的方法签名，而非方法实现。当一个类型继承了一个接口时，它只是在许诺提供其中的方法实现；如果类型没有提供接口方法的实现，那么类型将被认为是抽象的，从而不可能被执行实例化。

接口不会继承自任何的System.Object派生类型。接口仅仅是一个包含着一组虚方法的抽象类型，其中每一个方法都有它们的名称、参数和返回值类型。接口方法不能包括任何实现，因此接口是不完整的(抽象的)。注意接口中也可以定义事件、无参属性以及含参属性(C#中又称索引器)，因为它们都只不过是映射到方法上的语法缩写而已。CLR还允许接口包含静态方法、静态字段、常数、以及静态构造器。但是，与CLS兼容的接口类型不允许有任何静态成员，因为一些编程语言不能定义或者访问它们。实际上，C#编译器阻止我们在一个接口中定义任何的静态成员。另外，CLR也不允许接口中包含任何的实例字段或实例构造器。

下面是.NET框架类库(FCL)中的4个接口定义(译注：这里使用“命名空间＋接口”的形式只是为了说明各个接口所属的命名空间，而非定义接口的正确做法)：

```
public interface System.IComparable {
   Int32 CompareTo(Object object);
}

public interface System.Collections.IEnumerable {
   IEnumerator GetEnumerator();
}

public interface System.Collections.IEnumerator {

   Boolean MoveNext();

   void Reset();

   Object Current { get; }                    // 只读属性
}
```

```
public interface System.Collections.ICollection : IEnumerable {

    void CopyTo(Array array, Int32 index);

    Int32 Count { get; }                    // 只读属性

    Boolean IsSynchronized { get; }         // 只读属性

    Object SyncRoot { get; }                // 只读属性
}
```

根据约定，接口类型的名称要加一个大写的字母 I 前缀。接口定义允许使用修饰符——例如 public、protected、internal 以及 private——这与类或结构没什么两样。当然，定义接口时，99%的情况都使用的是 public。这些修饰符控制着接口定义的可见性，指出了哪些引用可以看到它。

> **重要** 一个接口的非静态方法总被认为是公有的虚方法。这不可以改变。但是，在 C#中，如果我们在一个类型内实现接口方法的时候忽略了 virtual 关键字，那么该方法将被认为是一个密封(sealed)的虚方法——继承了该实现类型的其他类型将不可以再重写该方法。

虽然一个接口不能继承其他类型的实现，但是它可以"继承"其他接口所约定的合同(例如 ICollection 就继承了 IEnumerable)。实际上，一个接口可以包含多个接口的合同。当一个类型"继承"某个接口时，它不仅要实现该接口定义的所有方法，还要实现该接口从其他接口中"继承"而来的所有方法。例如，任何实现了 ICollection 接口的类型除了必须提供 CopyTo、Count、IsSynchronized 和 SyncRoot 成员的实现之外，它还要提供 IEnumerable 接口中 GetEnumerator 方法的实现。

下面的代码展示了 MSCorLib.dll 中定义的 System.ICloneable 接口：

```
public interface ICloneable {
    Object Clone();
}
```

下面的代码演示了怎样定义一个实现该接口的类型，以及如何使用该接口来克隆一个对象：

```
using System;

// Point 继承自 System.Object，并实现了 ICloneable 接口
public sealed class Point : ICloneable {
   public Int32 x, y;

   public Point(Int32 x, Int32 y) {
      this.x = x;
      this.y = y;
   }

   public override Boolean Equals(Object o) {
      Point other = o as Point;
      if (other == null) return false;
      return (x == other.x) && (y == other.y);
   }

   // 这里是对 ICloneable.Clone 方法的实现
   public Object Clone() { return MemberwiseClone(); }
}

class App {
   static public void Main() {
      Point p1 = new Point(1, 2);

      // 创建另一个有着相同值的 Point
      Point p2 = (Point) p1.Clone();

      // p1 和 p2 指向两个不同的对象：结果将显示 False
      Console.WriteLine(Object.ReferenceEquals(p1, p2));

      // p1 和 p2 有着相同的值：结果将显示 True
      Console.WriteLine(p1.Equals(p2));
   }
}
```

如前所述，一个类型必须继承自另外一个类型(上面的例子继承的是 System.Object)。除此之外，一个类型还可以实现 0 个或多个接口。例如，FCL 中的 System.String 类型就继承自 System.Object，并实现了 IComparable、ICloneable、IConvertible 和 IEnumerable 共 4 个接口。String 类型不需要实现它的基类 Object 提供的方法，它可以简单地将这些方法继承下来就可以了。但是 String 类型必须实现所有接口中声明的方法。否则，它将是一个不完整的类型(即抽象类型)。

在本节开始，我们曾经谈到过接口使得我们可以将一个对象看作为不同的类型。实际上，实现了多个接口的类型允许我们将它的对象看作这些接口中的任何一个。看下面的代码：

```
// 创建一个 String 对象
String s = "Jeffrey";
// 利用 s 我们可以调用任何在 String、Object、IComparable、
// ICloneable、IConvertible、或 IEnumerable 中定义的方法

// 定义一个 IComparable 变量使其引用 s
IComparable comparable = s;
// 利用 comparable，我们可以调用任何在 IComparable 中定义的方法

// 定义一个 ICloneable 变量使其引用 s
ICloneable cloneable = s;
// 利用 cloneable，我们可以调用任何在 ICloneable 中定义的方法

// 定义一个 IEnumerable 变量使其引用 s
IEnumerable enumerable = (IEnumerable) cloneable;
// 我们可以将一个对象从一个接口转型为另一个接口，只要该对象
// 的类型同时实现了两个接口
```

在这段代码中，使用哪个变量没有多大关系；我们一直是通过变量 s 来操作 String 对象的。但是，变量的类型界定了我们可以在 String 对象上进行哪些合法的操作。

再回到 SortedList 的讨论。现在我们有可能将一个 String 对象放在 SortedList 中了，因为它实现了 IComparable 接口。剩下的次要问题就是 String 类型必须实现 CompareTo 方法(定义在 IComparable 接口中)了，该方法比较两个 Object，然后返回一个值来表示应该把哪个对象放在前面。

但是，IComparable 接口的 CompareTo 方法接受的是一个 Object 参数，而不是一个 String 参数。任何实现 CompareTo 方法的类型都必须定义该方法使其签名和接口方法的签名完全匹配。在该方法内部，代码可以做任何必要的转型来执行期望的行为。本章 15.5 一节会对接口的这种“缺点”给出相应的解决办法。

只要一个类型(如 String、DataTime、Int32 等)实现了 IComparable 接口，那么 SortedList 类型就可以管理它们的对象。

重要 和引用类型相似，一个值类型可以实现0个或多个接口。但是，当我们将一个值类型实例转型为一个接口类型时，该值类型实例必须被执行装箱。这是因为接口总被认为是引用类型，并且它们定义的方法总是虚方法。大家应该记得本书前面讲过未装箱形式的值类型没有指向类型方法表的指针。对值类型执行装箱将使得CLR能够查询类型的方法表，从而便可以在其上调用虚方法。

经常有开发人员问我这样的问题，“我应该设计一个基类型、还是一个接口类型呢？”这个问题的答案并不总是显而易见。下面的一些指导原则也许能够在这个问题上帮助大家：

- **IS-A 与 CAN-DO 关系** 一个类型仅可以继承一个实现。如果派生类型不能和基类型构成一个IS-A关系，那么就不要使用基类型，而应该采用接口。接口隐含有一个CAN-DO关系。如果认为CAN-DO功能属于多个对象类型，那么也应该采用接口。

- **易用性** 对于开发人员来说，通过继承一个基类型来定义新的类型要比创建一个接口更为方便。基类型可以提供很多功能，这样我们在派生类型中只需要对它的行为做一些少量的改动即可。但如果是接口，新类型就必须实现其所有的成员。

- **一致的实现** 不管一个接口合同在文档中记录的有多好，要每个人都百分之百地正确实现它们是不可能的。实际上，COM在这个问题上就深受其害，这也是为什么一些COM对象只能在微软的Word或IE流览器中正常工作的原因。通过提供有着良好默认实现的基类型，我们可以先使用一个正常工作、并经过严格测试的类型，然后再根据需要对相应的部分做改动。

- **版本** 如果我们向一个基类型中添加了一个方法，那么其派生类型将自动继承新成员的默认实现。实际上，用户的源代码甚至不需要重新编译。而向接口中添加新的成员将会强制要求接口的用户修改源代码，并重新编译。

在 FCL 中，和数据流相关的类就使用了实现继承的设计。System.IO.Stream 类是一个抽象基类。它提供了很多方法，如 Read 和 Write。而其他的一些数据流类——例如 System.IO.FileStream、System.IO.MemoryStream、System.Net.Sockets.NetworkStream——就继承自 Stream。微软在 Stream 类和这 3 个类之间选择了 IS-A 关系，因为这样会使得这些具体类的实现变得更加容易。例如，派生类只需要实现同步 I/O 操作，然后就可以直接从 Stream 基类中继承获得异步 I/O 操作的能力。

必须承认，为数据流相关的类选择使用实现继承并不是一个明显的决定。Stream 基类实际上提供了很少的实现。但是，如果我们看一下 Windows 窗体控件相关的类，如 Button、Checkbox、ListBox、以及所有其他继承自 System.Windows.Forms.Control 的控件，我们很容易想象各种各样的控件类都从 Control 中继承了很多实现代码。

与为数据流相关的类选择实现继承相反，微软对 FCL 中的集合类使用了基于接口的设计。例如，System.Collections 命名空间中就定义了几种与集合相关的接口：IEnumerable、ICollection、IList 和 IDictionary。然后又提供了许多具体类，例如 ArrayList、Hashtable、Queue、SortedList 等等，并让它们实现上述接口。因为各种集合类的实现彼此之间都有很大的差别，所以微软的设计者们为它们选择了类和接口之间的 CAN-DO 关系。换句话说，在 ArrayList、Hashtable 和 Queue 之间没有太多的代码可共享。

然而，所有这些集合类提供的操作都相当一致。例如，它们都包含一组可枚举的元素，并且它们都允许添加和删除元素。这种一致性是为集合类采用基于接口的设计的真正意义所在。如果我们引用的一个对象的类型实现了 IList 接口，那么我们不用知道正在操作的集合的确切类型，就可以编写代码来对其中的元素进行添加、删除、以及搜索操作。这的确是一种非常强大的机制。

15.2　设计支持插件组件的应用程序

当我们创建可扩展的应用程序时，接口应该处于中心位置。假设我们正在编写一个应用程序，并且希望其他人创建的类型能够被我们的应用程序无缝地加载和使用。下面就提供了设计这样的应用程序的方法。

- 创建一个程序集，然后在其中定义接口，接口的方法将用于应用程序和插件组件的通信机制。在为接口方法定义参数和返回值时，我们应该尽可能地使用定义在 MSCorLib.dll 中的其他接口和类型。但如果确实希望传递、或者返回我们自己定义的数据类型时，则应该把它们也定义在该程序集中。一旦建立好接口定义后，我们应该给该程序集指定一个强命名(本书第 3 章对此有详细讨论)，然后将其打包并部署到合作伙伴和用户那里。这样以后就把该程序集视作为一个恒定不变的程序集——这里“恒定不变”的意思是指其内容不会改变。

> **注意** 我们可以使用 MSCorLib.dll 中定义的类型是因为 CLR 总是会加载和 CLR 自身版本匹配的那个 MSCorLib.dll。而且一个进程中只能载入一个版本的 MSCorLib.dll，换句话说，不同版本的 MSCorLib.dll 不会以并存(side-by-side)(第 3 章对此有详细的描述)的方式加载。这样，我们就不会有任何的类型版本不匹配的问题了。另外，使用这样的加载方式应用程序需要的内存也会更少。

- 创建一个单独的程序集用于包含我们的应用程序所使用的其他类型。该程序集显然要引用到前一个程序集中定义的接口和类型。我们可以按照我们的意愿任意改变该程序集中的代码。因为插件组件不会引用到该程序集，所以如果需要的话，我们甚至可以每隔一小时提供一个该程序集的新版本，这对插件开发人员不会造成任何影响。
- 插件开发人员当然会在他们的程序集中定义自己的类型。他们的程序集也将会引用到我们前面定义的接口程序集中的类型。插件开发人员也可以随时提供新版的程序集，我们的应用程序则可以继续正常使用这些新版的插件组件，而不会遇到任何问题。

本节包含的一些信息非常重要。当使用跨程序集的类型时，我们需要对程序集的版本问题给予足够的重视。我们应该在这个问题上花费一些时间，将用于跨程序集通信的那些类型分离到单独的程序集中实现，并在以后的开发过程中避免更换或者改变这些类型定义——也不要改变程序集的版本号。

另外，我们也要避免自己定义的类型的基类型出现在其他的程序集中，虽然“.NET 框架设计指南”(译注：读者可在安装.NET 框架 SDK 的文档中、或者 MSDN 网站上找到该部分)鼓励这么做。但是，这么做是错误的，因为这样会导致一个程序集和其他程序集的特定版本捆绑在一起。又由于 CLR 的并存(side-by-side)支持，这将很有可能导致一个程序集的好几个版本都被载入到同一个应用程序域(AppDomain)中。这会占用相当可观的内存，从而严重地损伤系统性能。而且还有可能阻止程序集之间的通信，或者至少是使其变得困难。

另外，大多数编译器都不允许我们在生成托管模块时引用一个程序集的多个版本。这种限制将使代码很难在利用一些新的特性(由新版的程序集所提供)的同时，还能够保持程序集之间的通信(通过旧版的程序集中必要的类型)。

15.3　使用接口改变已装箱值类型中的字段

让我们来做个游戏，考查一下大家对值类型、装箱、拆箱理解的怎么样。考虑下面的代码，大家看看是否能够判断出控制台上输出的内容：

```
using System;

// Point 是一个值类型
struct Point {
   public Int32 x, y;

   public void Change(Int32 x, Int32 y) {
      this.x = x; this.y = y;
   }

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

class App {
   static void Main() {
      Point p = new Point();

      p.x =p.y = 1;
      Console.WriteLine(p);            // 显示 (1, 1)

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

      Object o = p;
      Console.WriteLine(o);            // 显示 (2, 2)

      ((Point) o).Change(3, 3);        // 在堆栈上改变临时的 Point
      Console.WriteLine(o);            // 显示 (2, 2)
   }
}
```

这段代码很简单，Main 首先在它的堆栈上创建了一个 Point 值类型实例，然后又将其 x 和 y 字段设为 1。第 1 次调用 WriteLine 会在未装箱形式的 Point 上调用 ToString，结果输出显示为“(1, 1)”，这和我们期望的结果是一致的。然后，Main 又在 p 上调用了 Change 方法，这会将堆栈上的 p 的 x 和 y 字段都改变为 2。第 2 次调用 WriteLine 也将如我们所期望的那样显示“(2, 2)”。

接着，p 被执行装箱，o 指向装箱后的 Point 对象。第 3 次调用 WriteLine 仍将显示“(2, 2)”，这也是我们期望的结果。最后 Main 希望调用 Change 方法来改变已装箱 Point 对象中的字段。但是 Object(变量 o 的类型)对 Change 方法一无所知，所以我们首先必须将 o 转型为一个 Point。将 o 转型为 Point 会对 o 执行拆箱操作，并将已装箱 Point 中的字段拷贝到线程堆栈上一个临时的 Point 中。然后这个临时的 Point 上的 x 字段和 y 字段被改为 3 和 3，但是已装箱 Point 对象不受这种改变的影响。当第 4 次调用 WriteLine 时，结果仍为“(2, 2)”。许多开发人员可能都没有预料到这一点。

也有一些语言(如托管扩展 C++)允许我们改变已装箱值类型中的字段，然而 C#不允许我们这样做。但是我们可以利用接口来欺骗 C#使其改变已装箱值类型中的字段。下面的代码是对前面代码的一个修订：

```
using System;

// 定义 Change 方法的接口
interface IChangeBoxedPoint {
   void Change(Int32 x, Int32 y);
}

// 让 Point 值类型实现 IChangeBoxedPoint 接口
struct Point : IChangeBoxedPoint {
   public Int32 x, y;

   public void Change(Int32 x, Int32 y) {
      this.x = x; this.y = y;
   }

   public override String ToString() {
      return String.Format("({0}, {1})", x, y);
   }
}
```

```
class App {
    static void Main() {
        Point p = new Point();

        p.x = p.y = 1;
        Console.WriteLine(p);           // 显示 (1, 1))

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

        Object o = p;
        Console.WriteLine(o);           // 显示 (2, 2)

        ((Point) o).Change(3, 3);       // 在堆栈上改变临时的 Point
        Console.WriteLine(o);           // 显示 (2, 2)

        // 对 p 执行装箱，然后改变已装箱对象，最后将之丢弃
        ((IChangeBoxedPoint) p).Change(4, 4);
        Console.WriteLine(p);           // 显示 (2, 2)

        // 改变已装箱对象，并显示其内容
        ((IChangeBoxedPoint) o).Change(5, 5);
        Console.WriteLine(o);           // 显示 (5, 5)
    }
}
```

这段代码和前面的代码几乎是一样的。主要的区别在于我们用 IChangeBoxedPoint 接口定义了 Change 方法，然后又让 Point 类型实现了该接口。在 Main 中，前 4 次调用 WriteLine 方法以及输出结果和前面的代码都是一样的(与预期的也一样)。但是 Main 最后又增加了两个例子。

在第一个例子中，未装箱形式的 Point(即 p)被转型为一个 IChangeBoxedPoint。这将导致 p 被装箱。在已装箱的值上调用 Change 会将它的 x 字段和 y 字段改为 4 和 4。但是在 Change 返回之后，已装箱对象将立即成为可被垃圾收集器回收的垃圾对象。所以第 5 次调用 WriteLine 将显示“(2, 2)”——许多开发人员可能也没有预料到这样的结果。

在最后一个例子中，由 o 引用的已装箱形式的 Point 被转型为一个 IChangeBoxedPoint。这里没有装箱的必要，因为 o 已经是一个已装箱的 Point 了。然后当调用 Change 时，它将改变已装箱的 Point 中的 x 字段和 y 字段。是的，接口方法 Change 允许我们改变一个已装箱 Point 对象中的字段！这样当再次调用 WriteLine 时，它将显示“(5, 5)”，这正是我们期望的结果。

这里整个代码示例的目的是向大家展示如何利用接口方法来改变一个已装箱值类型中的字段。在 C#中，没有接口方法是做不到这一点的。

重要 许多开发人员都对本书的很多章节有过反馈。在他们读了一些代码示例(如上面演示的代码)后，他们告诉我他们已经决定不再使用值类型了。必须承认值类型和引用类型之间的一些差别也曾经耗去我好几天的调试时间，这也是我为什么在本书中花费笔墨解释它们的原因。我希望大家能够记住这些差别，这样在代码受到它们的影响时就有所准备。当然，大家不应该对使用值类型感到恐惧。它们仍是很有用的类型，程序中总有它们的用武之地。毕竟，一个程序总是要时不时地使用一些 Int32。我们需要记住的是根据它们使用情况的不同，值类型和引用类型会有一些不同的行为。实际上，我们应该在前面的代码中将 Point 声明为一个 class 而不是 struct 来得到期望的结果。

15.4 实现多个有相同方法的接口

定义实现一个接口的类型通常是很容易的。我们需要做的仅仅是在类型中实现方法使其匹配接口中定义的方法和签名即可。如前所述，我们还应该将这些方法定义为公有方法，至于接口方法内部的实现代码则没有任何特殊的要求。调用一个接口方法也相当容易，这和调用定义在一个类型中的其他方法没什么两样。

但是有时候我们也会发现一个类型需要实现多个接口，而碰巧它们的方法又有着相同的名称和签名。例如，考虑下面两个接口定义：

```
public interface IWindow {
    Object GetMenu();
}

public interface IRestaurant {
    Object GetMenu();
}
```

假设我们希望定义一个类型来同时实现上面两个接口。我们的实现代码可能看起来就像下面这样：

```
// 该类型继承自 System.Object 并实
// 现了 IWindow 和 IRestaurant 接口
public class GiuseppePizzaria : IWindow, IRestaurant {
    // 该方法包含了 IWindow 接口的 GetMenu 方法实现
    Object IWindow.GetMenu() { ... }

    // 该方法包含了 IRestaurant's 接口的 GetMenu 方法实现
    Object IRestaurant.GetMenu() { ... }

    // 这个 GetMenu 方法和接口没有任何关系
    public Object GetMenu() { ... }
}
```

因为 GiuseppePizzaria 类型实现了多个 GetMenu 方法，所以我们需要告诉 C#编译器哪个 GetMenu 方法实现了哪个接口。在 C#中，我们通过在方法名前添加接口名称来告诉 C#编译器这一点。所以在上面的例子中，IWindow.GetMenu 方法会告诉编译器它实现了 IWindow 的 GetMenu 方法，而 IRestaurant.GetMenu 方法会告诉编译器它实现了 IRestaurant 的 GetMenu 方法。而没有任何限定名的 GetMenu 方法则仅仅是 GiuseppePizzaria 类型中定义的一个普通方法而已，它与接口没有任何关系。(译注：但是如果我们将 IRestaurant.GetMenu 这一接口实现方法删除，没有任何限定名的 GetMenu 将成为 IRestaurant 接口的实现方法。C#编译器在辨析接口成员实现时，会按照“先完全限定接口成员，后非完全限定接口成员”的顺序进行辨析。)

注意上面的完全限定接口方法没有被声明为 public，不能这样做的原因是这些方法有着双重身份：它们有时候为公有方法，有时候又为私有方法。看下面的代码：

```
static void SomeMethod() {
    // 构造一个类型实例
    GiuseppePizzaria gp = new GiuseppePizzaria ();

    Object menu;

    // 调用公有的 GetMenu 方法。使用 GiuseppePizzaria 引用，
    // 完全限定接口方法将为私有方法，因此不可能被调用
    menu = gp.GetMenu();

    // 调用 IWindow 的 GetMenu 方法。使用 IWindow 引用，
    // 因此只有 IWindow.GetMenu 方法被调用
    menu = ((IWindow) gp).GetMenu();

    // 调用 IRestaurant 的 GetMenu 方法。使用 IRestaurant 引用，
    // 因此只有 IRestaurant.GetMenu 方法被调用
    menu = ((IRestaurant) gp).GetMenu();
}
```

当我们在一个类型中用完全限定接口名来定义一个接口方法时，该方法将被认为是私有方法，因此我们不能使用类型本身的引用来调用它。但是，当我们将该类型的引用转型为一个接口时，该接口中定义的方法将可以被调用，这时它又成为一个公有方法。所以，当我们将 gp 变量转型为一个 IWindow 时，IWindow.GetMenu 方法将是惟一可以调用的方法。同样地，IRestaurant.GetMenu 方法也有这样的行为。

如前所述，一个类型很少实现多个定义有相同方法的接口，所以我们一般不会用到像上面代码中那样的做法。但是，这种用接口名称来限定一个方法的技巧却很有趣，并且还有其他的用处。实际上，下一节的重点就是这项技巧。

15.5 显式接口成员实现

接口非常的了不起，因为它们为类型间的通信定义了一种标准的方式。然而，这种灵活性伴随有编译时类型安全的代价，因为大多数接口方法都只接受类型为 System.Object 的参数、或者返回一个类型为 System.Object 的值。我们来看一下十分常用的 IComparable 接口：

```
public interface IComparable {
   Int32 CompareTo(Object other);
}
```

该接口定义的方法接受一个 System.Object 类型的参数。如果我们定义了一个类型来实现该接口，那么类型定义可能会像下面这样：

```
struct SomeValueType : IComparable {
   private Int32 x;
   public SomeValueType(Int32 x) { this.x = x; }
   public Int32 CompareTo(Object other) {
      return(x - ((SomeValueType) other).x);
   }
}
```

使用 SomeValueType 类型，我们可以编写下面的代码：

```
static void Main() {
   SomeValueType v = new SomeValueType(0);
   Object o = new Object();
   Int32 n = v.CompareTo(o);
}
```

在上面的代码中，我们比较了两个互不相干的对象：SomeValueType 和 Object。我们知道这段代码是没有多大意义的，因为 Object 没有 x 字段。但是，编译器不会检测到这样的逻辑缺陷，它会顺利编译，而不会报告任何警告或者错误。但是在运行时，当调用 CompareTo 方法的时候，在 other 被转型为 SomeValueType 的地方会抛出一个 InvalidCastException 异常。

相较于运行时错误，开发人员总是更喜欢编译时错误。实际上，CLR 确实提供了一种称作显式接口成员实现(explicit interface member implementation)的技术可以将上面的运行时错误变为编译时错误。下面使用显式接口成员实现对 SomeValueType 类型进行了一些修改：

```
struct SomeValueType : IComparable {

    private Int32 x;
    public SomeValueType(Int32 x) { this.x = x; }

    public Int32 CompareTo(SomeValueType other) {
        return(x - other.x);
    }

    // 注意：该方法没有标记 public/protected
    Int32 IComparable.CompareTo(Object other) {
        return CompareTo((SomeValueType) other);
    }
}
```

对于这个新版的 SomeValueType，我们需要注意几个地方。首先，它现在有两个 CompareTo 方法。第一个 CompareTo 方法是一个公有方法，它不再接受 Object 作为参数；相反它接受一个 SomeValueType。因为这个参数改变了，所以也就没有必要再将 other 转型为 SomeValueType，转型操作的代码自然也被删除了。代码在变得更加容易维护的同时，还获得了编译时类型安全。下面的代码演示了一些用法：

```
SomeValueType v1 = new SomeValueType(1);
SomeValueType v2 = new SomeValueType(2);
Int32 n;

n = v1.CompareTo(new Object());        // 编译时错误
n = v1.CompareTo(v2);                  // 调用 CompareTo
```

改变第一个 CompareTo 方法在获得类型安全的同时也意味着 SomeValueType 不再遵循 IComparable 接口的实现合同了。因此 SomeValueType 必须实现一个 CompareTo 方法来满足 IComparable 接口合同。这正是第二个方法 IComparable.CompareTo(一个显式接口成员实现)的工作。

IComparable.CompareTo 方法接受一个 System.Object，并返回一个 Int32，和 IComparable 接口中定义的方法一样。但是，我们要对 CompareTo 方法的三个特点有所认识。首先，该方法的名字前面添加了一个接口限定名：IComparable.CompareTo。它的作用很重要。它告诉 CLR 只有在使用一个 IComparable 的对象引用时方法 IComparable.CompareTo 才应该被调用。为了帮助大家清晰地理解这里的约束，我们来看下面的代码：

```
SomeValueType v1 = new SomeValueType(1);
SomeValueType v2 = new SomeValueType(2);
Int32 n;

n = v1.CompareTo(v2);              // 调用 CompareTo
n = v2.CompareTo(v1);              // 调用 CompareTo

// 注意：获取 IComparable 引用将导致 v1 被装箱
IComparable comparable = (IComparable) v1;

// 注意：下面将调用 IComparable.CompareTo，其
// 接受一个 Object 作为参数，因此会强制 v2 装箱
n = comparable.CompareTo(v2);
```

其次，IComparable.CompareTo 的实现是在将 other 转型为 SomeValueType 后，通过调用 CompareTo 方法来完成的。现有的代码得到了利用。

最后需要注意的是，IComparable.CompareTo 方法的前面没有标识任何 public 或者 protected 访问修饰符。实际上，如果我们添加了 public 或者 protected 访问修饰符，C#编译器将会产生下面的错误：

```
error CS0106: 修饰符“public”对该项无效
error CS0106: 修饰符“protected”对该项无效
```

IComparable.CompareTo 方法前面不能标识 public 或者 protected 访问修饰符的事实可以为我们提供许多有关显式接口方法实现的信息。当我们使用 SomeValueType 实例时，Comparable.CompareTo 方法是一个私有方法，因此不可能被外部调用。在前面的代码中，惟一能通过 v1 和 v2 来访问的 CompareTo 方法是接受 SomeValueType 类型作为参数、类型安全的那个版本的方法。

但是，如果我们有一个 IComparable 对象引用，IComparable.CompareTo 方法便成为一个公有方法。实际上，它是这时候惟一可以访问的方法，因为一个 IComparable 引用变量只可以用来调用 IComparable 接口中的方法。

这样，显式接口成员实现 IComparable.CompareTo 有时候成为一个私有方法，有时候又成为一个公有方法——这就是 C#编译器不允许我们为 IComparable.CompareTo 方法显式提供访问修饰符的原因。

显式接口成员实现为应用程序开发提供了更多的类型安全。虽然上面的例子使用的是值类型，但这种机制也可以用于引用类型来提高类型安全。除了可以提高类型安全外，当一个显式接口成员实现被定义在一个值类型上时，我们还会获得额外的好处。SomeValueType 类型碰巧就是这种情况。

因为 IComparable 的 CompareTo 方法接受一个 Object 作为参数，所以当我们给该参数传递一个值类型实例时，它就必须首先被执行装箱。我们知道，装箱操作会从托管堆上分配内存，然后还要拷贝字段，从而会损伤系统性能，所以我们应该尽可能地避免它。幸运的是，显式接口成员实现可以帮助我们避免这种多余的装箱操作。看下面的代码：

```
public static void Main() {
    SomeValueType v1 = new SomeValueType(1);
    SomeValueType v2 = new SomeValueType(2);
    Int32 n;

    n = v1.CompareTo(v2);                    // 没有任何装箱操作
    n = ((IComparable) v1).CompareTo(v2);    // v1 和 v2 都被装箱
}
```

在这段代码中，第 1 次访问 CompareTo 会调用类型安全的那个 CompareTo 方法。因为该方法不是接口的一部分，所以我们可以在未装箱实例 v1 上调用它。另外，因为它接受一个 SomeValueType 作为参数，所以 v2 也不必被装箱，可以直接传递给方法。

在第 2 次调用 CompareTo 时，我们首先将 v1 转型为 IComparable。这会对 v1 执行装箱操作，然后返回一个 IComparable 引用，后面我们会用它来调用 IComparable.CompareTo 方法。这个版本的 CompareTo 是一个显式接口方法实现，它接受一个 System.Object 作为参数。所以 v2 也必须在调用 IComparable.CompareTo 方法之前被执行装箱。这里出现了两次装箱操作！

显式接口方法实现这一技巧在实现一些接口(如 ICloneable、IComparable、ICollection、IList 和 IDictionary)时经常被用到。它们不仅为这些接口方法提供了更多的类型安全，并且还可以减少值类型的装箱操作。

谨慎使用显式接口方法实现

当在.NET框架参考文档中查看一个类型的方法时，我们不会看到其中的显式接口方法实现。这会使很多开发人员感到困惑。例如，如果我们在参考文档中查找System.Int32类型，我们将会看到Int32实现了IConvertible接口。但是，如果我们再查看“Int32成员”帮助页面，我们将看不到IConvertible中定义的方法(ToBoolean、ToByte、ToChar、ToSingle等)。为什么呢?

当.NET框架还在beta版本时，微软注意到一个Int32类型就提供了近20个方法，他们担心开发人员在查找一个“简单的Int32类型”时会很快被淹没在其中。所以微软决定不显示其中的显式接口方法实现，从而减少一点文档的混乱状况。然后，他们又不得不将一些类型中的方法改变为显式接口方法实现，以此来隐藏这些方法。对于Int32类型来说，他们选择将实现了IConvertible接口的那些方法定义为显式接口方法实现。因为IConvertible接口定义了15个方法(译注：实际上IConvertible接口定义了17个方法，而Int32类型只将其中的15个方法定义为显式接口方法实现)，而Int32类型的文档现在只显示5个方法。这当然减少了一点帮助页面的混乱，并使得Int32类型看起来更像一个“简单类型”。

然而，我遇到过很多开发人员都为这种决定感到困惑。他们看到Int32类型实现了IConvertible接口，但文档中却没有反映出这些方法的存在。而且更糟糕的是，我们不能直接在一个Int32上调用IConvertible接口中定义的方法。例如，下面的代码将不能通过编译:

```
static void Main() {
    Int32 x = 5;
    Single s = x.ToSingle(null);
}
```

当编译该方法时，C#编译器将产生以下错误：“error CS0117: ‘int’ 并不包含对‘ToSingle’ 的定义”。这种错误信息使得开发人员更加困惑，因为它明确地声称Int32类型没有定义ToSingle方法，而事实上它确实定义了。

谨慎使用显式接口方法实现(续)

为了在 Int32 上调用 ToSingle 方法，我们必须首先将 Int32 转型为一个 IConvertible 接口，如下面的代码所示：

```
static void Main() {
    Int32 x = 5;
    Single s = ((IConvertible) x).ToSingle(null);
}
```

这样的转型要求并不明显，许多开发人员都不大可能会想到这一点。但是还有一个更令人烦恼的问题：将 Int32 值类型转型为一个 IConvertible 接口将导致该值类型被执行装箱，既浪费了内存，也损伤了性能。

真的有必要用这样的方法来减少文档的混乱吗？我个人不这么认为。微软应该让文档独立，它应该是精确的、完整的、不易令人产生困惑的，并且代码编写起来要清晰和方便，执行起来也要高效。这里的例子仅仅谈的是 Int32 类型是如何实现 IConvertible 接口方法的。实际上，微软为很多实现了各种接口的类型都选择了“隐藏文档”。

这里的讨论向大家清楚地展示了应该谨慎地使用显式接口方法实现。在许多开发人员掌握了显式接口方法实现这一技巧后，他们便认为这非常的酷，并且一有可能就使用它们。不要这么做！显式接口方法实现在某些情况下的确很有用，但是我们应该避免任何可能的时候都去使用它们，因为它们有时候会使类型使用起来不是很方便。

16

定制特性

本章讨论 Microsoft .NET 框架提供的最具创新的一种构造：**定制特性**(custom attribute)。任何人(不只是微软)都可以使用定制特性来定义一些信息，并将这些信息应用于几乎所有的元数据表项上，然后在运行时通过查询这些可扩展的元数据信息来动态地改变代码的执行方式。在我们使用各种.NET 框架技术(Windows 窗体、Web 窗体、XML Web 服务等)的时候，我们将会看到它们都广泛地利用了定制特性来帮助开发人员方便地表达他们的意图。深刻理解定制特性对于任何.NET 框架开发人员来说都是很有必要的。

16.1 使用定制特性

特性(如 public、private、static 等)可以被应用于各种类型和成员。相信大家都同意使用这些特性有很多好处，但是如果我们能够定义自己的特性不是更好吗？例如，如果我们定义了一个类型，那么我们便可以用一种特性来表示该类型可以通过序列化的方式进行远程传送。或者，我们也可以将一种特性应用到一个方法上来表示在方法调用之前必须经过某种安全许可。

创建用户定义的特性、并将它们应用于类型和方法上对于编程工作显然是一件非常美妙和方便的事情。但是这要求编译器首先能够识别这些特性，然后它才能将这些特性信息存放在生成的元数据中。因为编译器厂商通常不会发布编译器的源代码，所以微软采取了另外一种方式来允许开发人员使用这种用户定义的特性。这种机制称作**定制特性**，它可以在应用程序的设计时和运行时为我们提供非常强大的功能。任何人都可以定义并使用定制特性，并且所有面向通用语言运行时(CLR)的编译器都必须能够识别定制特性，并将它们存放在生成的元数据中。

关于定制特性，我们首先需要知道的是它们仅仅是为目标元素提供关联附加信息的一种方式。编译器的工作只是将这些附加的信息存放在托管模块的元数据中而已。大多数特性对于编译器没有任何意义。编译器仅仅只是检测源代码中的定制特性，然后产生相应的元数据。

.NET 框架类库(FCL)发布的时候已经带有很多预定义特性。例如，System.FlagsAttribute 特性允许我们将一个枚举类型看作一组位标记，System.SerializableAttribute 特性允许一个类型的字段被序列化和反序列化(典型地用于方法参数和返回值的远程传送)，几个安全相关的特性可以确保方法在试图进行某种特殊访问的时候具有要求的特权级别，许多互操作相关的特性可以帮助托管代码调用非托管代码，等等。

下面的 C#代码中应用了许多特性。目前大家还没有必要完全理解这些代码的行为，这里仅仅是希望使大家对定制特性有一个初步的印象。

```
[StructLayout(LayoutKind.Sequential, CharSet=CharSet.Auto)]
class OSVERSIONINFO {
   public OSVERSIONINFO() {
      OSVersionInfoSize = (UInt32) Marshal.SizeOf(this);
   }
   public UInt32 OSVersionInfoSize = 0;
   public UInt32 MajorVersion = 0;
   public UInt32 MinorVersion = 0;
   public UInt32 BuildNumber = 0;
   public UInt32 PlatformId = 0;

   [MarshalAs(UnmanagedType.ByValTStr, SizeConst=128)]
   public String CSDVersion = null;
}

class MyClass {
   [DllImport("Kernel32", CharSet=CharSet.Auto, SetLastError=true)]
   public static extern Boolean GetVersionEx(
      [In, Out] OSVERSIONINFO ver);
}
```

在 C#中，将定制特性放在紧挨着目标元素前的一个方括号([,])中，就表示将该定制特性应用到目标元素上了。在上面的例子中，OSVERSIONINFO 类上应用了 StructLayout 特性，CSDVersion 字段上应用了 MarshalAs 特性，GetVersionEx 方法上应用了 DllImport 特性，GetVersionEx 的 ver 参数上应用了 In 和 Out 两个特性。每个编程语言都为开发人员在一个目标元素上应用定制特性定义了自己的语法。例如 Visual Basic 要求使用尖括号(<,>)而不是方括号。

CLR 允许将特性应用于任何可以在一个文件的元数据中表示的元素。特性经常被应用于以下一些元数据定义表中的条目上：TypeDef(类、结构、枚举、接口、委托)、MethodDef(包括构造器)、ParamDef、FieldDef、PropertyDef、EventDef、AssemblyDef，以及 ModuleDef。虽然很少见，但是特性也可以应用到元数据引用表中的条目上，例如 AssemblyRef、ModuleRef、TypeRef，以及 MemberRef。最后，定制特性也可以应用到其他一些元数据中，例如安全许可、导出类型以及资源等。

虽然 CLR 允许将定制特性应用于任何上述这些条目上，但是大多数语言只允许将它们应用于各种元数据定义表中的条目上。微软的 C#编译器就是这样。具体而言，C#只允许我们将特性应用于定义了以下构造的源代码中：程序集、模块、类型、字段、方法、方法参数、方法返回值、属性，以及事件。

当我们应用一个特性时，C#允许我们指定一个前缀来表示特性所应用的目标元素。下面的代码演示了所有可能的前缀。在许多情况下，如果我们省去前缀，编译器仍会判断出特性所应用的目标元素(比如前面的例子就是这样的情况)。但是，显式指定前缀会消除可能的二义性。

```
using System;

[assembly: MyAttribute(1)]                // 应用于程序集上
[module: MyAttribute(2)]                  // 应用于模块上

[type: MyAttribute(3)]                    // 应用于类型上
class SomeType {

    [property: MyAttribute(4)]            // 应用于属性上
    public String SomeProp { get { return null; } }

    [event: MyAttribute(5)]               // 应用于事件上
    public event EventHandler SomeEvent;

    [field: MyAttribute(6)]               // 应用于字段上
    public Int32 SomeField = 0;

    [return: MyAttribute(7)]              // 应用于返回值上
    [method: MyAttribute(8)]              // 应用于方法上
    public Int32 SomeMethod(
        [param: MyAttribute(9)]           // 应用于参数上
        Int32 SomeParam) { return SomeParam; }
}
```

既然我们已经知道了怎样应用一个定制特性，下面我们来仔细探究一下特性到底是什么。实际上，一个特性仅仅是一个类型的实例。要与通用语言规范(CLS)兼容，定制特性的类型必须直接或间接继承自 System.Attribute。C#只允许使用与 CLS 兼容的定制特性。通过查看.NET 框架 SDK 文档，我们会看到其中定义有以下类型(对应于前面的例子)：StructLayoutAttribute、MarshalAsAttribute、DllImportAttribute、InAttribute 和 OutAttribute。所有这些类型碰巧都定义在 System.Runtime.InteropServices 命名空间中，但是定制特性的类型可以定义在任何命名空间中。如果再进一步观察，大家会注意到这些类型都继承自 System.Attribute，这是所有与 CLS 兼容的特性类型都必须遵守的约定。

如前所述，一个特性就是类型的一个实例。该类型必须有一个公有构造器来创建它的实例。所以当我们在一个目标元素上应用一个特性时，其语法类似于调用类型的实例构造器。另外，一些语言可能允许我们使用一些特殊的语法来设置特性类型的公有字段或属性。我们来看一个例子。回想前面将 DllImport 特性应用到 GetVersionEx 方法上的那个应用程序：

```
[DllImport("Kernel32", CharSet=CharSet.Auto, SetLastError=true)]
```

上面一行的语法看起来很奇怪，因为大家可能从来没有使用过这样的方式来调用一个构造器。如果我们查看 DllImportAttribute 类型的文档，我们会发现它的构造器只要求一个 String 参数。在上面的例子中，“Kernel32”被作为这样的参数传递进去。一个特性类型的构造器参数被称为**定位参数**(positional parameter)，该参数是强制性的，也就是说在应用特性时必须指定这样的参数。

那么其他两个“参数”呢？上面那种特殊的语法允许我们在 DllImportAttribute 对象构造之后设置它的公有字段或属性。在上面的例子中，当 DllImportAttribute 对象被构造时，实际上发生了 3 件事情：“Kernel32”被传入 DllImportAttribute 构造器，CharSet 被设为 CharSet.Auto，SetLastError 被设为 true。设置字段或属性的“参数”被称作**命名参数**(named parameter)，并且它们是可选的。也就是说，当我们应用一个特性的实例时，并非一定要指定这样的参数。稍后本章会解释什么会导致 DllImportAttribute 类型的实例被构造。

另外注意我们可以在一个目标元素上应用多个不同的特性。例如，GetVersionEx 方法的 ver 参数就应用了 In 和 Out 两个特性。当在一个目标元素上应用多个特性时，它们的顺序可以任意。在 C#中，我们可以将每一个特性放在一个方括号中，或者将多个特性用逗号分割、然后放在一个方括号中。特性名称中的后缀 Attribute 是可选的。如果特性类型的构造器不接受任何参数，那么小括号也是可选的。下面几行代码向大家演示了声明多个特性所有可能的方式，它们的行为都是相同的：

```
[Serializable][Flags]
[Serializable, Flags]
[FlagsAttribute, SerializableAttribute]
[FlagsAttribute()][Serializable()]
```

16.2 定义自己的特性

我们已经知道特性是一个继承自 System.Attribute 的类型，并且我们也知道怎样应用一个特性。下面我们来看一下怎样定义自己的定制特性。假设我们是微软的员工，负责为枚举类型添加位标记支持。要实现这一点，我们首先需要定义一个 FlagsAttribute 类型：

```
namespace System {
   public class FlagsAttribute : System.Attribute {
      public FlagsAttribute() {
      }
   }
}
```

注意 FlagsAttribute 类型继承自 System.Attribute，这使得 FlagsAttribute 类型成为一个与 CLS 兼容的定制特性。另外，所有的非抽象特性都必须具有 public 访问限制，并且根据约定，所有特性类型的名称都应该有一个“Attribute”后缀。最后，所有的非抽象特性都必须包含至少一个公有构造器。上面的 FlagsAttribute 类型的构造器没有任何参数，当然也没有做任何事情。

现在，我们定义的 FlagsAttribute 类的实例就可以应用到任何目标元素上了，但是该特性应该只允许应用于枚举类型上，将其应用于一个属性或者方法上是没有任何意义的。为了告诉编译器哪些地方可以应用该特性，我们可以在我们定义的特性类型上应用一个 System.AttributeUsageAttribute 类的实例。下面是修改后的代码：

```
namespace System {
   [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
   public class FlagsAttribute : System.Attribute {
      public FlagsAttribute() {
      }
   }
}
```

在修改后的新版本中，我们在定义的特性类型上应用了一个 AttributeUsageAttribute 实例。毕竟，我们定义的特性类型本身就是一个类，而对于一个类我们自然可以在其上应用特性。AttributeUsageAttribute 特性允许我们告诉编译器我们自己定义的定制特性可以应用在哪些地方。所有的编译器都对 AttributeUsageAttribute 特性有内置的支持，它们在用户定义的定制特性应用到了一个无效的目标元素上时会报告错误。在上面的例子中，AttributeUsage 特性表示 Flags 特性实例只可以应用于枚举类型上。

由于所有的特性也都只是类型，所以我们可以很容易理解 AttributeUsageAttribute 类型。下面是 FCL 中 AttributeUsageAttribute 类型的实现源码：

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
[Serializable]
public sealed class AttributeUsageAttribute : Attribute {
   internal AttributeTargets m_attributeTarget = AttributeTargets.All;
   internal Boolean m_allowMultiple = false;
   internal Boolean m_inherited = true;

   public AttributeUsageAttribute(AttributeTargets validOn) {
      m_attributeTarget = validOn;
   }

   public AttributeTargets ValidOn {
      get { return m_attributeTarget; }
   }

   public Boolean AllowMultiple {
      get { return m_allowMultiple; }
      set { m_allowMultiple = value; }
   }

   public Boolean Inherited {
      get { return m_inherited; }
      set { m_inherited = value; }
   }
}
```

如我们所见，AttributeUsageAttribute 类型有一个构造器，它允许我们传递位标记来表示我们的特性可以应用的地方。System.AttributeTargets 枚举类型在 FCL 中的定义如下：

```
[Flags, Serializable]
public enum AttributeTargets {
    Assembly     = 0x0001,
    Module       = 0x0002,
    Class        = 0x0004,
    Struct       = 0x0008,
    Enum         = 0x0010,
    Constructor  = 0x0020,
    Method       = 0x0040,
    Property     = 0x0080,
    Field        = 0x0100,
    Event        = 0x0200,
    Interface    = 0x0400,
    Parameter    = 0x0800,
    Delegate     = 0x1000,
    ReturnValue  = 0x2000,
    All          = Assembly   | Module     | Class  | Struct     | Enum   |
              Constructor | Method     | Property | Field     | Event  |
              Interface   | Parameter  | Delegate | ReturnValue
}
```

AttributeUsageAttribute 类另外提供有两个公有属性：AllowMultiple 和 Inherited，在我们将该特性应用于一个特性类型时，我们可以设置它们的值，当然这样的设置是可选的。

对于大多数特性来讲，将它们多次应用于同一个目标元素上是没有意义的。例如，将 Flags 或 Serializable 特性多次应用于同一个目标元素上不会有任何效果。实际上，如果我们试图编译下面的代码，编译器将报告以下错误："error CS0579: 重复 "Flags" 属性"。

```
[Flags][Flags]
enum Color {
    Red
}
```

但是，对于少数一些特性，将它们多次应用于同一个目标元素上还是有意义的。在 FCL 中，ConditionalAttribute 特性类型以及许多与安全许可相关的特性类型（例如 EnvironmentPermissionAttribute、FileIOPermissionAttribute、ReflectionPermissionAttribute、Registry PermissionAttribute 等）都允许将它们的多个实例应用于同一个目标元素上。如果我们没有显式设置 AllowMultiple，我们定义的特性将获得默认的行为，也就是只允许将其实例在同一个目标元素上应用一次。

AttributeUsageAttribute的另外一个属性Inherited，表示特性是否应该应用于派生类或派生方法上。在FCL中定义的所有特性中，只有不到5个特性的Inherited属性被设为true。下面的代码演示了个特性的继承含义：

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
    Inherited=true)]
class TastyAttribute : Attribute {
}

[Tasty][Serializable]
public class SomeType {

    [Tasty] public virtual void DoSomething() { }
}

public class AnotherType : SomeType {
     public override void DoSomething() { }
}
```

在上面的代码中，AnotherType和它的DoSomething方法都被认为应用了Tasty，这是因为Tasty特性的Inherited属性被设为true。但是，AnotherType并不被认为是可序列化的，因为FCL中SerializableAttribute类型的Inherited属性被设为false。

注意.NET框架仅考虑类、方法、属性、事件、参数是否可以继承。所以当我们定义一个特性类型时，只有当我们的目标元素是上述这些构造时，我们才应该将Inherited设为true。注意可继承的特性不会导致在派生类型所在的托管模块中存放额外的元数据。16.4节会对此有更详细的讨论。

注意 如果我们定义了自己的特性类，并且忘记了在类上应用AttributeUsage特性，那么编译器和CLR将对我们的特性类做以下3点假设：它可以应用于所有的目标元素；它在一个目标元素上只可以应用一次；它可以被继承。这些假设实际上参照的是AttributeUsageAttribute类型中字段的默认值。

16.3　特性构造器与字段/属性的数据类型

当我们定义自己的特性类型时，我们可以定义一个带有参数的构造器，这些参数是开发人员在应用我们的特性类型实例时必须指定的。另外，我们还可以在特性类型中定义非静态的公有字段和属性，它们标识一些开发人员在应用特性类型实例时可以有选择地指定的一些设置。

当定义一个特性类型的实例构造器、字段和属性时，它们的类型被严格地限制在一个很小的子集中。具体而言，合法的数据类型集合仅限于以下类型：Boolean、Char、Byte、SByte、Int16、UInt16、Int32、UInt32、Int64、UInt64、Single、Double、String、Type、Object、或者一个枚举类型。另外，我们也可以使用任何这些类型的一维0基数组。

当应用一个特性时，我们必须传递一个和特性类中定义的类型相匹配的编译时常数表达式。在我们定义Type构造器参数、Type字段、或者Type属性的地方，我们必须像下面的代码中那样使用C#的typeof操作符。在我们定义Object构造器参数、Object字段、或者Object属性的地方，我们可以传递一个Int32、一个String，或者任何其他的常数表达式。如果常数表达式表示的是一个值类型，该值类型在运行时特性实例被构造的时候会被执行装箱。

下面的例子演示了一个特性和与它相关的一个应用(译注：下面的代码中使用了Color.Red来作为SomeAttribute的第二个参数。作者这里假定Color是一个自己定义的枚举类型。读者不要使用System.Drawing命名空间中定义的Color类型，因为它是一个结构类型，如果使用它会导致编译出错)：

```
[AttributeUsage(AttributeTargets.All)]
class SomeAttribute : Attribute {
    public SomeAttribute(String name, Object o, Type[] types) {
        // 'name' 指向一个String。
        // 'o' 指向任何一个合法类型(必要的时候执行装箱)。
        // 'types' 指向一个一维0基的Type对象数组
        ...
    }
}

[Some("Jeff", Color.Red, new Type[] { typeof(Math), typeof(Console) })]
public class SomeType {
   ...
}
```

从逻辑上来看，当编译器检测到一个目标元素上应用了定制特性时，它会调用该特性的构造器并传入指定的参数来构造特性类型的实例。随后编译器会初始化所有指定的公有字段和属性。在对定制特性对象完成这些初始化工作之后，编译器就会将其序列化到目标元素的元数据表内对应的条目中。

重要 我发现有一种方式可以很好地帮助大家来理解定制特性，我们可以把定制特性看作是一些被序列化为字节流的类型实例，这些字节流在编译时被存储在生成模块的元数据中。到了运行时，CLR便通过对元数据中的字节流执行反序列化操作来构造特性类型的实例。

注意 每个参数的序列化方式为前面1个字节的类型ID，后面紧接着参数的值。在"序列化"完构造器参数之后，编译器开始存放每一个指定的字段和属性的值，这时的序列化方式为前面1个字节的类型ID，后面紧接着字段或属性的值。对于数组来说，最开始处是数组中元素的个数，其后紧接着是数组中的各个元素。

16.4 检测定制特性

定义一个特性类型本身没有什么用处。当然，我们可以定义我们自己喜欢的特性类型，并将它们应用在任何地方。但是，这只会导致额外的元数据被写入到托管模块中——应用程序代码的行为不会因此有任何改变。

在第13章中，我们看到了在枚举类型上应用Flags特性可以改变System.Enum的ToString、Format，以及Parse方法的行为。这些方法具有不同行为的原因是，它们会在运行时检测所操作的枚举类型上是否应用了Flags特性。应用程序代码可以利用一种称作反射(reflection)的技术来查找目标元素上应用了哪些特性。这里只对反射做一个简单的演示，第20章会对这一技术进行详细的讨论。

如果我们是微软公司负责实现Enum类型中Format方法的员工，我们可能会像下面这样来实现它：

```
public static String Format(Type enumType, Object value, String format) {

    // 检测传入的枚举类型是否应用
    // 了 FlagsAttribute 类型的实例
    if (enumType.IsDefined(typeof(FlagsAttribute), false)) {
        // 答案肯定：执行代码(将 value 看作一个位标记枚举类型)
        ...
    } else {
        // 答案否定：执行代码(将 value 看作一个一般的枚举类型)
        ...
    }
    ...
}
```

上面的代码调用了 Type 的 IsDefined 方法，它会查寻枚举类型的元数据来判断其上是否应用了 FlagsAttribute 特性。如果 IsDefined 返回 true，那么就可以判定该枚举类型上应用了 FlagsAttribute 特性，Format 方法将把 value 看作是一组位标记来对待。如果 IsDefined 返回 false，那么 Format 方法将把 value 看作是一个一般的枚举类型。

所以如果我们定义了自己的特性类型，我们还必须实现某些代码来检测目标元素上是否应用了这些特性类型的实例，然后根据检测结果执行相应的代码路径。只有这样，定制特性才有意义。

FCL 提供了许多检测特性是否存在的方式。如果我们通过一个 System.Type 对象来检测特性，那么我们可以使用前面演示的 IsDefined 方法。但是有时候我们可能希望执行特性检测的目标元素不是一个类型，而是一个程序集、一个模块或者一个方法。我们目前先把讨论集中在 System.Attribute 类型提供的方法上。大家应该还记得所有与 CLS 兼容的特性都继承自 System.Attribute，该类型定义了 3 个静态方法来获取应用于目标元素上的特性：IsDefined，GetCustomAttributes，以及 GetCustomAttribute。每个方法都有好几个重载版本。例如，对于下列每种构造，几个方法都有一个在其上进行操作的版本：类型及类型成员(类、结构、枚举、接口、委托、构造器、方法、属性、字段、事件和返回值类型)、参数、模块以及程序集。其中一些版本还允许我们上溯继承体系，将继承特性也包括在结果之内。表 16.1 简要描述了每个方法所做的工作。基本上而言，这些方法都是通过在元数据上执行某种反射操作来查询与 CLS 兼容的定制特性。

如果我们只是想查看一下一个目标元素上是否应用了某个特性，我们应该调用 IsDefined，因为它的效率要比其他两个方法高许多。我们知道当在一个目标元素上应用特性时，我们可以指定特性构造器的参数，也可以有选择地设置某些字段和属性。使用 IsDefined 不会导致构造特性对象、调用构造器、或者设置它的字段和属性。

如果我们希望构造一个特性对象，那么我们必须调用 GetCustomAttributes 或 GetCustomAttribute。每次调用这两个方法时，它们都会构造指定特性类型的实例，并会根据源代码中指定的值设置每个实例的字段和属性。这两个方法的返回值将为完全构造好的特性实例的引用。

当我们调用这些方法时，它们会在内部扫描托管模块的元数据，执行一些字符串比较来查找指定的特性类。很明显，这样的操作要花费一些时间。如果大家比较关心性能，则可以考虑缓存这些方法的执行结果，而不是重复调用它们去查询同样的信息。

表 16.1 System.Attribute 类的一些方法

方 法	描 述
IsDefined	如果至少有一个指定的 Attribute 的派生类型应用在目标元素上，该方法就会返回 true。该方法执行速度比较快，因为它不需要构造(反序列化)任何特性类型的实例
GetCustomAttributes	该方法返回一个数组，其中的元素是指定的、应用在目标元素上的特性实例。每个实例都会用编译期间指定的参数、字段和属性来构造(反序列化)。如果目标元素上没有应用指定的特性实例，方法将返回一个空数组。该方法通常用于那些将 AllowMultiple 设为 true 的特性
GetCustomAttribute	该方法返回一个指定的、应用在目标元素上的特性实例。返回的实例使用编译期间指定的参数、字段和属性来构造(反序列化)。如果目标元素上没有应用指定的特性实例，方法将返回一个 null。如果目标元素上应用了多个指定的特性实例，方法将抛出一个 System.Reflection.AmbiguousMatchException 异常。该方法通常用于那些将 AllowMultiple 设为 false 的特性

另外，System.Reflection 命名空间中定义的一些类型还允许我们查看一个模块中的元数据内容。这些类型有：Assembly、Module、Enum、ParameterInfo、MemberInfo、Type、MethodInfo、ConstructorInfo、FieldInfo、EventInfo、PropertyInfo 以及它们各自的*Builder 类型。所有这些类型也都提供了 IsDefined 和 GetCustomAttributes 方法。但只有 System.Attribute 提供了非常方便的 GetCustomAttribute 方法。

注意，由这些反射类型定义的 GetCustomAttributes 方法返回的是一个 Object 数组(Object[])，而非 Attribute 数组(Attribute[])。这是因为这些反射类型能够返回与 CLS 不兼容的特性类型的实例。大家不必对这种不一致有太多的担心，因为与 CLS 不兼容的特性类型非常少见。事实上，从我开始使用.NET 框架以来，还从来没有遇到过这样的情况。

注意　注意反射类型中的方法仅认为应用于类和方法上的特性才是可继承的。也就是说，如果我们用 EventInfo、PropertyInfo、或者 ParameterInfo 来调用 IsDefined 或 GetCustomAttributes 方法，它们将忽略传入的 inherit 参数，而假定其为 false。只有 Attribute 类型中的方法才会为事件、属性和参数考虑 inherit 参数。

另外还有一个问题需要大家注意：当我们向 IsDefined、GetCustomAttribute、或 GetCustomAttributes 传递一个类型时，这些方法会在应用程序中搜索我们指定的特性类型、及其所有的派生类型。如果我们的代码找到一个具体的特性类型，我们应该对返回值做一个额外的检查，以确保这些方法返回的类型就是我们正在搜索的类型。大家因此可能会考虑将自己的特性类型定义为密封类型，从而减少潜在的混淆并省去这种额外的检查。但必须承认，我个人目前看到的所有特性类型都是直接继承自 System.Attribute，所以也就不会遇到这样的问题。

下面的示例代码首先得到一个类型中定义的所有方法，然后显示每个方法上应用的特性。注意，代码只是做演示之用，通常情况下，我们不会像下面那样去应用这些特殊的定制特性。

```
using System;
using System.Diagnostics;
using System.Reflection;

[assembly:CLSCompliant(true)]

[Serializable]
[DefaultMemberAttribute("Main")]
class App {
   [Conditional("Debug")][Conditional("Release")]
   public void DoSomething() {}

   public App() {
   }

   [CLSCompliant(true)]
   [STAThread]
   public static void Main() {
      // 显示类型的名称
      Console.WriteLine("Attributes applied to: {0}", typeof(App));
```

```
        // 获取并显示应用于类型上的特性集合
        ShowAttributes(typeof(App).GetCustomAttributes(false));

        // 获取与类型相关的方法集合
        MemberInfo[] members = typeof(App).FindMembers(
           MemberTypes.Constructor | MemberTypes.Method,
           BindingFlags.DeclaredOnly | BindingFlags.Instance |
           BindingFlags.Public | BindingFlags.Static,
           Type.FilterName, "*");

        foreach (MemberInfo member in members) {
           // 显示类型的成员名称
           Console.WriteLine("Attributes applied to: {0}", member.Name);

           // 获取并显示应用于成员上的特性集合
           ShowAttributes(member.GetCustomAttributes(false));
        }
    }

    public static void ShowAttributes(Object[] attributes) {
       foreach (Object attribute in attributes) {
          // 显示每个特性的类型
          Console.Write(" {0}", attribute.GetType().ToString());
          if (attribute is ConditionalAttribute)
             Console.Write(" ({0})",
                ((ConditionalAttribute) attribute).ConditionString);

          if (attribute is CLSCompliantAttribute)
             Console.Write(" ({0})",
                ((CLSCompliantAttribute) attribute).IsCompliant);

          Console.WriteLine();
       }

       if (attributes.Length == 0)
          Console.WriteLine(" No attributes applied to this target.");

       Console.WriteLine();
    }
}
```

编译并运行上面的代码，我们将会看到以下输出：

```
Attributes applied to: App
   System.Reflection.DefaultMemberAttribute
```

```
Attributes applied to: DoSomething
   System.Diagnostics.ConditionalAttribute (Release)
   System.Diagnostics.ConditionalAttribute (Debug)

Attributes applied to: Main
   System.CLSCompliantAttribute (True)
   System.STAThreadAttribute

Attributes applied to: ShowAttributes
   No attributes applied to this target.

Attributes applied to: .ctor
   No attributes applied to this target.
```

16.5 特性实例间的匹配

既然我们已经知道了怎样查看一个目标元素上是否应用了某个特性实例，那么我们也可能希望查看特性实例的字段有着什么样的值。有一种实现方式是编写代码来查看特性实例中字段的值。但是，我们也可以在特性类型中重写 System.Attribute 的 Match 方法，然后在代码中构造一个特性类型的实例，并调用 Match 方法将其和应用于目标元素上的特性实例进行比较。下面的代码演示了这种做法：

```
using System;

[Flags]
public enum Accounts {
   Savings      = 0x0001,
   Checking     = 0x0002,
   Brokerage    = 0x0004
}

[AttributeUsage(AttributeTargets.Class)]
public class AccountsAttribute : Attribute {

   private Accounts accounts;

   public AccountsAttribute(Accounts accounts) {
      this.accounts = accounts;
   }

   public override Boolean Match(object obj) {
      // 如果基类实现了 Match，并且基类不是 Attribute，
      // 那么不要注释下面一行
      // if (!base.Match(obj)) return false;
```

```
        // 因为 'this' 不为 null，所以如果 obj 为 null，那么
        // 两个对象不可能匹配。注意：如果我们确信基类
        // 型正确地实现了 Match，那么可以删除下面一行
        if (obj == null) return false;

        // 如果两个对象的类型不同，那么它们不可能匹配
        // 注意：如果我们确信基类型正确地实现了 Match，
        // 那么可以删除下面一行
        if (this.GetType() != obj.GetType()) return false;

        // 将 obj 转型以便访问其中的字段。注意：转型不
        // 可能失败，因为我们已经知道两对象的类型相同
        AccountsAttribute other = (AccountsAttribute) obj;

        // 以任意合适的方式比较两个字段。这里检查 'this'
        // 的 accounts 是否为 other 的 accounts 的子集
        if ((other.accounts & accounts) != accounts)
            return false;

        return true;        // 两个对象匹配
    }

    public override Boolean Equals(object obj) {
        // 如果基类实现了 Equals，并且基类不是 Object，
        // 那么不要注释下面一行
        // if (!base.Equals(obj)) return false;

        // 因为 'this' 不为 null，所以如果 obj 为 null，那么
        // 两个对象不可能相等。注意：如果我们确信基类
        // 型正确地实现了 Equals，那么可以删除下面一行
        if (obj == null) return false;

        // 如果两个对象的类型不同，那么它们不可能相等
        // 注意：如果我们确信基类型正确地实现了 Equals，
        // 那么可以删除下面一行
        if (this.GetType() != obj.GetType()) return false;

        // 将 obj 转型以便访问其中的字段。注意：转型不
        // 可能失败，因为我们已经知道两对象的类型相同
        AccountsAttribute other = (AccountsAttribute) obj;

        // 比较两个字段看它们的值是否相同。这里检查 'this'
        // 的 accounts 是否和 other 的 accounts 相同
        if (other.accounts != accounts) return false;

        return true;            // 两个对象相等
    }
```

```
    // 因为我们重写了 Equals，所以也要重写 GetHashCode
    public override Int32 GetHashCode() {
        return (Int32) accounts;
    }
}

[Accounts(Accounts.Savings)]
class ChildAccount { }

[Accounts(Accounts.Savings | Accounts.Checking | Accounts.Brokerage)]
class AdultAccount { }

class App {
    static void Main() {

        CanWriteCheck(new ChildAccount());
        CanWriteCheck(new AdultAccount());

        // 下面一行仅仅为了演示我们的方法可以正确处理
        // 没有应用 AccountsAttribute 特性的类型
        CanWriteCheck(new App());
    }

    public static void CanWriteCheck(Object obj) {

        // 构造一个特性类型的实例，并将其初始化为我们
        // 希望查找的值
        Attribute checking = new AccountsAttribute(Accounts.Checking);

        // 构造应用于当前类型上的特性实例
        Attribute validAccounts = Attribute.GetCustomAttribute(
            obj.GetType(), typeof(AccountsAttribute), false);

        // 如果当前类型上应用了 Accounts 特性，并且
        // 该特性指定了“支票账号”，那么我们将可以从该
        // 账号上开出支票
        if ((validAccounts != null) && checking.Match(validAccounts)) {
            Console.WriteLine("{0} types can write checks.",
                obj.GetType());
        } else {
            Console.WriteLine("{0} types can NOT write checks.",
                obj.GetType());
        }
    }
}
```

编译并运行上面的代码，我们将会得到以下输出：

```
ChildAccount types can NOT write checks.
AdultAccount types can write checks.
App types can NOT write checks.
```

我们可以看到 Match 的实现代码和前面第 6 章中讨论过的 Equals 的实现代码非常相似。在这两个方法的实现中，我们都必须谨慎地进行转型。如果需要的话，我们还要在适当的地方调用基类的 Match 方法。如果我们定义了一个特性类型，而又没有重写 Match 方法，那么它将继承 Attribute 的 Match 方法实现。Attribute 的 Match 方法实现只是简单地调用 Equals。

16.6 伪定制特性

微软定义的某些特性应用的十分频繁，如果将所有这些特性信息都存放到元数据中将导致生成托管模块的大小急剧膨胀。因此，这些特性在编译时都经过了特殊的处理，它们将以位的形式存放在元数据中。CLR 和 FCL 也知道怎样在元数据中以一种特殊的方式来查找这些**伪定制特性**(pseudo-custom attribute)。看下面的代码：

```
[Serializable]
class SomeType {
    ...
}
```

FCL 提供了一个 System.SerializableAttribute 类型，在上面的代码中，我们将该类型的一个实例应用在了 SomeType 上。因为 Serializable 特性很常用，所以编译器会将它以一个位的形式编译到元数据中，而不必将 Serializable 特性实例的所有元数据都编译进去。这就是我们称这些特性为伪定制特性的缘故：它们看起来和通常的特性很像，并且它们在源代码中的应用方式也和通常的特性一样，但是它们是以一种高度压缩的形式(一个位)来存储的。

关于伪定制特性最重要的一点就是我们不能在运行时像检测通常的定制特性那样的方式来检测它们是否存在。在前面的代码中，我们在 App 类型上应用了 Serializable 和 DefaultMemberAttribute 两个特性。但是，当应用程序运行时，却只显示了 DefaultMemberAttribute 特性。IsDefined、GetCustomAttributes 以及 GetCustomAttribute 这样的方法不能处理伪定制特性。

理想情况下，.NET 框架组应该隐藏伪定制特性和通常的特性之间存在的一些处理方式的差别，.NET 框架在将来的版本中也许会这么做。但就目前来讲，我们只能采用其他一些方式来检测代码中是否应用了这些伪定制特性。

例如，System.Type 提供了一些只读属性如 IsSerializable、IsAutoLayout、IsExplicitLayout、IsLayoutSequential、等等。System.Reflection.FieldInfo 也提供有只读属性如 IsNotSerialized。但是，对于大多数伪定制特性来讲，FCL 中并没有定义允许我们检测它们是否存在的一些类型或者方法。(顺便提一句，有一种方式可以帮助我们检测伪定制特性是否存在，即用非托管代码来直接访问 CLR 的 COM 接口。)

下面列出了.NET 框架中定义的一些伪定制特性：

- System.NonSerializedAttribute
- System.SerializableAttribute
- System.Diagnostics.DebuggableAttribute
- System.Runtime.CompilerServices.MethodImplAttribute
- System.Runtime.InteropServices.DllImportAttribute
- System.Runtime.InteropServices.InAttribute
- System.Runtime.InteropServices.ComImportAttribute
- System.Runtime.InteropServices.FieldOffsetAttribute
- System.Runtime.InteropServices.GuidAttribute
- System.Runtime.InteropServices.InterfaceTypeAttribute
- System.Runtime.InteropServices.OptionalAttribute
- System.Runtime.InteropServices.OutAttribute
- System.Runtime.InteropServices.PreserveSigAttribute
- System.Runtime.InteropServices.StructLayoutAttribute
- System.Runtime.InteropServices.MarshalAsAttribute

委　　托

本章讨论回调函数。回调函数是一项非常有用的编程机制，它在程序设计界的应用已有很多年了。Microsoft .NET 框架使用一种称作委托(delegate)的技术来提供回调函数机制。与其他平台(例如非托管 C++)中使用的回调机制不同，委托为我们提供了更多的功能。例如，委托可以确保回调方法是类型安全的(这也是通用语言运行时最重要的目标之一)。委托还集成了按序调用多个方法的能力，并且同时支持调用静态方法和实例方法。

17.1 认识委托

C 语言运行时中的 qsort 就是利用回调函数来对一个数组中的元素进行排序的。在 Windows 中，窗口过程、钩子过程、异步过程调用等都要求使用回调函数。在.NET 框架中，回调方法也用于很多场合。例如，我们可以通过登记回调方法来获得各种各样的通知，如未处理的异常、窗口状态的改变、菜单项目的选中、文件系统的变更以及异步操作的完成等。

在非托管 C/C++中，函数的地址就是一个内存地址。该地址不会携带任何额外的信息，例如函数期望的参数个数、参数类型、函数的返回值类型以及函数的调用约定。一言以蔽之，非托管 C/C++中的回调函数是非类型安全的。

在.NET框架中，回调函数仍然像在非托管Windows编程中一样有用和普遍。但是，.NET框架为回调函数提供了一种称作委托(delegate)的类型安全的机制。在详细讨论委托之前，首先向大家展示一下它们的使用方法。下面的代码演示了怎样声明、创建和使用委托。

```
using System;
using System.Windows.Forms;
using System.IO;

class Set {
   private Object[] items;

   public Set(Int32 numItems) {
      items = new Object[numItems];
      for (Int32 i = 0; i < numItems; i++)
         items[i] = i;
   }

   // 定义一个 Feedback 委托类型
   // 注意：该类型嵌套在 Set 类中
   public delegate void Feedback(
      Object value, Int32 item, Int32 numItems);

   public void ProcessItems(Feedback feedback) {
      for (Int32 item = 0; item < items.Length; item++) {
         if (feedback != null) {
            // 如果指定有回调函数，则调用它们
            feedback(items[item], item + 1, items.Length);
         }
      }
   }
}

class App {
   static void Main() {
      StaticCallbacks();
      InstanceCallbacks();
   }

   static void StaticCallbacks() {
      // 创建一个含有 5 个 Object 的 Set 对象
      Set setOfItems = new Set(5);

      // 处理 items 元素，但是不给任何反馈
      setOfItems.ProcessItems(null);
      Console.WriteLine();
```

```
        // 处理 items 元素，并将反馈输出到控制台上
        setOfItems.ProcessItems(new Set.Feedback(App.FeedbackToConsole));
        Console.WriteLine();

        // 处理 items 元素，并将反馈输出到消息框上
        setOfItems.ProcessItems(new Set.Feedback(App.FeedbackToMsgBox));
        Console.WriteLine();

        // 处理 items 元素，并将反馈同时输出到控制台和消息框上
        Set.Feedback fb = null;
        fb += new Set.Feedback(App.FeedbackToConsole);
        fb += new Set.Feedback(App.FeedbackToMsgBox);
        setOfItems.ProcessItems(fb);
        Console.WriteLine();
    }

    static void FeedbackToConsole(
        Object value, Int32 item, Int32 numItems) {
        Console.WriteLine("Processing item {0} of {1}: {2}.",
            item, numItems, value);
    }
    static void FeedbackToMsgBox(
        Object value, Int32 item, Int32 numItems) {
        MessageBox.Show(String.Format("Processing item {0} of {1}: {2}.",
            item, numItems, value));
    }

    static void InstanceCallbacks() {
        // 创建一个含有 5 个 Object 的 Set 对象
        Set setOfItems = new Set(5);

        // 处理 items 元素，并将反馈输出到一个文件上
        App appobj = new App();
        setOfItems.ProcessItems(new Set.Feedback(appobj.FeedbackToFile));
        Console.WriteLine();
    }
    void FeedbackToFile(
        Object value, Int32 item, Int32 numItems) {

        StreamWriter sw = new StreamWriter("Status", true);
        sw.WriteLine("Processing item {0} of {1}: {2}.",
            item, numItems, value);
        sw.Close();
    }
}
```

下面先简要描述一下上述代码。注意代码最开始处的 Set 类。我们假定该类包含一组有待单独处理的元素。当我们创建一个 Set 对象时，我们将其应该管理的元素数目传递给它的构造器。Set 构造器会据此创建一个 Object 数组，并将每个对象初始化为一个整数值。

Set 类中还定义了一个公有委托类型 Feedback。委托表示一个回调方法的签名。在本例中，Feedback 委托表示一个接受 3 个参数(一个 Object 和两个 Int32)、且返回值为 void 的回调方法。从某种角度来看，委托和非托管 C/C++中表示函数地址的 typedef 非常类似。

最后，Set 类还定义了一个名为 ProcessItems 的公有方法。该方法接受一个参数 feedback(一个指向 Feedback 委托对象的引用)，然后遍历 Object 数组中所有的元素，并对每一个元素调用由 feedback 变量所指定的回调方法。传递给回调方法的参数分别为待处理的元素值、元素序号、以及数组中元素的总数。回调方法可以选择任何的方式来处理每个元素。

17.2 使用委托回调静态方法

既然我们已经理解了 Set 类型的设计和工作原理，下面我们就来看看如何使用委托来回调静态方法。本节重点讲解前面代码中的 StaticCallbacks 方法。

StaticCallbacks 方法首先构造了一个含有 5 个 Object 的 Set 对象。然后它又调用了 ProcessItems 方法，其中传递给 feedback 参数的值为 null。ProcessItems 方法会对 Set 对象维护的每个元素进行某种处理。因为本例中传给 feedback 参数的值为 null，所以 ProcessItems 方法在处理每个元素的时候不会调用任何回调方法。

当 StaticCallbacks 第 2 次调用 ProcessItems 方法时，它首先构造了一个新的 Set.Feedback 委托对象。该委托对象是一个方法的封装，这使得该方法可以通过委托封装进行间接的回调。本例中传递给 Feedback 构造器的是一个静态方法名 App.FeedbackToConsole，这使得 App.FeedbackToConsole 方法成为 Feedback 委托封装的对象。接着，new 操作符返回的引用被传递给 ProcessItems 方法。当 ProcessItems 方法执行时，它将为 Set 对象中的每一个元素调用 App 类型上的静态方法 FeedbackToConsole。FeedbackToConsole 方法简单地在控制台上输出一个字符串来表示正在处理的元素，以及元素的值。

> **注意** FeedbackToConsole 方法在 App 类型中定义为 private，但是 Set 类型的 ProcessItems 方法却能够调用它。这里不存在任何的安全问题，因为是 App 的代码自己显式决定返回一个私有方法的委托封装。

第 3 次调用 ProcessItems 方法和第 2 次非常类似。惟一的差别在于 Feedback 委托对象封装的是静态的 App.FeedbackToMsgBox 方法。FeedbackToMsgBox 方法首先创建一个字符串来表示正在处理的元素和它的值，然后将该字符串显示在一个消息框上。

第 4 次，也是最后一次调用 ProcessItems 方法为我们演示了怎样将委托对象链接在一起形成一个委托链。上面的例子首先创建了一个指向 Feedback 委托对象的引用变量 fb，并将其初始化为 null。该变量指向委托链表的头部，null 值表示链表上没有节点。接着它又创建了一个 Feedback 委托对象，其封装的对象为 App 的 FeedbackToConsole 方法。C#的 += 操作符用来将该委托对象追加到由 fb 引用的委托链表上。变量 fb 现在指向了一个有效链表的头部。

最后，又一个 Feedback 委托对象被创建，其封装的对象为 App 的 FeedbackToMsgBox 方法。同样，C#的 += 操作符用来将该委托对象追加到委托链表上，fb 也被更新指向新的链表头部。然后，当 ProcessItems 方法被调用时，新得到的 Feedback 委托链表的头部将被作为参数传递给它。在 ProcessItems 方法内部，调用回调方法实际上会调用委托链表上所有的委托对象封装的回调方法。换句话说，对于遍历的每一个元素，FeedbackToConsole 方法调用后紧跟着便是 FeedbackToMsgBox 方法调用。本章稍后将解释委托链的工作原理。

上面的例子中所有的操作都是类型安全的。例如，当构造一个 Feedback 委托对象时，编译器会确保 App 的 FeedbackToConsole 方法和 FeedbackToMsgBox 方法的原型与 Feedback 委托定义的原型完全相同。也就是说两个方法都必须接受 3 个参数(一个 Object 和两个 Int32)，并且两个方法要有同样的返回值类型(void)。如果 FeedbackToConsole 方法的原型被定义为下面的样子呢？

```
static void FeedbackToConsole(
    Object value, Int32 item, Int32 numItems, String s) {
    ...
}
```

C#编译器不会编译上面的代码，并且会报告以下错误："error CS0123: 方法'App.FeedbackToConsole(object, int, int,string)'与委托'void Set.Feedback(object, int, int)'不匹配"。

17.3 使用委托回调实例方法

前面向大家解释了怎样使用委托来回调静态方法。我们也可以使用委托来在一个指定的对象上回调实例方法。为了理解回调一个实例方法的工作原理，我们来看前面代码中的 InstanceCallbacks 方法：

```
static void InstanceCallbacks() {
    // 创建一个含有 5 个 Object 的 Set 对象
    Set setOfItems = new Set(5);

    // 处理 items 元素，并将反馈输出到一个文件上
    App appobj = new App();
    setOfItems.ProcessItems(new Set.Feedback(appobj.FeedbackToFile));
    Console.WriteLine();
}
```

注意这段代码在构造 Set 对象之后又构造了一个 App 对象。App 对象没有任何相关的字段或属性，这里创建它仅供演示之用。创建完 App 对象之后，InstanceCallbacks 方法又创建了一个新的 Feedback 委托对象，这一次传递给 Feedback 构造器的参数为 appobj.FeedbackToFile。也就是说，Feedback 的封装对象为 FeedbackToFile 方法，注意该方法是一个实例方法(而非静态方法)。当该实例方法被调用时，appobj 引用的对象就是实例方法操作的对象(传递隐含的 this 参数)。

FeedbackToFile 方法的行为和 FeedbackToConsole 方法、FeedbackToMsgBox 方法的行为类似，只不过它是打开一个文件，将表示正在处理的元素的字符串添加到文件末尾而已。

本例旨在向大家演示委托可以像封装静态方法一样封装实例方法。需要指出的是，对于封装实例方法的情况，委托需要知道方法将要操作的实例对象。

17.4 委托揭秘

从表面上看，委托好像很容易使用：我们用C#的关键字delegate来定义它们，然后用new操作符来构造它们的实例，最后再用我们所熟悉的“方法调用”的语法来调用委托表示的回调函数(不同之处在于我们使用的是指向委托对象的变量，而非方法名)。

但是，整个事情并非前面的例子所演示的那样简单。编译器和CLR在后台做了很多工作来隐藏问题本身的复杂性。本节将着重向大家解释编译器和CLR究竟是怎样来实现委托的。具备这些知识可以提高大家对委托的理解，从而帮助大家更好地使用它们。另外，本节还会谈到委托具有的一些额外的特性。

再来看一遍下面一行代码：

```
public delegate void Feedback(
   Object value, Int32 item, Int32 numItems);
```

当编译器遇到这段代码时，它会产生如下面所示的一个完整的类定义：

```
public class Feedback : System.MulticastDelegate {

   // 构造器
   public Feedback(Object target, Int32 methodPtr);

   // 下面的方法和源代码中指定的原型一样
   public void virtual Invoke(
      Object value, Int32 item, Int32 numItems);

   // 下面两个方法允许我们对委托进行异步回调
   public virtual IAsyncResult BeginInvoke(
      Object value, Int32 item, Int32 numItems,
      AsyncCallback callback, Object object);
   public virtual void EndInvoke(IAsyncResult result);
}
```

编译器定义的类中有4个方法：一个构造器、Invoke，BeginInvoke，以及EndInvoke。本章中我们只关注其中的构造器和Invoke。

我们可以用ILDasm.exe查看生成的模块来确定编译器的确自动产生了这样的类，如图17.1所示。

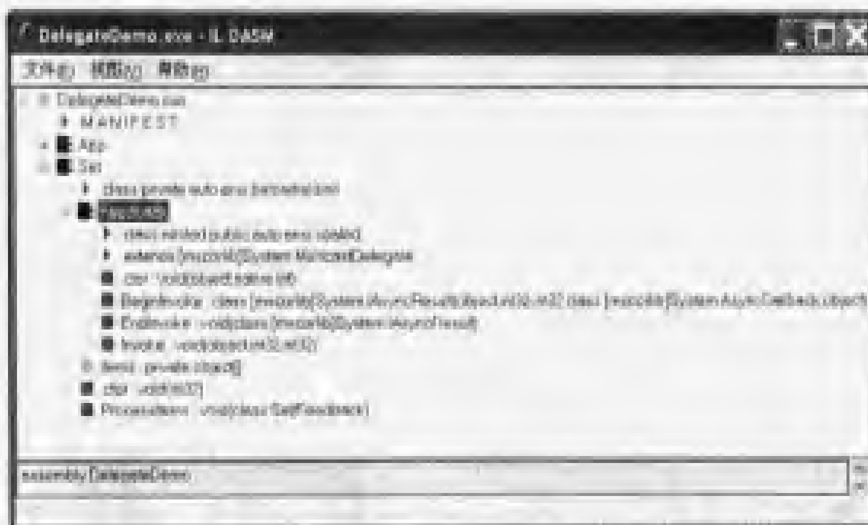


图 17.1 使用 ILDasm.exe 查看编译器为委托产生的元数据

本例中，编译器定义了一个名为 Feedback 的类，其继承自.NET 框架类库(FCL)中定义的 System.MulticastDelegate。(所有的委托类型都继承自 MulticastDelegate。)由于我们在源代码中将 Feedback 委托类型声明为 public，所以编译器产生的 Feedback 为一个公有类。如果我们在源代码中将其声明为 private 或者 protected，那么编译器产生的 Feedback 类也将为 private 或者 protected。委托类型可以定义在一个类中(在上面的例子中，Feedback 就被定义在 Set 类内)，也可以定义在一个全局的范围内。因为委托本身就是类，所以一个类可以在哪里定义，一个委托就可以在哪里定义。

因为所有的委托类型都继承自 MulticastDelegate，它们自然也就继承了 MulticastDelegate 的字段、属性和方法。在所有这些成员中，有 3 个私有字段可能是最重要的。表 17.1 描述了它们。

表 17.1 MulticastDelegate 中几个重要的私有字段

字 段	类 型	描 述
_target	System.Object	指向回调函数被调用时应该被操作的对象。该字段用于实例方法的回调
_methodPtr	System.Int32	一个内部的整数值(译注：准确地讲，该字段的类型为 System.IntPtr，其主要用于表示指针或句柄)，CLR 用它来标识回调方法
_prev	System.MulticastDelegate	指向另一个委托对象。该字段通常为 null

注意所有的委托都有一个构造器，并且该构造器接受两个参数：一个对象引用和一个指向回调方法的整数。但是，如果我们查看前面的源代码，我们会发现我们传递的是 App.FeedbackToConsole 或 appobj.FeedbackToFile 这样的值。大家直觉上可能会认为这样的代码不应该通过编译。

然而，实际上编译器知道我们正在构造的是一个委托，它会通过分析源代码来确定我们引用的是哪个对象和方法。其中的对象引用会被传递给 target 参数，一个特殊的标识方法的 Int32 值(由 MethodDef 或者 MethodRef 元数据标记获得)会被传递给 methodPtr 参数。对于静态方法而言，null 会被传递给 target 参数。在构造器内部，这两个参数会被保存在相应的私有字段中。

另外，构造器会将_prev 字段设置为 null。该字段用于创建一个 MulticastDelegate 对象的链表，本章 17.7 节对此有详细的解释。

每个委托对象实际上是对方法及其调用时操作的对象的一个封装。MulticastDelegate 类定义了两个只读公有实例属性：Target 和 Method。给定一个委托对象的引用，我们可以查询这些属性。Target 属性返回一个方法回调时操作的对象引用。如果是静态方法，Target 将返回 null。Method 属性返回一个标识回调方法的 System.Reflection.MethodInfo 对象。

我们可以用几种方式来使用上述两个属性。例如，我们可以查看一个委托对象是否引用有一个特定类型的实例方法：

```
Boolean DelegateRefersToInstanceMethodOfType(
   MulticastDelegate d, Type type) {

   return((d.Target != null) && d.Target.GetType() == type);
}
```

我们也可以编写代码来查看回调方法是否有着一个特定的名称(例如 FeedbackToMsgBox)：

```
Boolean DelegateRefersToMethodOfName(
   MulticastDelegate d, String methodName) {

   return(d.Method.Name == methodName);
}
```

我们已经知道了委托对象是怎样构造的，下面我们来看回调方法是怎样被调用的。为方便起见，下面再次列出 Set 类中 ProcessItems 方法的代码：

```
public void ProcessItems(Feedback feedback) {
    for (Int32 item = 0; item < items.Length; item++) {
        if (feedback != null) {
            // 如果指定有回调函数，则调用它们
            feedback(items[item], item + 1, items.Length);
        }
    }
}
```

注释下面就是调用回调方法的代码。看起来好像我们在调用一个名为 feedback 的函数，并且传递给了它 3 个参数。但是，实际上并没有什么 feedback 函数。因为编译器知道 feedback 是一个指向委托对象的变量，所以它会产生代码来调用该委托对象的 Invoke 方法。换句话说，当编译器遇到下面的代码时：

```
feedback(items[item], item + 1, items.Length);
```

它产生的代码就像编译自下面的源代码一样：

```
feedback.Invoke(items[item], item + 1, items.Length);
```

我们可以使用 ILDasm.exe 来查看 ProcessItems 方法编译后生成的 IL 代码来验证上面的说法。图 17.2 演示了 ProcessItems 方法编译后的 IL 代码。图中括起来的代码行表示调用 Set.Feedback 的 Invoke 方法。

如果我们在 feedback 委托对象上显式调用 Invoke 方法，C#编译器将产生以下错误："error CS1533: Invoke 无法直接在委托中调用"。C#不允许我们显式调用 Invoke 方法，但其他的编译器可能会要求我们通过显式调用 Invoke 来执行回调方法。实际上，Visual Basic 就要求我们这样做。(译注：Visual Basic 既可以用 Invoke 来执行回调方法，也可以像 C#那样使用"方法调用"的语法来执行回调方法。)

我们应该还记得编译器在定义 Feedback 类时定义了 Invoke 方法。当 Invoke 被调用时，它使用 _target 和 _methodPtr 两个私有字段来在指定的对象上调用期望的方法。注意 Invoke 方法的签名和 Feedback 委托的签名是相匹配的。换句话说，因为 Feedback 委托接受 3 个参数、并且返回 void，所以 Invoke 方法也接受同样的 3 个参数、并且返回 void。

图 17.2 使用 ILDasm.exe 验证编译器确实产生了调用 Set.Feedback 委托类型的 Invoke 方法的代码

17.5 委托史话：System.Delegate 与 System.MulticastDelegate

我们知道 System.MulticastDelegate 类定义于 FCL 中，但实际上 MulticastDelegate 类本身又继承自 System.Delegate(也定义于 FCL 中)，而 Delegate 又继承自 System.Object。在刚开始设计.NET 框架时，微软的工程师们感觉有必要提供两种不同类型的委托：一种是单播(single-cast)委托，一种是多播(multicast)委托。他们期望用继承自 MulticastDelegate 的类型来表示可以被链接在一起的委托对象，而用继承自 Delegate 的类型来表示不可以被链接在一起的委托对象。System.Delegate 被设计为一个基类型，它实现了回调一个方法所有必要的功能。MulticastDelegate 类继承自 Delegate，并且为创建 MulticastDelegate 对象链表提供了支持。

当编译器编译源代码时，它会检查委托的签名，然后在这两个类中选择一个最合适的作为编译器产生的委托类型的基类型。具体来讲就是使那些签名具有非 void 返回值的方法原型所表示的委托继承自 System.Delegate，而使那些签名具有 void 返回值的方法原型所表示的委托继承自 System.MulticastDelegate。做这样的选择具有一定的合理性，因为我们只能得到委托链表调用时最后一个方法的返回值。

然而，在.NET 框架的 beta 测试阶段，采用两种不同的基类型给开发人员造成了很大的困惑。而且这样设计委托也给它们的使用带来了一些限制。例如，对于许多具有返回值的方法，我们在很多情况下实际上是可以忽略这些返回值的。但是，因为这些方法具有非 void 的返回值，我们便不能从 MulticastDelegate 类来继承它们，从而导致我们不能将它们的实例放到一个委托链表中。

为了减少开发人员的困惑，微软的工程师们又希望将 Delegate 和 MulticastDelegate 合并为一个类，从而允许任何委托对象都可以被放入一个委托链表中。这样，所有的编译器产生的委托类也都继承自这一个类。这种改变显然会令.NET 框架组、CLR 组、编译器组，以及使用委托的开发人员轻松许多。

不幸的是，合并 Delegate 和 MulticastDelegate 的想法在整个.NET 框架开发周期中来得太晚了，微软担心这样的改变会带来很多潜在的 bug、以及不小的测试负担。所以在.NET 框架的版本 1 中，这些类并没有得到合并。我期望在.NET 框架未来的某个版本中这两个类能被合并为一个类。

虽然微软选择了延缓这两个类的合并，但是他们还是能够改变他们生产的编译器。实际上，微软所有的编译器目前产生的委托类型都继承自 MulticastDelegate 类。所以本章前面提出的所有委托类型都继承自 MulticastDelegate 类的说法并非是在撒谎。由于编译器的这种改动，不管回调方法是否具有返回值，委托类型的所有实例都将可以被放到一个委托链表中。

读到这里，大家可能会问：“我为什么需要知道这些？”实际上，随着使用委托机会的增多，大家肯定会遇到.NET 框架 SDK 文档中的 Delegate 类和 MulticastDelegate 类。这里的目的就是让大家理解二者之间的关系。另外，即使我们创建的所有委托类型都是以 MulticastDelegate 类作为基类型，我们偶尔还是会使用到 Delegate(而非 MulticastDelegate)提供的一些方法。例如，Delegate 类有两个静态方法 Combine 和 Remove(本章稍后会解释它们)，它们所接受的参数类型都为 Delegate。而我们自己定义的委托类型总是继承自 MulticastDelegate，后者又继承自 Delegate，所以我们的委托类型的实例将可以被传递给 Combine 和 Remove。

17.6 委托判等

在 FCL 提供的两个委托基类中，Delegate 重写了 Object 的 Equals 虚方法，MulticastDeleagate 又继承了 Delegate 的 Equals 实现。Delegate 重写的 Equals 方法在比较两个委托对象时会看它们的_target

和_methodPtr字段是否指向同样的对象和方法。如果这两个字段匹配，那么Equals将返回true，否则返回false。(译注：这段表述是错误的。经与Jeffrey先生沟通，他对这段文字进行了修正。下面是修正后的文本：Delegate重写了Object的Equals虚方法，MulticastDelegate又重写了Delegate的Equals实现。MulticastDelegate 重写的 Equals 方法在比较两个委托对象时会首先看它们的_target 和_methodPtr字段是否都指向同样的对象和方法。如果这两个字段不匹配，那么Equals将返回false。如果这两个字段都匹配，那么再看两个委托对象是否表示委托链表的头部——也就是说它们的_prev字段不为null。如果两个委托对象的_prev字段指示的链表有相同的长度，并且两个链表上对应委托对象的_target 和_methodPtr字段也都互相匹配，那么Equals将返回true，否则将返回false。)下面的代码演示了单个委托对象(非委托链)的判等情况：

```
// 构造两个指向相同回调目标/回调方法的委托对象
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToConsole);

// 即使 fb1 和 fb2 引用的是两个不同的对象，但它们
// 在内部却都引用着相同的回调目标/回调方法
Console.WriteLine(fb1.Equals.(fb2));        // 显示 "True"
```

另外，Delegate和MulticastDelegate还重载了 == 和 != 操作符。我们可以使用它们来代替调用Equals方法。看下面的代码：

```
// 构造两个指向相同回调目标/回调方法的委托对象
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToConsole);

// 即使 fb1 和 fb2 引用的是两个不同的对象，但它们
// 在内部却都引用着相同的回调目标/回调方法
Console.WriteLine(fb1==fb2);                // 显示 "True"
```

在我们操作委托链(这将在下一节中予以讨论)时，理解如何比较两个委托对象是否相等非常重要。

17.7 委 托 链

委托本身已经很有用了，再给它们加上链接支持，委托将变得更有价值。前面已经说过每个MulticastDelegate对象都有一个私有字段_prev。该字段为指向另一个MulticastDelegate对象的引用。也就是说，每个MulticastDelegate(或者任何继承自MulticastDelegate的类型)对象都有一个指向另一个MulticastDelegate(或者任何继承自MulticastDelegate的类型)对象的引用，这使得多个委托对象可以组合成一个链表。

Delegate 类中定义了 3 个静态方法来帮助我们操作委托链表：

```
class System.Delegate {
    // 组合 head 和 tail 所表示的链表，并返回 head。注意：head 将
    // 是最后一个被调用的委托对象
    public static Delegate Combine(Delegate tail, Delegate head);

    // 创建一个由委托数组表示的委托链表。注意：索引为 0 的元
    // 素将为链表头部，并且将是最后一个被调用的委托对象
    // （译注：索引为 0 的元素应为链表尾部，且是第一个被调用的委托对象）
    public static Delegate Combine(Delegate[] delegateArray);

    // 从链表中移除一个和 value 的回调目标/回调方法相匹配的委托，
    // 新的链表头会被返回，并且将是最后一个被调用的委托对象
    public static Delegate Remove(Delegate source, Delegate value);
}
```

当我们构造一个新的委托对象时，其_prev 字段会被设为 null，这表示委托链表上没有其他对象。要将两个委托对象组合成一个链表中，我们可以调用 Delegate 两个静态 Combine 方法中的任何一个：

```
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToMsgBox);
Feedback fbChain = (Feedback) Delegate.Combine(fb1, fb2);
// 图 17.3 的左边显示了上面的代码执行后委托链的样子

App appobj = new App();
Feedback fb3 = new Feedback(appobj.FeedbackToFile);
fbChain = (Feedback) Delegate.Combine(fbChain, fb3);
// 图 17.3 的右边显示了上面的代码执行后委托链的样子
```

图 17.3 展示了委托链的内部结构。

大家可能注意到了 Delegate 提供的另一个 Combine 方法接受的是一个 Delegate 引用数组。我们用该 Combine 方法可将上面的代码做如下改写：

```
Feedback[] fbArray = new Feedback[3];
fbArray[0] = new Feedback(FeedbackToConsole);
fbArray[1] = new Feedback(FeedbackToMsgBox);
App appobj = new App();
fbArray[2] = new Feedback(appobj.FeedbackToFile);

Feedback fbChain = (Feedback)Delegate.Combine(fbArray);
// 图 17.3 的右边显示了上面的代码执行后委托链的样子
```

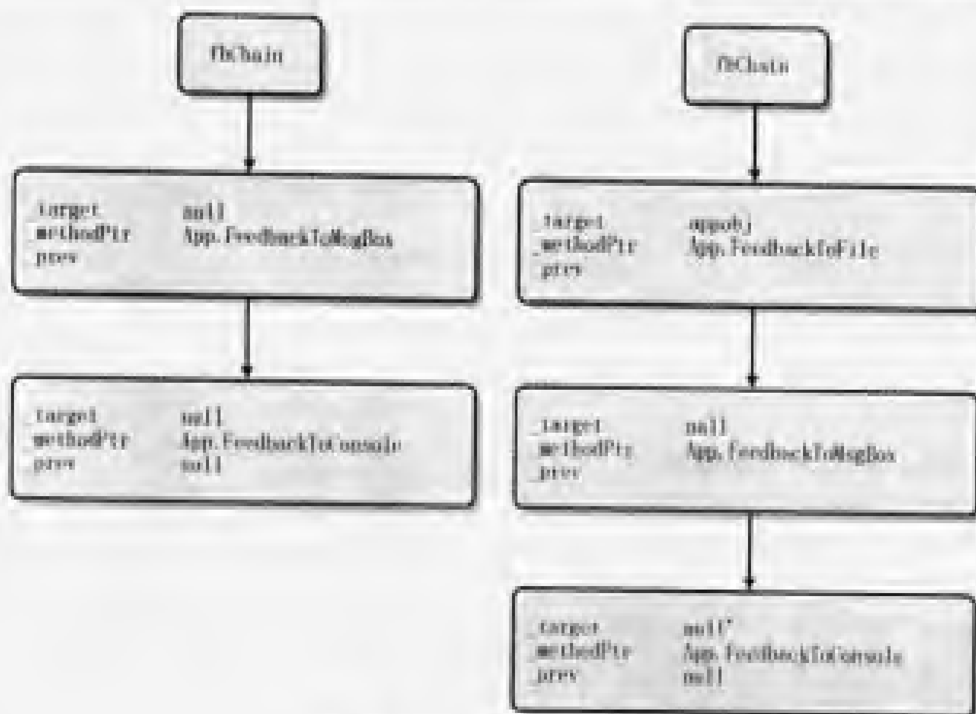


图 17.3 委托链的内部结构

当一个委托对象被调用时，编译器会产生对该委托类型的 Invoke 方法的调用(本章前面对此已经有过讨论)。为了帮助大家保持记忆，下面将 Feedback 委托声明再列一遍：

```
public delegate void Feedback(
   Object value, Int32 item, Int32 numItems);
```

这会导致编译器产生一个 Feedback 类，其中包含一个如下所示的 Invoke 方法(伪码描述)：

```
class Feedback : MulticastDelegate {
   public void virtual Invoke(
      Object value, Int32 item, Int32 numItems) {

      // 如果链表上包含有任何应该被首先调用的
      // 委托，那么将对它们进行递归调用
      if (_prev != null) _prev.Invoke(value, item, numItems);

      // 在指定的目标对象上调用回调方法
      _target.methodPtr(value, item, numItems);
   }
}
```

如我们所见，调用一个委托对象会导致它前面的委托首先被调用。当其前面的委托调用返回时，所得的返回值将被丢弃。在调用了其前面的委托之后，委托对象才会调用自己封装的回调目标/回调方法。下面的代码演示了这一点。

```
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToMsgBox);
Feedback fbChain = (Feedback) Delegate.Combine(fb1, fb2);

// 程序执行到此，fbChain 指向一个封装 FeedbackToMsgBox
// 的委托，该委托又指向另一个封装 FeedbackToConsole 的
// 委托。最后一个委托指向 null


// 现在我们来调用链表头部的委托，它在内部又会调用其前面
// 的委托，依此类推
if (fbChain != null) fbChain(null, 0, 10);

// 注意：在上面一行中，fbChain 显然不会为 null，但是
// 在调用委托之前首先检查它是否为 null 是一个好习惯
```

目前，我们展示的例子中的委托类型 Feedback 被定义为一个返回值为 void 的方法。但是，我们也可以将 Feedback 委托定义为如下的样子：

```
public delegate Int32 Feedback(
   Object value, Int32 item, Int32 numItems);
```

这样的话，它的 Invoke 方法在内部将为如下的样子(同样用伪码描述)：

```
class Feedback : MulticastDelegate {
   public Int32 virtual Invoke(
      Object value, Int32 item, Int32 numItems) {

      // 如果链表上包含有任何应该被首先调用的
      // 委托，那么将对它们进行递归调用
      if (_prev != null) _prev.Invoke(value, item, numItems);

      // 在指定的目标对象上调用回调方法
      return _target.methodPtr(value, item, numItems);
   }
}
```

当委托链表的头部被调用时，它首先会调用委托链中位于其前面的委托对象。但是，注意这时其前面的委托调用的返回值将被丢弃。应用程序代码只保留了从链表头部的那个委托调用(最后一次调用回调方法)中返回的值。

> **注意** 委托对象一旦被构造，它们就被认为是恒定不变的。也就是说，委托对象的_prev 字段总被设置为 null，并且不会改变。当我们调用 Combine 将一个委托对象组合到一个委托链中时，Combine 方法在内部会构造一个新的委托对象，新委托对象和源委托对象有着同样的_target 和_methodPtr 字段，但是其_prev 字段会被设置指向原先的委托链的头部。最后 Combine 方法返回新委托对象的地址。看下面的代码：
>
> ```
> Feedback fb = new Feedback(FeedbackToConsole);
> Feedback fbChain = (Feedback) Delegate.Combine(fb, fb);
> // 变量 fbChain 指向由两个委托对象组成的委托链
> // 其中一个委托对象就是 fb 原来引用的那个对象
> // 另一个是由 Combine 方法内部新构造的一个对象
> // 这个新构造的对象的_prev 字段指向 fb 引用的对象
> // Combine 方法最后返回新构造的委托对象
>
> // fb 和 fbChain 指向同一个委托对象吗？"False"
> Console.WriteLine(Object.ReferenceEquals(fb, fbChain));
>
> // fb 和 fbChain 指向同一个回调目标/回调方法吗？"True"
> Console.WriteLine(fb.Equals(fbChain));
> //(译注：上面一行代码的执行结果应为 False，
> // 这是因为 MulticastDelegate 类重写了 Object 的
> // 虚方法 Equals。重写后的 Equals 会比较_target、
> // _methodPtr,以及_prev 3个字段。)
> ```

我们已经知道了怎样创建委托链，下面我们来看一看怎样从一个委托链上移除委托对象。要从一个委托链上移除委托对象，我们需要调用 Delegate 的静态方法 Remove：

```
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToMsgBox);
Feedback fbChain = (Feedback) Delegate.Combine(fb1, fb2);
// fbChain 指向含有两个委托对象的委托链

// 调用委托链：有两个方法被调用
if (fbChain != null) fbChain(null, 0, 10);
fbChain = (Feedback)
    Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));
// fbChain 指向含有一个委托对象的委托链
// 调用委托链：有一个方法被调用
if (fbChain != null) fbChain(null, 0, 10);
```

```
fbChain = (Feedback)
    Delegate.Remove(fbChain, new Feedback(FeedbackToConsole));
// fbChain 指向含有 0 个委托对象的委托链(fbChain 为 null)

// 调用委托链：没有方法被调用
if (fbChain != null) fbChain(null, 0, 10);
// 现在大家该明白了我们为什么每次都要比较 fbChain 是否为 null 了！
```

在上面的代码中，我们首先创建了两个委托对象，然后调用 Combine 方法将它们组合为一个链表。之后我们又调用了 Remove 方法。Remove 方法的第 1 个参数指向委托链的头部，第 2 个参数指向要移除的委托对象。构造一个新的委托对象的目的就是为了将其从委托链中移除，这看起来很奇怪。要完全理解这样做的必要性还需要一些额外的解释。

在调用 Remove 时，我们构造了一个新的委托对象。该委托对象的_target 和_methodPtr 字段都得到了适当的初始化，_prev 字段被初始化为 null。Remove 方法首先扫描 fbChain 引用的委托链，检查其上是否有和新创建的委托对象相等的委托对象。注意 Delegate 类重写的 Equals 方法仅仅比较_target 和_methodPtr 字段，而忽略_prev 字段。(译注：Delegate 类重写的 Equals 方法确实是仅比较_target 和_methodPtr 两个字段，并忽略_prev 字段。但是 Remove 方法在判断两个委托对象是否相等时，使用的并不是 Delegate 重写的 Equals 方法，而是一个内部的判等方法。这个内部的判等方法基本思想和 MulticastDelegate 重写的 Equals 方法类似。换句话说它除了比较_target 和_methodPtr 字段外，还会比较_prev 字段——在_prev 不为 null 的情况下。)

如果找到一个匹配，Remove 方法便从委托链上移除找到的委托对象，这可以通过修正委托链上找到的那个委托对象前一个委托对象的_prev 字段来达到。Remove 方法最终返回新委托链的头部。如果找不到，Remove 方法将不执行任何操作(也不会抛出异常)，而直接返回传递给它的第一个参数。

注意，如果在委托链上找到了要移除的委托对象，那么每一次调用 Remove 方法都只移除一个这样的对象。(译注：Remove 方法实际上是从第 1 个参数表示的委托链中移除第 2 个参数表示的委托链。所以这里准确的说法应该是“如果在委托链上找到了要移除的委托链，那么每一次调用 Remove 方法都只移除一个这样的委托链”。当然如果要移除的委托链的长度为 1，那么就变成了委托对象了。另外补充一点，Remove 方法每一次都是移除从链表头开始第一个匹配的委托链。)

```
Feedback fb = new Feedback(FeedbackToConsole);
Feedback fbChain = (Feedback) Delegate.Combine(fb, fb);
// fbChain 指向含有两个委托对象的委托链

// 调用委托链：FeedbackToConsole 被调用了两次
if (fbChain != null) fbChain(...);
// 从委托链上移除一个回调方法
fbChain = (Feedback) Delegate.Remove(fbChain, fb);
// 调用委托链：FeedbackToConsole 被调用了一次
if (fbChain != null) fbChain(...);
```

```
// 从委托链上移除一个回调方法
fbChain = (Feedback) Delegate.Remove(fbChain, fb);

// 调用委托链：没有方法被调用
if (fbChain != null) fbChain(...);
```

17.8 C#对委托链的支持

为了减轻开发人员的编程工作，C#编译器自动为委托类型实例提供了 += 和 -= 操作符重载支持。这两个操作符分别会调用 Delegate.Combine 和 Delegate.Remove 方法。使用这些操作符可以简化委托链的构造。看下面的代码：

```
Feedback fb = new Feedback(FeedbackToConsole);
App appobj = new App();
fb += new Feedback(appobj.FeedbackToFile);

// 调用委托链：FeedbackToFile 和 FeedbackToConsole 方法被调用
if (fb != null) fb(...);

// 从委托链上移除一个回调方法
fb -= new Feedback(FeedbackToConsole);

// 调用委托链：FeedbackToFile 方法被调用
if (fb != null) fb(...);

// 从委托链上移除最后一个回调方法
fb -= new Feedback(appobj.FeedbackToFile);

// 调用委托链：没有方法被调用
if (fb != null) fb(...);
```

C#编译器在内部会把在委托实例上应用的 += 操作符翻译为对 Delegate.Combine 方法的调用，而把在委托实例上应用的 -= 操作符翻译为对 Delegate.Remove 方法的调用。大家可以使用 ILDams.exe 来验证这一点。

17.9 对委托链调用施以更多的控制

到目前为止，大家应该已经理解了怎样创建 个委托对象链表，以及怎样调用委托链上所有的委托对象。由于委托类型的 Invoke 方法具有调用一个委托对象之前的委托对象(如果存在的话)的能力，所以我们可以保证委托链上所有的对象都会得到调用。这显然是一个很简单的算法。虽然这个简单的算法对于许多情况来说已经足够好了，但是它仍有许多限制。

例如，使用这种简单的算法，除了最后 一个回调方法的返回值外，其他回调方法的返回值都会被丢弃，我们无法得到所有回调方法的返回值。不仅如此，如果被调用的委托中有一个抛出了异常，或者阻塞了很长时间呢？由于委托链上的对象是按序调用的，所以如果有一个委托对象调用出了问题，它会阻止调用委托链上所有其他的委托对象。很明显，这样的算法不够强健。

对于那些该算法不能满足的情况，MulticastDelegate 类提供了一个实例方法 GetInvocationList，我们可以使用它来显式调用委托链上的每一个委托对象，这时我们可以采用任何满足我们需要的算法。

```
public class MulticastDelegate {
    // 创建一个委托数组，每一个元素都是委托链上对象的一个克隆
    // 注意：第 0 个元素是链表的尾部,它通常会被首先调用
    public virtual Delegate[] GetInvocationList();
}
```

GetInvocationList 方法在委托链上操作，并返回一个委托对象数组。从内部来看，GetInvocationList 方法会遍历指定的委托链，然后为委托链上的每一个对象创建一个克隆体，并将其添加在数组中。其中每一个克隆体的_prev 都会被设为 null，所以每一个对象都是孤立的，不会引用任何其他的对象链表。

利用 GetInvocationList 方法，我们可以很容易编写一个算法来显式调用数组中的每个对象。看下面的代码：

```
using System;
using System.Text;
// 定义 一个 Light 组件
class Light {
    // 该方法返回 light 的状态
    public String SwitchPosition() {
        return "The light is off";
    }
}
```

```
// 定义一个 Fan 组件
class Fan {
    // 该方法返回 fan 的状态
    public String Speed() {
        throw new Exception("The fan broke due to overheating");
    }
}

// 定义 个 Speaker 组件
class Speaker {
    // 该方法返回 speaker 的状态
    public String Volume() {
        return "The volume is loud";
    }
}

class App {

    // 委托定义，该委托允许我们查询一个组件的状态
    delegate String GetStatus();

    static void Main() {
        // 声明一个空委托链
        GetStatus getStatus = null;

        // 构造 3 个组件，并将它们的状态方法添加到委托链上
        getStatus += new GetStatus(new Light().SwitchPosition);
        getStatus += new GetStatus(new Fan().Speed);
        getStatus += new GetStatus(new Speaker().Volume);

        // 显示一个统一的状态报告来反映 3 个组件的状态
        Console.WriteLine(GetComponentStatusReport(getStatus));
    }

    // 该方法查询多个组件的状态并返回一个状态报告
    static String GetComponentStatusReport(GetStatus status) {

        // 如果委托链为空，那么不做任何事情
        if (status == null) return null;

        // 下面的变量用来创建状态报告
        StringBuilder report = new StringBuilder();

        // 获得一个由委托链上的对象组成的数组
        Delegate[] arrayOfDelegates = status.GetInvocationList();
```

```
        // 遍历数组中的每个委托对象
        foreach (GetStatus getStatus in arrayOfDelegates) {

            try {
                    // 获得一个组件的状态字符串，并将其追加在 report 后
                    report.AppendFormat("{0}{1}{1}",
                        getStatus(), Environment.NewLine);
            }
            catch (Exception e) {
                // 在报告中为组件产生一个错误条目
                Object component = getStatus.Target;
                report.AppendFormat(
                     "Failed to get status from {1}{2}{0} Error: {3}{0}{0}",
                    Environment.NewLine,
                    ((component == null) ? "" : component.GetType() + "."),
                    getStatus.Method.Name, e.Message);
            }
        }

        // 将统一的报告返回给调用者
        return report.ToString();
    }
}
```

编译并运行上面的代码，我们将会看到以下输出：

```
The light is off

Failed to get status from Fan.Speed
    Error: The fan broke due to overheating


The volume is loud
```

17.10 委托与反射

到目前为止，如果我们想使用一个委托，我们还必须提前知道其对应回调方法的原型。例如，如果 feedback 是一个指向 Feedback 委托的引用，那么调用该委托的代码看起来就应该像下面这样：

```
feedback(items[item], item +1, items.Length);
```

如我们所见，开发人员在编码的时候必须知道回调方法需要多少个参数，以及每个参数的类型。幸运的是，开发人员大多数时候都知道这些信息，所以像上面这样编写代码一般没有什么问题。

但是在某些情况下(比较少见)，开发人员在编译时并不知道这些信息。本书第 11 章在讨论 EventHandlerSet 类型时展示了一个例子，其中就使用了一个散列表来维护一组不同的委托类型。在运行时，我们通过在散列表中查找并调用特定的委托对象来触发事件。但在编译时，我们不可能知道哪个委托将被调用，以及需要传递哪些参数给委托的回调方法。

幸运的是，System.Delegate 提供了几个方法允许我们在编译时不知道这些信息的情况下仍能创建并调用一个委托。下面是 Delegate 定义的这些方法：

```
public class Delegate {
    // 创建一个封装 MethodInfo 的 delType 委托
    public static Delegate CreateDelegate(Type delType,
        MethodInfo mi);

    // 创建一个封装静态方法的 delType 委托
    public static Delegate CreateDelegate(Type delType,
        Type type, String methodName);

    // 创建一个封装实例方法的 delType 委托
    public static Delegate CreateDelegate(Type delegateType,
        Object obj, String methodName);

    // 创建一个封装实例方法的 delType 委托
    public static Delegate CreateDelegate(Type delegateType,
        Object obj, String methodName, Boolean ignoreCase);

    public Object DynamicInvoke(Object[] args);
}
```

这里，所有的 CreateDelegate 方法都会创建一个新的继承自 Delegate 的对象(其类型由第一个参数指定)。CreateDelegate 方法其余的参数决定了继承自 Delegate 的对象需要封装的回调方法。我们可以为该参数指定一个 MethodInfo(本书第 20 章讨论)、一个类型的静态方法(一个 String)，或者一个对象的实例方法(一个 String)。

Delegate 的实例方法 DynamicInvoke 允许我们在运行时传递一组参数来调用一个委托对象的回调方法。当我们调用 DynamicInvoke 方法时，它在内部会确保我们传递的参数和回调方法期望的参数兼容。如果它们是兼容的，回调方法将被调用。如果它们不兼容，将有一个异常抛出。DynamicInvoke 最后返回委托对象所封装的回调方法的返回值。

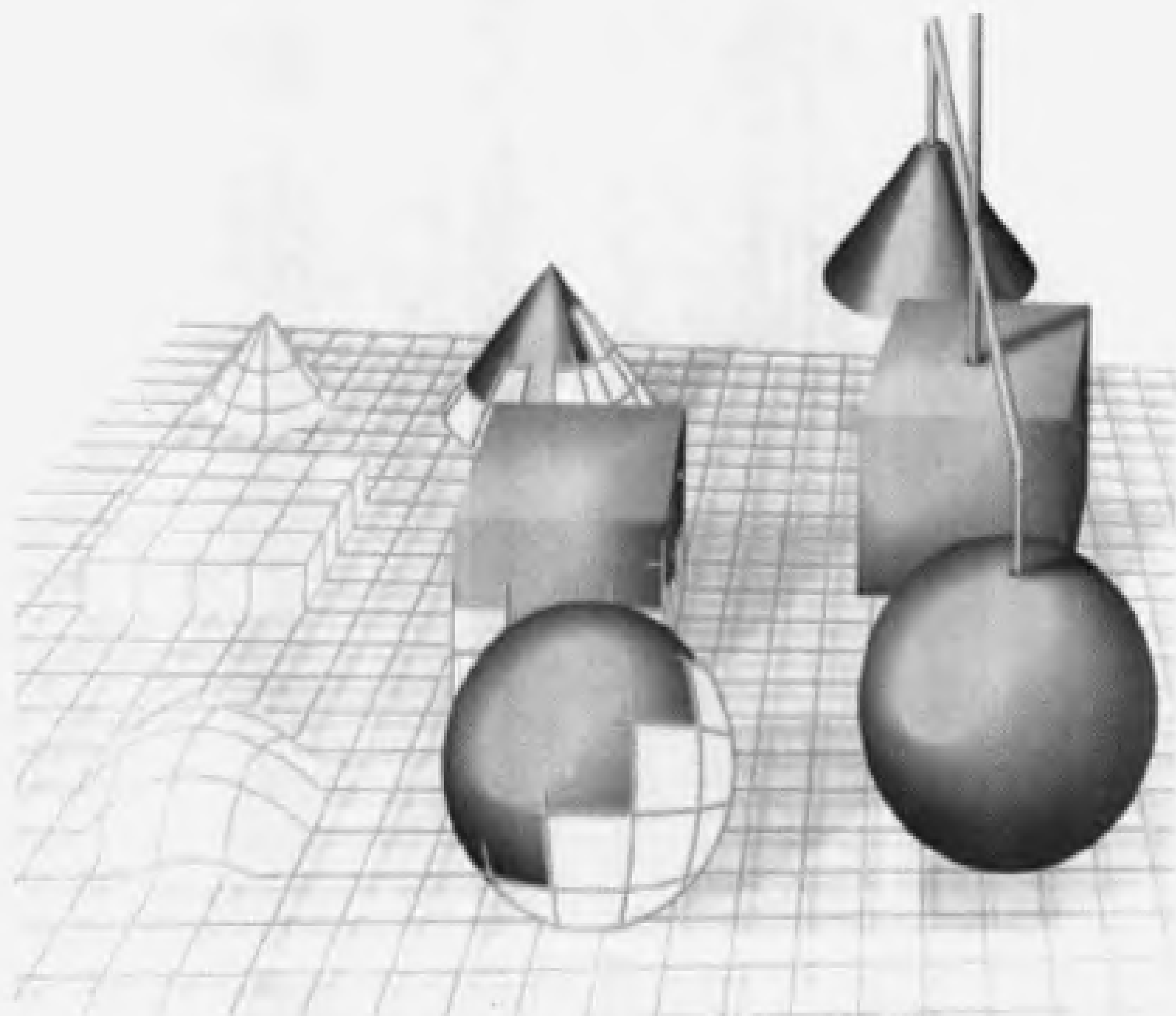
下面的代码演示了上述方法的用法：

```
using System;
using System.Reflection;
using System.IO;

// 下面是一些不同的委托定义
delegate Object TwoInt32s(Int32 n1, Int32 n2);
delegate Object OneString(String s1);

class App {
    static void Main(String[] args) {
        if (args.Length < 2) {
            String fileName =
                Path.GetFileNameWithoutExtension(
                    Assembly.GetEntryAssembly().CodeBase);
            Console.WriteLine("Usage:");
            Console.WriteLine("{0} delType methodName [Param1] [Param2]",
                fileName);
            Console.WriteLine(" where delType must be TwoInt32s or OneString");
            Console.WriteLine(" if delType is TwoInt32s, " +
                "methodName must be Add or Subtract");
            Console.WriteLine(" if delType is OneString, " +
                "methodName must be NumChars or Reverse");
            Console.WriteLine();
            Console.WriteLine("Examples:");
            Console.WriteLine(" {0} TwoInt32s Add 123 321", fileName);
            Console.WriteLine(" {0} TwoInt32s Subtract 123 321", fileName);
            Console.WriteLine(" {0} OneString NumChars \"Hello there\"",
                fileName);
            Console.WriteLine(" {0} OneString Reverse \"Hello there\"",
                fileName);
            return;
        }
        Type delType = Type.GetType(args[0]);
        if (delType == null) {
            Console.WriteLine("Invalid delType argument: " + args[0]);
            return;
        }
        Delegate d;
        try {
            d = Delegate.CreateDelegate(delType, typeof(App), args[1]);
        }
        catch (ArgumentException) {
            Console.WriteLine("Invalid methodName argument: " + args[1]);
```

```
            return;
        }

        Object[] callbackArgs = new Object[args.Length - 2];
        if (d.GetType() == typeof(TwoInt32s)) {
            try {
                for (Int32 a = 2; a < args.Length; a++)
                    callbackArgs[a - 2] = Int32.Parse(args[a]);
            }
            catch (FormatException) {
                Console.WriteLine("Parameters must be integers.");
                return;
            }
        }
        if (d.GetType() == typeof(OneString)) {
            Array.Copy(args, 2, callbackArgs, 0, callbackArgs.Length);
        }

        try {
            Object result = d.DynamicInvoke(callbackArgs);
            Console.WriteLine("Result = " + result);
        }
        catch (TargetParameterCountException) {
            Console.WriteLine("Incorrect number of parameters specified.");
        }
    }

    // 下面的回调方法接受两个 Int32 参数
    static Object Add(Int32 n1, Int32 n2) {
        return n1 + n2;
    }
    static Object Subtract(Int32 n1, Int32 n2) {
        return n1 - n2;
    }

    // 下面的回调方法接受一个 String 参数
    static Object NumChars(String s1) {
        return s1.Length;
    }
    static Object Reverse(String s1) {
        Char[] chars = s1.ToCharArray();
        Array.Reverse(chars);
        return new String(chars);
    }
}
```

第V部分

类型管理



异 常

本章讨论**异常处理**(exception handling)。异常处理是.NET框架中一种非常强大的机制，它使得我们编写的代码更加健壮、也更具可维护性。下面列出了异常处理为我们带来的一些具体的好处：

- **利用异常处理，我们可以将资源清理代码放在一个固定的位置，并且确保它们得到执行** 将资源清理(cleanup)代码从应用程序的主逻辑移到一个固定的位置后，应用程序将会变得更加容易编写、理解和维护。确保资源清理代码的运行意味着我们可以将应用程序保持在一个一致的状态。例如，当执行文件写入的代码由于某种原因不能正常工作时，我们可以利用异常处理来确保文件得到关闭。

- **利用异常处理，我们可以将处理异常的代码放在一个集中的位置** 有许多原因可以导致代码的失败：算术溢出、堆栈溢出、内存耗尽、参数范围越界、数组索引越界、试图访问已经关闭的资源(如一个文件)、等等。如果不使用异常处理，要使我们编写的代码可以很容易地检测这些故障、并从中恢复过来，即使可能的话，也是非常困难的。如果将检测这些潜在故障的代码分散到应用程序的主逻辑中将会使代码非常难以编写、理解和维护。另外编写代码来检测这些潜在的故障也会为应用程序带来不小的性能损失。

如果使用异常处理，我们就不需要自己编写代码来检测这些潜在的故障。相反，我们只管编写我们的代码，并假设它们不会出现问题。这自然使代码更容易编写、理解和维护。而且，代码的效率也会很高。最后，我们只需将异常恢复代码放在一个集中的位置即可。只有当应用程序产生故障时，异常处理机制才会执行我们的恢复代码。

- **利用异常处理，我们可以很容易定位和修复代码中的 bug** 当我们的代码出现问题时，通用语言运行时(CLR)会遍历线程的调用堆栈，查找能够处理该异常的代码。如果找不到这样的代码，我们将收到一个“未处理异常”的通知。根据得到的“未处理异常”通知，我们可以很容易定位产生故障的源代码，判定故障原因，然后修改源代码去除 bug。这意味着我们可以在应用程序的开发和测试阶段检测出 bug，并在部署之前将其修复。这样部署的应用程序自然会变得更加健壮，同时也可以提高终端用户的体验。

如果异常处理使用正确的话，它将是一个非常强大的工具，可以极大地减轻软件开发人员的负担。但是，如果使用的不正确，异常处理也可能会隐藏代码中一些严重的问题，或者为我们传递一些关于实际问题的错误信息。本章大部分内容将致力于向大家解释如何正确使用.NET 框架中的异常处理。

18.1 异常处理的演化

当设计 Win32 API 和 COM 时，微软没有使用异常处理来向调用者通知代码中的问题。相反，大多数 Win32 API 都是通过返回 FALSE 来表示函数调用产生了问题，而调用者则可以通过调用 GetLastError 函数来获得这些问题的相关信息。另一方面，COM 选择了返回一个 HRESULT 来描述问题。如果 HRESULT 的高位为 1，则表示一个假设被违反，HRESULT 的其余位表示的值则可以帮助我们判断问题的原因。

微软没有为 Win32 API 和 COM 选择异常处理有许多原因：

- 大多数开发人员还不熟悉异常处理。
- 许多编程语言，包括 C 和早期版本的 C++，都不支持异常处理。
- 一些开发人员感到异常难以使用和理解，微软也不愿意强迫大家接受它们。(就个人而言，我认为异常处理所带来的好处远远超出了异常处理的学习曲线。)

- 异常处理可能会损伤应用程序的性能。在某些情况下，异常处理的效率确实不如简单地返回错误码。但是，如果异常很少被抛出，那么这种额外的负担也是微不足道的。同样，我认为异常处理的好处超过了它可能带来的性能损失。而且我还相信，只要异常处理能够在整个应用程序中正确地使用，那么系统的性能会因此而提高。本章后面会对异常处理的性能问题予以探讨。另外，异常处理在托管代码中的代价要比在某些系统(如非托管 C++)中小得多。

如我们所见，旧有的报告问题的方式所做的工作非常有限，因为调用者所能得到的只是一个 32 位的值。如果该值是一个表示无效参数的代码，那么调用者将不能知道哪个参数是无效。如果该值是一个表示除零错误的代码，那么调用者将不能知道是哪行代码导致了这种错误，因此也就不能方便地修复代码。在这种条件下，如果还要保持不丢失有关问题的重要信息将是十分困难的。如果应用程序代码检测到了一个问题，我们当然希望能够得到有关问题的所有信息，因为只有这样我们才能尽可能高效、方便地采取正确的补救措施。

异常对象相较于 32 位的错误码值有着诸多的优势。每个异常对象都包含着一个描述字符串，利用该字符串，我们将能够知道到底是哪个参数导致了问题。该字符串还可能包含一些额外的信息来帮助我们改善代码。另外，每个异常对象还包含着一个堆栈踪迹(stack trace)，根据堆栈踪迹，我们将可以知道是哪段代码路径导致了异常。

异常处理的另一个好处是我们不必在异常出现的地方捕获或者检测它们。这会极大地简化编码工作，因为我们不必再为每一个可能失败的语句或者方法调用都添加错误检测和矫正代码。

和前一个好处密切相关的可能是异常处理所带来的最大好处——我们不能轻易忽略一个异常。在 Win32 环境下，如果我们调用了一个 Win32 函数，而该函数又返回了一个错误代码，我们很容易忽略该代码，从而使得应用程序按正常的情况继续运行。但是在.NET 框架中，如果一个方法抛出了一个异常，该方法将不能再按正常的方式继续运行。如果应用程序没有捕获该异常，CLR 将中断应用程序。对于某些人来讲，这种行为可能过于激烈和苛刻。然而，我认为这种做法是值得肯定的。

如果我们调用了一个方法，而该方法又抛出了一个异常，那么让应用程序再继续运行是不恰当的。因为应用程序的剩余部分会假设前面的操作都已按照期望的方式执行完毕了。但如果事实不是这样，应用程序再继续运行将可能产生无法预期的后果。例如，如果应用程序中的用户数据被损坏，那么该数据将不能再继续被操作。如果使用 Win32 和 COM 的 32 位错误代码，应用程序产生不可预期的结果的可能性将非常大。但如果换成异常处理，这种情况就可以被避免。

Microsoft .NET 框架中的类型定义的所有方法都是通过抛出异常来表明调用代码违反了某个假设，它们不会再返回 32 位的状态码值。因此，所有的.NET 开发人员都必须对异常以及怎样在代码中处理它们有一个坚实的理解。微软在这方面做了一个伟大的决定！大家很快将会看到，使用异常处理非常简单，它也使得我们的代码更加容易实现、阅读和维护。另外，异常处理还允许我们编写可以从任何情形都能恢复的更加健壮的代码。如果使用的正确，异常处理还可以防止应用程序崩溃，从而提高软件用户的满意度。

18.2 异常处理机制

本节向大家介绍.NET 框架中异常处理的机制和 C#中一些相关的构造，但是这里不打算在细节处做过多的着墨。本章的目的是为大家提供一些关于在代码中何时以及怎样使用异常处理的指导原则。如果大家希望得到更多的细节，可参见.NET 框架 SDK 文档以及相关的编程语言参考。顺便提一下，.NET 框架中的异常处理机制是由 Windows 提供的结构化异常处理(structured exception handling，简称 SEH)机制所构造的。SHE 在很多地方都有讨论，包括我自己的书 Programming Application for Microsoft Windows(第 4 版，微软出版社，1999)，其中关于 SEH 有三章的讨论内容。

下面的 C#代码为我们展示了异常处理机制的标准用法，并从总体上为我们描述了异常处理块的样子，以及它们的用途。代码后面的各个小节详细地解释了 try、catch、finally 三种语句块，并提供了有关使用它们的一些注意事项。

```
void SomeMethod() {

    try {
        // 我们在 try 块中放入那些需要恢复或者
        // 清理操作的代码
    }
    catch (InvalidCastException) {
        // 我们在这个 catch 块中放入那些能够从
        // InvalidCastException 异常(或者任何继承自
        // InvalidCastException 的异常)中恢复的代码
    }
```

```
    catch (NullReferenceException) {
        // 我们在这个 catch 块中放入那些能够从
        // NullReferenceException 异常(或者任何继承自
        // NullReferenceException 的异常)中恢复的代码
    }
    catch (Exception e) {
        // 我们在这个 catch 块中放入那些能够从
        // 任何与 CLS 兼容的异常中恢复的代码

        // 当捕获到一个和 CLS 兼容的异常，我们通常应
        // 该将其重新抛出。本章稍后解释异常的重新抛出
        throw;
    }
    catch {
        // 我们在这个 catch 块中放入那些能够从任何与
        // CLS 兼容、或者不兼容的异常中恢复的代码

        // 当捕获到任何一个异常，我们通常应该将其重
        // 新抛出。本章稍后解释异常的重新抛出
        throw;
    }
    finally {
        // 在 finally 块中我们放入那些对 try 块中所启
        // 动的操作进行清理的代码。不管是否有异常抛出，
        // finally 块中的代码总是会被执行
    }

    // 如果 try 块没有抛出异常、或者一个 catch 块捕
    // 获了异常并且没有抛出别的异常或重新抛出捕获
    // 到的异常，finally 块后的代码会被执行
}
```

上面的代码演示了使用各种异常处理块的一种可能的方式。不要被这些代码吓着——大多数方法都只有一个 try 块和一个匹配的 finally 块、或者一个 try 块和一个匹配的 catch 块。通常一个方法内不会有这里出现的这么多的 catch 块，这里列出它们仅仅处于演示目的。

18.2.1 try 块

try 块中包含的通常是一些需要资源清理或异常恢复的操作。所有的资源清理代码都应该放在一个 finally 块中。try 块还可以包含可能会抛出异常的代码。异常恢复代码应该放在一个或多个 catch

块中。我们应该为应用程序可以从中恢复的每一种异常事件创建一个 catch 块。一个 try 块必须有至少一个与之相关联的 catch 块或者 finally 块，单独一个 try 块是没有任何意义的。

18.2.2 catch 块

catch 块中包含的是出现异常时需要执行的响应代码。一个 try 块可以有 0 个或多个 catch 块与之相关联。如果 try 块中的代码没有抛出异常，CLR 将永远不会执行与该 try 块相关联的所有 catch 块中的代码。这时，线程就会跳过所有的 catch 块，直接执行 finally 块中的代码(如果存在的话)。在 finally 块中的代码执行完毕后，线程将会继续执行紧接着 finally 块后的语句。

出现在 catch 关键字后的表达式被称作**异常筛选器**(exception filter)。异常筛选器本身是一个类型，它表示开发人员预料到的、并且可以从中恢复的一种异常情况。在 C#中，一个捕获筛选器(译注：即异常筛选器)中的类型必须是 System.Exception 或者一个继承自 System.Exception 的类型。例如，前面代码中包含的几个 catch 块中打算处理的异常就包括 InvalidCastException(或者任何继承自它的异常)，NullReferenceException(或者任何继承自它的异常)，或者任何的 Exception(任何与 CLS 兼容的异常类型)。

注意代码执行时是自上而下搜索 catch 块的，我们应该将更具体的异常(即其基类在继承体系中离 System.Object 较远的那些类型)放在上面。实际上，如果更具体的 catch 块出现在离代码底部更近的位置，C#编译器将会产生一个错误，因为这样的 catch 块不可能被执行到。

如果 try 块(或者任何被 try 块中的代码调用的方法)中的代码在执行时抛出了一个异常，CLR 将搜索那些筛选器能够识别该异常的 catch 块。如果与该 try 块相关联的捕获筛选器中没有一个能够接受该异常，CLR 将沿着调用堆栈向更高一层搜索能够接受该异常的捕获筛选器。如果在搜索到了调用堆栈的顶部还没有找到能够处理该异常的 catch 块，那么就会出现一个未处理异常。本章后面将对未处理异常做更多的探讨。

一旦 CLR 找到了一个能够处理所抛出异常的捕获筛选器，它将执行从抛出异常的 try 块开始，到匹配异常的 catch 块为止的范围内所有的 finally 块。(译注：注意这里 finally 块的范围定义，这是一个非常精确的定义，读者要从方法调用的堆栈结构来理解它，而不要从简单的代码路径上来理解它——因为在异常抛出以后，很多代码路径是执行不到的。)注意，和匹配异常的 catch 块相关联的 finally 块这时还没有执行。该 finally 块中的代码一直要等到 catch 块中的处理代码执行完毕之后才会执行。

重要 C#只能抛出与 CLS 兼容的异常——即从 System.Exception 继承的异常类型。但是，CLR 允许抛出任何类型的对象。C#为我们提供了一种特殊的 catch 块来捕获与 CLS 不兼容的异常：

```
catch {          // 注意这里没有指定异常筛选器
    // 这里执行恢复代码
    ...
    throw;       // 重新抛出异常以使其他代码了解发生的事情
}
```

因为我们不能捕获这样的 catch 块中的对象，所以我们不能得到所捕获异常的任何信息。我们惟一能够做的就是执行一些恢复代码，然后重新抛出异常。顺便提一句，这样的 catch 块也可以捕获任何与 CLS 兼容的异常。

根据文档记录，.NET 框架类库(FCL)不会抛出任何与 CLS 不兼容的异常，事实上，我自己也从来没有看到任何托管代码抛出过与 CLS 不兼容的异常。当然，如果我们在中间语言(IL)下编程，我们将能够抛出与 CLS 不兼容的异常(例如 Int32)。托管扩展 C++也允许我们抛出与 CLS 不兼容的异常。显然，因为许多托管编程语言都没有为与 CLS 不兼容的异常提供足够高的支持，所以我们编写的任何代码都应该只抛出那些与 CLS 兼容的异常。

在 C#中，捕获筛选器可以指定一个异常变量。当捕获到一个异常时，该变量将指向那个被抛出的、类型继承自 System.Exception 的对象。catch 块中的代码可以通过引用该变量来获取有关异常的特定信息(例如异常所历经的堆栈踪迹)。尽管可以改变该对象，但是我们不应该这么做，而应该把它当成只读对象。本章稍后将解释 Exception 类型，以及我们可以在其上进行的操作。

在所有 finally 块中的代码都执行完毕后(译注：这里指前面定义的“从抛出异常的 try 块开始，到匹配异常的 catch 块为止的范围内所有的 finally 块”)，catch 块中的处理代码才开始执行。catch 块中的代码一般执行一些从异常中恢复的操作。在 catch 块的末尾，我们有三种选择：

- 重新抛出所捕获的异常，向更高一层的调用堆栈中的代码通知该异常的发生。

- 抛出一个不同的异常，向更高一层的调用堆栈中的代码提供更多的异常信息。
- 让线程从 catch 块的底部退出。

本章稍后会就何时使用这些技巧提供一些指导原则。

如果我们选择使用前两种技巧，我们将抛出一个异常，CLR 的行为将和以前处理异常时的一样：它将遍历调用堆栈搜索能够恢复该异常的捕获筛选器。如果我们选择使用最后一种技巧，当线程从 catch 块的底部退出后，它将立即开始执行包含在与之匹配的 finally 块中的代码——当然前提是存在这样的 finally 块。在 finally 块中的所有代码都执行完毕以后，线程将退出 finally 块，开始执行紧跟着 finally 块后的语句。如果不存在这样的 finally 块，线程将执行紧跟着最后一个 catch 块后的语句。

18.2.3 finally 块

finally 块中包含的代码是确保要执行的代码。一般地，finally 块中的代码执行的是一些资源清理操作，这些清理操作通常是对应的 try 块中的行为所需要的。例如，如果我们在一个 try 块中打开一个文件，那么我们就应该将关闭文件的代码放在与其对应的 finally 块中：

```
void ReadData(String pathname) {

   FileStream fs = null;
   try {
      fs = new FileStream(pathname, FileMode.Open);
      // 处理文件中的数据
      ...
   }
   catch (OverflowException) {
      // 在这里，我们放入那些从 OverflowException
      // (或者任何继承自 OverflowException 的异常)
      // 中恢复的代码
      ...
   }
   finally {
      // 确保文件被关闭
      if (fs != null) fs.Close();
   }
}
```

在上面的代码中，如果 try 块中的代码执行后没有抛出异常，那么 finally 块中的代码将会执行，从而将文件关闭。如果 try 块中的代码抛出了一个异常，不管该异常是否被捕获到，finally 块中的代码也会执行，从而也能确保文件得到关闭。将关闭文件的语句放在 finally 块之后是不正确的，因为这样的语句在有异常抛出而没有被捕获到的情况下将不会执行，从而使得文件得不到关闭。

一个 try 块并非必须要有一个 finally 块相关联，有时候 try 块中的代码并不需要任何清理工作。但是，如果有 finally 块，那么它必须出现在所有的 catch 块之后，并且一个 try 块最多只能有一个相关联的 finally 块。

当线程到达 finally 块中的代码底部时，它将从其中退出来，然后执行紧接着 finally 块后的语句(译注：这里的前提是没有出现未处理异常，换句话说就是 try 块中的代码没有抛出异常，或者抛出的异常得到了捕获)。记住，finally 块中的代码是清理代码，它们只应该撤销 try 块中发起的操作。我们应该避免将那些可能抛出异常的代码放在 finally 块中。但是，如果 finally 块中抛出了异常，事情也并非严重的不可救药——应用程序也不会因此而终止，系统的异常机制会继续工作，就像异常在 finally 块之后的代码中抛出的一样。

18.3 异常的本质

多年来，我遇到过很多开发人员都认为异常是指某些很少发生的事情，即一个异常事件。这时，我总是会请他们来定义“异常事件”，他们便会这样回答“一些你不能预料到的事情”。然后又会补充道“如果你从一个文件中读取字节，你总会到达文件的末尾。因为这种情况是可以预料到的，所以在你到达文件末尾时就不应该引发异常。相反在到达文件末尾时，Read 方法应该返回一个特殊的值。”

下面是我对这个问题的回答：“如果我有一个应用程序需要从一个文件中读取一个 20 字节的数据结构。但是由于某种原因，该文件只包含了 10 个字节。在这种情况下，我就没有预计到在读取文件的时候会到达文件末尾。但是因为我事实上会提早到达文件的末尾，我自然期望能有一个异常被抛出。”实际上，大多数文件都包含的是结构化数据。很少有应用程序先从文件中读取一大堆字节，然后再一个一个地处理它们，直至到达文件末尾。因此，我认为下面的做法更有意义：在 Read 方法试图读取超出文件末尾的部分时，让其总是抛出一个异常。

重要　很多开发人员都为异常处理(exception handling)这个术语所误导。他们总是认为异常(exception)这个词和事情发生的频率有关。例如，设计文件 Read 方法的开发人员总会这样说，“在读取一个文件时，你最终总是会到达数据的末尾的。既然这种行为总是会发生，那么我就应该将 Read 方法设计为让它通过返回一个特殊的值来报告这种行为，而不是让它抛出异常。”这句话的问题在于这是设计 Read 方法的开发人员的想法，而不是调用 Read 方法的开发人员的想法。

在设计 Read 方法时，开发人员不可能知道该方法在被调用时的所有可能情况。因此，开发人员不可能知道调用 Read 方法的程序试图读取超过文件末尾部分的频率有多高。实际上，由于大多数文件包含的都是结构化数据，试图读取超过文件末尾部分的行为很少发生。

另一种常见的误解是“异常”就是“错误”。错误意味着程序员做了某些不正确的事情。但是当调用代码在应用程序中错误地调用了 Read 方法时，设计该方法的开发人员是不可能知道的。只有调用该方法的开发人员才能对此做出判断，因此，只有调用程序才能判断调用结果是否出现了“错误”。所以我们应该避免有下面的想法“我将在我的代码中抛出一个异常来报告错误”。实际上，因为异常并不必然代表错误，所以本章将避免使用错误处理(error handling)这样的术语(当然本句除外)。

上面解释了关于异常的一些错误的认识。下面来解释其真正的含义。异常是对程序接口隐含假设的一种违反。例如，在设计一个类型时，我们一般首先会设想类型被使用的各种情况。然后再为类型定义字段、属性、方法、事件等。这些成员的定义方式(属性的数据类型，方法的参数、返回值等)就是类型的程序接口。

我们定义的接口通常会有一些隐含的假设。当这些程序接口中的假设被违反时，异常就出现了。

看下面的类定义：

```
public class Account {
    public static void Transfer(Account from, Account to, Decimal amount) {
        ...
    }
}
```

Transfer方法接受两个Account对象和一个Decimal值，该值表示在两个账号之间转账的数额。当调用Transfer方法时，我们会有一些很明显的隐含假设：from参数指向一个有效的Account，并且该账户的余额大于指定的转账金额。而且，从该方法的原型来看，我们并不清楚amount是否必须是一个正数、或者amount是否可以为一个负数。另外，如果from参数和to参数指向了同一个账号会发生什么情况？在同一个账户中进行转帐是否合法？如果amount参数超出了类设计者设定的范围怎么办？转帐金额为0合法吗？

上面这些问题的答案要到Account类的开发人员所做的Transfer实现中去找。理想情况下，设计该类的开发人员会清楚地将所有这些假设记录在文档中，这样使用该类的开发人员就可能以最高效的方式来实现调用代码，从而尽可能地减少运行时出现的怪异行为。遗憾的是，开发人员通常会发现文档缺乏所有隐含假设的描述，从而导致开发人员只能在运行时发现存在的违例情况。我们所希望的是所有这些违例都能在应用程序测试阶段被检测到，这样当应用程序部署到终端用户手中并运行时，它们就不会再出现了。

那么，一个方法怎样通知它的调用程序它所做的假设被违反了呢？答案是抛出一个异常。毕竟，异常就是对程序接口假设的一种违反。

另外希望大家能够理解的是，违反程序接口的假设并不必然是一件坏事情。实际上，这还可能是一件好事情，因为异常处理允许我们捕获异常、并顺利地继续往下执行。

在设计一个类型时，我们应该首先假设类型最常见的使用方式，然后设计其接口使之能够很好地处理这种情况。最后再考虑接口带来的隐含假设，并且当任何这些假设被违反时便抛出异常。

开发人员几乎从来不会去考虑的隐含假设

在访问任何方法时，开发人员通常都会认定几种假设——我们几乎很难考虑到的一些假设。当调用一个方法时，我们会假设有足够的堆栈空间；我们会假设有足够的内存来存放方法的 IL 指令被以 JIT 方式编译成的二进制代码；我们还会假设 CLR 在调用方法时本身没有任何 bug。虽然很少发生，但是这些假设有时还是有可能被违反的。

当我们编写捕获和处理异常的代码时，我们必须清楚这些假设随时都有可能被违反，从而引起 CLR 自身分别抛出 System.StackOverflowException，System.OutOfMemoryException，或者 System.ExecutionEngineException 异常。例如，考虑下面的方法：

```
void InfiniteLoop() {
    while (true) ;
}
```

上面方法中的循环可能会正确执行 1000 次，但是在第 1001 次有可能抛出一个异常。如果 CLR 需要执行垃圾收集，它可能会劫持调用线程(第 19 章有讨论)，并使其调用一个内部函数，从而导致抛出一个 StackOverflowException 异常。

这三个异常和其他大多数异常都不一样，因为它们只有在 CLR 处在非常严重的情况下才会被抛出、并且很难恢复。根据导致异常的情况，我们的代码也许不能捕获到它们，并且 finally 块也可能不会执行。下面的列表描述了这些特殊的异常被抛出时发生的事情：

- **OutOfMemoryException** 该异常在我们试图新建一个对象、而垃圾收集器又找不到任何可用内存时被抛出。在这种情况下，我们的应用程序代码可以成功地捕获到该异常，并且 finally 块也会执行。在 CLR 需要某些内部内存、而又没有找到可用内存的时候也会抛出该异常。在这种情况下，CLR 会向控制台显示一个信息，并且立即中断进程；我们的应用程序不可能捕获到该异常，并且 finally 块中的代码也不会执行。

开发人员几乎从来不会去考虑的隐含假设(续)

因为在CLR内部耗尽内存时不能顺利地恢复，所以我们在用.NET框架设计服务器程序时就要倍加小心。特别地，我们应该有一个独立的守护进程(watchdog process)，当它发现服务器中断运行时，应该自动重启服务器。

- **StackOverflowException** CLR在线程耗尽所有的堆栈空间时会抛出该异常。我们的应用程序可以捕获到该异常，但是finally块不会执行，因为它们需要额外的堆栈空间，而这时已经没有了。另外catch块可能会捕获到该异常(记录某些信息以帮助调试)，这时，我们不应该忽略该异常。原因是应用程序现在处于一种未定义的状态，因为finally块没有执行。任何捕获到StackOverflowException的catch块都应该将它重新抛出，以便让CLR来中断进程。如果堆栈溢出出现在CLR内部，我们的应用程序将不能捕获到StackOverflowException异常，也不会有任何的finally块执行。这种情况下，CLR会连接到一个进程的调试器上；如果没有安装调试器，CLR会中断进程。

- **ExecutionEngineException** CLR在检测到它的内部数据结构毁坏，或者自身的一些bug时会抛出该异常。当CLR抛出该异常时，它会连接到一个进程的调试器上；如果没有安装调试器，它会中断进程。在该异常抛出时，我们的应用程序不会执行任何的catch块或finally块。

顺便提一句，一些其他的线程总是可以调用System.Threading.Thread的Abort方法，来在一个线程中强制抛出System.Threading.ThreadAbortException异常。这是异常可以随时被抛出的又一个例子。

如果我们的类型要用于许多不同的情形，将接口设计的使其适应所有的情况是不可能的。在这种情况下，我们必须尽我们最大的努力，并从用户那里获取反馈信息，从而在设计类型接口的下一个版本时将情况考虑进去。

在开发应用程序时，我们使用的类型接口也可能对我们的情况并不合适，从而可能会导致频繁地违反类型接口的假设。这并不是一件坏事情，因为我们可以捕获异常，然后顺利地恢复并继续执行。例如，我们可能会编写一个应用程序为用户磁盘上的所有文件建立一个索引。要建立这个索引，我们必须打开每个文件，并分析其内容。但是，我们试图访问的某些文件可能是受保护的，因此会阻止我们打开文件。在这种情况下，应用程序应该会得到许多 System.Security.SecurityException 异常。另外，我们不应该将异常总认为是我们自己的错误——我们的应用程序中可能会频繁地抛出异常，其原因只不过是应用程序需要使用另一个开发人员设计的类型。我们的代码可以捕获这些异常，然后做适当的恢复工作后即可继续运行。但是，我们必须清楚系统在搜索处理异常的捕获筛选器时，应用程序的性能会受到一些负面影响。应用程序抛出的异常越少，它运行的速度也将越快。

18.4 System.Exception 类

通用语言运行时(CLR)允许我们将任何类型——Int32、String 等——的任何实例作为一个异常抛出。但是，微软认为不应该强制所有的编程语言都抛出并捕获任何类型的异常。因此微软定义了 System.Exception 类型，并规定所有和 CLS 兼容的编程语言都必须能够抛出并捕获那些继承自 System.Exception 的异常类型。继承自 System.Exception 的异常类型被认为是与 CLS 兼容的。C#和许多其他的语言都允许代码只抛出与 CLS 兼容的异常。

System.Exception 类型是一个很简单的类型，表 18.1 描述了它所包含的一些属性。

表 18.1 System.Exception 类型的属性

属 性	访问权限	类 型	描 述
Message	只读	String	一段辅助性的消息文本，表示异常抛出的原因。该段消息文本应该是本地化后的文本，因为如果应用程序代码没有捕获异常、或者捕获异常的目的是记录它，用户都有可能会看到该段消息文本
Source	读/写	String	包含产生异常的程序集名称
StackTrace	只读	String	包含异常所历经的方法的名称和签名。该属性对于调试工作非常有用
TargetSite	只读	MethodBase	包含抛出异常的方法
HelpLink	只读	String	包含一个指向文档的 URL(例如 file://C:\MyApp\Help.htm#MyExceptionHelp)，它可以帮助用户理解异常
InnerException	只读	Exception	如果当前的异常是在处理另一个异常时产生的，那么该属性表示前一个异常。该属性通常为 null。Exception 类型还提供有一个公有方法 GetBaseException，它可以遍历所有内部异常组成的链表，并返回最开始抛出的那个异常
HResult	读/写	Int32	该属性为一个受保护属性，它仅用于托管代码和非托管 COM 代码互操作的情况

18.5 FCL 定义的异常类

.NET 框架类库定义了许多异常类型(它们都直接或间接继承自 System.Exception)。下面的层次结构展示了 MSCorLib.dll 程序集中定义的异常类型。当然，其他的程序集中还有更多的异常类型。(第20章有产生该层次结构的应用程序。)

```
System.Exception
   System.ApplicationException
      System.Reflection.InvalidFilterCriteriaException
      System.Reflection.TargetException
      System.Reflection.TargetInvocationException
      System.Reflection.TargetParameterCountException
   System.IO.IsolatedStorage.IsolatedStorageException
   System.SystemException
      System.AppDomainUnloadedException
      System.ArgumentException
         System.ArgumentNullException
         System.ArgumentOutOfRangeException
         System.DuplicateWaitObjectException
      System.ArithmeticException
         System.DivideByZeroException
         System.NotFiniteNumberException
         System.OverflowException
      System.ArrayTypeMismatchException
      System.BadImageFormatException
      System.CannotUnloadAppDomainException
      System.ContextMarshalException
      System.ExecutionEngineException
      System.FormatException
         System.Reflection.CustomAttributeFormatException
      System.IndexOutOfRangeException
      System.InvalidCastException
      System.InvalidOperationException
         System.ObjectDisposedException
      System.InvalidProgramException
      System.IO.IOException
         System.IO.DirectoryNotFoundException
         System.IO.EndOfStreamException
         System.IO.FileLoadException
         System.IO.FileNotFoundException
         System.IO.PathTooLongException
      System.MemberAccessException
         System.FieldAccessException
         System.MethodAccessException
```

```
    System.MissingMemberException
      System.MissingFieldException
      System.MissingMethodException
  System.MulticastNotSupportedException
  System.NotImplementedException
  System.NotSupportedException
    System.PlatformNotSupportedException
  System.NullReferenceException
  System.OutOfMemoryException
  System.RankException
  System.Reflection.AmbiguousMatchException
  System.Reflection.ReflectionTypeLoadException
  System.Resources.MissingManifestResourceException
  System.Runtime.InteropServices.ExternalException
    System.Runtime.InteropServices.COMException
    System.Runtime.InteropServices.SEHException
  System.Runtime.InteropServices.InvalidComObjectException
  System.Runtime.InteropServices.InvalidOleVariantTypeException
  System.Runtime.InteropServices.MarshalDirectiveException
  System.Runtime.InteropServices.SafeArrayRankMismatchException
  System.Runtime.InteropServices.SafeArrayTypeMismatchException
  System.Runtime.Remoting.RemotingException
    System.Runtime.Remoting.RemotingTimeoutException
  System.Runtime.Remoting.ServerException
  System.Runtime.Serialization.SerializationException
  System.Security.Cryptography.CryptographicException
    System.Security.Cryptography.CryptographicUnexpectedOperationException
  System.Security.Policy.PolicyException
  System.Security.SecurityException
  System.Security.VerificationException
  System.Security.XmlSyntaxException
  System.StackOverflowException
  System.Threading.SynchronizationLockException
  System.Threading.ThreadAbortException
  System.Threading.ThreadInterruptedException
  System.Threading.ThreadStateException
  System.TypeInitializationException
  System.TypeLoadException
    System.DllNotFoundException
    System.EntryPointNotFoundException
  System.TypeUnloadedException
  System.UnauthorizedAccessException
```

微软认为 Exception 应该是所有异常的基类型，另两个类型 System.SystemException 和 System.ApplicationException 都继承自 Exception。

CLR自身抛出继承自SystemException的异常类型。大多数继承自SystemException的异常(例如DivideByZeroException、InvalidCastException、IndexOutOfRangeException)表示的情况都不是很严重，应用程序一般都能从中恢复过来。但是也有一些异常，如StackOverflowException，表示的情况就比较严重。在遇到这些异常时，我们不应该再尝试进行恢复，因为那样的恢复几乎不可能成功。

另外，FCL类型定义的方法也应该抛出继承自SystemException的异常。例如，所有FCL方法在执行任何操作之前都会验证它们的参数。如果有参数不能满足方法的隐含假设，则将有System.ArgumentNullException、或者System.ArgumentOutOfRangeException、或者System.DuplicateWaitObjectException异常抛出。所有这些异常都继承自ArgumentException。这使得应用程序可以通过捕获ArgumentException来捕获任何一种更具体的参数异常类型。

微软认为ApplicationException类型是一个专门为应用程序使用的保留的基类型。也就是说微软自己定义的异常不会继承自ApplicationException。

但是，如果我们查看异常类型的层次结构，我们会发现微软的开发人员并没有遵循自己拟定的原则。例如，FCL中定义的一些和反射相关的异常类型就继承自ApplicationException。另外，某些异常类型，如IsolatedStorageException则直接继承自Exception、而不是SystemException。

大家可能认为这些“bug”非常糟糕。但是在做出这样的判断之前，大家可能很想知道为什么让CLR/FCL中定义的异常类型继承自SystemException，而让所有应用程序中定义的异常类型继承自ApplicationException。一旦我们开始思考这个问题，不用太长时间我们就会发现这样的层次结构的惟一好处就是允许我们的代码方便地捕获一组相关的异常类型。换句话说，编写代码捕获ArithmeticException要比捕获所有继承自它的异常类型(DivideByZeroException、NotFiniteNumberException和OverflowException)更加容易。

那么现在，大家是否愿意捕获所有继承自SystemException的异常类型、而非捕获所有继承自ApplicationException的异常类型呢？我个人不这么认为。我们有时可能只希望知道是否有异常抛出，这可以通过简单地捕获System.Exception来实现。所以使所有的异常类型都继承自Exception是有意义的。另外让DirectoryNotFoundException、EndOfStreamException、FileLoadException、FileNotFoundException都继承自IOException也是有意义的。

但是，我认为在异常层次结构中定义 SystemException 和 ApplicationException 基类型没有太大价值。实际上，我认为它们只能使人感到迷惑。

另外，我个人认为 ExecutionEngineException 和 StackOverflowException 这两个特殊的异常应该位于一个特殊的层次结构中，原因是它们和其他的异常不同。只有 CLR 本身——应用程序代码永远都不可以——可以抛出这些异常(译注：实际上编译器并不会禁止我们在应用程序中抛出这两个特殊的异常，只是对于应用程序，抛出这些异常的做法是不适当的)，因为应用程序很难从它们之中恢复过来。

18.6 定义自己的异常类

在我们实现自己的方法时，我们可能会遇到某些希望抛出异常的情况。例如我们的非私有方法应该总是对传入的参数进行校验，如果有参数不能满足方法的隐含假设，我们就应该抛出一个异常。在这种情况下，推荐大家抛出 FCL 中定义的下列异常：ArgumentNullException、或 ArgumentOutOfRangeException、或 DuplicateWaitObjectException。

强烈建议大家抛出一个具体的异常类型，即一个没有其他类继承的异常类型。例如，不要抛出一个 ArgumentException，因为它的表意不是很清楚，它可能意味着其他三种派生类型，它也没有为异常的捕获者提供尽可能多的信息。我们永远都不应该抛出 Exception、ApplicationException、或 SystemException 类型。

> **注意** 通过抛出一个我们自己定义的异常类型实例，我们可以使捕获代码精确地知道所发生的事情，并以合适的方式进行恢复。

现在假设我们定义了一个方法，并要求该方法接受的对象引用必须实现 ICloneable 和 IComparable 接口，我们可能会像下面这样实现该方法：

```
class SomeType {
   public void SomeMethod(Object o) {
      if (!((o is ICloneable) && (o is IComparable)))
         throw new MissingInterfaceException(...);

      // 这里执行期望的操作
      ...
   }
}
```

由于 FCL 没有为我们定义合适的异常类型，所以我们必须自己定义一个 MissingInterfaceException 类型。注意，根据约定，一个异常类型的名称应该以“Exception”结尾。当定义该类型时，我们必须确定它的基类型。我们该选择 Exception，还是 ArgumentException，抑或是其他不同的类型？这个问题花费了我数月的思考时间，但是很不幸，我竟然不能得出一个好的答案，下面我将解释其中的原因。

如果我们让 MissingInterfaceException 继承自 ArgumentException，那么所有捕获 ArgumentException 的现存代码都将能够捕获到这个新的异常。有些时候，这是一件好事情。但是，有些时候这也可能是一个 bug。说它是一件好事情是因为希望捕获任何参数异常(通过 ArgumentException)的代码现在也可以自动捕获这个新的参数异常(MissingInterfaceException)。说它是一个 bug 是因为 MissingInterfaceException 表示的是一个新的异常事件，它不是捕获 ArgumentException 异常的代码当初所预期的情况。当我们定义 MissingInterfaceException 类型时，我们可能认为它和 ArgumentException 是类似的，所以应该以同样的方式来处理。但是，这种不可预期的关系可能导致不可预期的行为。

另一方面，如果我们让 MissingInterfaceException 直接继承自 Exception，我们的代码可能会抛出一个应用程序根本不知道的新的异常类型。这很可能成为一个未处理异常而导致应用程序中断。这种行为很容易发生，因为我们违反了一个隐含的假设，而应用程序又没有提供任何补救措施。如果我们捕获了这个新的异常，然后便忽略它并继续执行应用程序，同样可能会产生不可预期的结果。

回答这种应用程序设计的问题更像是一门艺术，而非科学。在定义一个新的异常类型时，我们应该仔细考虑应用程序代码会怎样捕获它(或者它的基类型)，然后选择一个对调用者来说负面影响最小的类型作为它的基类型。

在定义自己的异常类型时，我们可以自由地定义自己的子层次结构，只要它们对于我们所做的事情来说合适就行了。我们可以让它们直接继承自 Exception、或者其他的基类型。另外要确保我们所构造的子层次结构的位置对于调用者来说有意义。作为一个一般的原则，异常层次结构应该宽而且浅：异常类型应该继承自一个与 Exception 相近的类型，并且继承深度一般不应该超过 2 到 3 层。如果我们定义的异常类型不打算作为其他类型的基类型，我们应该将其标识为 sealed。

Exception 基类型定义了三个公有构造器：

- 一个无参(默认)的构造器，它创建一个异常类型的实例，并将所有的字段和属性设为默认值。
- 一个参数为 String 的构造器，它创建一个异常类型的实例，并将异常的消息文本设置为指定的字符串。
- 一个参数为 String 和 Exception 的构造器，它创建一个异常类型的实例，并将异常的消息文本和内部异常分别设为构造器参数中指定的 String 和 Exception。不幸的是，FCL 中很多继承自 Exception 的类型都没有这个构造器。微软会在.NET 框架的后续版本中修复这个疏漏。

当定义自己的异常类型时，我们应该为其实现上述这三个构造器，并调用基类型中相应的构造器。

当然，我们的异常类型会继承 Exception 中定义的字段和属性。另外，我们也可以加入自己的字段和属性。例如，System.ArgumentException 就添加了一个名为 ParamName 的 String 属性。ArgumentException 自然也定义了一个新的构造器(除了上述三个构造器之外)，新构造器接受一个额外的 String 参数来初始化 ParamName 属性，其中 ParamName 属性表示违反方法隐含假设的那个参数的名称。

当捕获到一个 ArgumentException 时，我们可以通过读取 ParamName 属性来判断到底是哪个参数引起了问题。这对于应用程序的调试来说是一个非常棒的特性！如果我们在定义自己的异常类型时也添加了新的字段，我们应该记住也要定义一些额外的构造器来初始化这些字段。

所有的异常类型都应该是可序列化的，因为只有这样异常对象才能在跨越应用程序域(AppDomain)或机器边界时得到封送处理(marshal)、并在客户端重新抛出。使一个异常类型成为可序列化类型也使得我们可以将其保存在一个日志文件或者数据库中。要使新定义的异常类型成为可序列化类型，我们必须在类型上应用一个[Serializable]定制特性，并且如果类型定义了任何新的字段，我们还必须让其实现 ISerializable 接口的 GetObjectData 方法和一个特殊的受保护的构造器(两者的参数都为 SerializationInfo 和 StreamingContext)。(译注：注意这句话是针对那些基类已经实现了 ISerializable 接口，而本身又在基类的基础上添加了新字段的类型而说的。如果一个类型的基类没有实现 ISerializable 接口，那么只需要在其上应用[Serializable]定制特性就可以让其成为可序列化类型。关于对象序列化的更多知识可参见.NET 框架 SDK 文档中的相关内容。)下面的代码演示了如何正确定义自己的异常类型：

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
```

```
// 允许 DiskFullException 的实例被序列化
[Serializable]
sealed class DiskFullException : Exception, ISerializable {
    // 三个公有构造器
    public DiskFullException()
        : base() {       // 调用基类的构造器
    }

    public DiskFullException(String message)
        : base(message) {  // 调用基类的构造器
    }

    public DiskFullException(String message, Exception innerException)
        : base(message, innerException) {   // 调用基类的构造器
    }

    // 定义一个私有字段
    private String diskpath;

    // 定义一个只读属性，该属性返回新定义的字段
    public String DiskPath { get { return diskpath; } }

    // 重写公有属性 Message，将新定义
    // 的字段包含在异常的消息文本中
    public override String Message {
        get {
            String msg = base.Message;
            if (diskpath != null)
                msg += Environment.NewLine + "Disk Path: " + diskpath;
            return msg;
        }
    }

    // 因为定义了至少一个新的字段，所以要定义一个特殊的
    // 构造器用作反序列化。由于该类是一个密封类，所以该
    // 构造器的访问限制被定义为私有方式。否则，该构造器
    // 的访问限制应该被定义为受保护方式
    private DiskFullException(SerializationInfo info,
        StreamingContext context)
        : base(info, context) {     // 让基类反序列化其内定义的字段

        // 反序列化新定义的每个字段
        diskpath = info.GetString("DiskPath");
    }
```

```
    // 因为定义了至少一个字段，所以要
    // 定义序列化方法
    void ISerializable.GetObjectData(SerializationInfo info,
        StreamingContext context) {
        // 序列化新定义的每个字段
        info.AddValue("DiskPath", diskpath);

        // 让基类序列化其内定义的字段
        base.GetObjectData(info, context);
    }

    // 定义额外的构造器设置新定义的字段
    public DiskFullException(String message, String diskpath)
        : this(message) {  // 调用另一个构造器
        this.diskpath = diskpath;
    }

    public DiskFullException(String message, String diskpath,
        Exception innerException)
        : this(message, innerException) {   // 调用另一个构造器
        this.diskpath = diskpath;
    }
}

// 下面的代码测试异常的序列化
class App {
    static void Main() {

        // 构造一个 DiskFullException 对象，并对其进行序列化
        DiskFullException e =
            new DiskFullException("The disk volume is full", @"C:\");
        FileStream fs = new FileStream(@"Test", FileMode.Create);
        IFormatter f = new SoapFormatter();
        f.Serialize(fs, e);
        fs.Close();

        // 反序列化 DiskFullException 对象，并查看它的字段
        fs = new FileStream(@"Test", FileMode.Open);
        e = (DiskFullException) f.Deserialize(fs);
        fs.Close();
        Console.WriteLine("Type: {1}{0}DiskPath: {2}{0}Message: {3}",
            Environment.NewLine, e.GetType(), e.DiskPath, e.Message);
    }
}
```

18.7 如何正确使用异常

理解异常机制当然非常重要，但理解如何正确地使用异常也同样重要。我经常发现一些类库开发人员会去捕获所有的异常，而这往往会阻止应用程序开发人员发现真正的问题。本节为大家提供一些这方面的指导原则——所有的开发人员在使用异常时都应该了解它们。

重要 类库开发人员(那些设计类型供其他应用程序开发人员使用的开发人员)应该严格遵循这里描述的原则。类库开发人员肩负着很大的责任，因为他们设计类型接口时所作的隐含假设要适用于各种各样的应用程序。但是，类库开发人员并不熟悉自己所要调用的代码(通过委托、虚方法、或者接口方法)，也不知道那些调用自己的代码。因为不可能预料到使用类型的每一种场合，所以类库开发人员不能做任何策略抉择，不能认定哪种情况就是错误，而应该把这种判断交给调用者。如果类库开发人员没有遵循本章给出的原则，应用程序开发人员在使用类库中的类型时将注定烦恼无比。

对于应用程序开发人员来说，则可以定义任何自认为合适的策略。遵循本章给出的原则有助于更快地发现并修复代码中的问题，从而增强应用程序的健壮性。但是，如果是经过了仔细的考虑，应用程序开发人员也完全可以背离这些原则并设定自己的策略。比如，应用程序代码在异常捕获方面就可以做得更积极一些。(译注：这里更积极的意思是指将捕获筛选器中的异常类型设置为更高层的基类型，如 System.Exception，来一次捕获多种类型的具体异常。)

18.7.1 避免过多的 finally 块

finally 块非常强大，它允许我们指定一个代码块，不管线程是否抛出异常，该代码块中的代码都能确保被执行。我们应该使用 finally 块来清理那些成功启动的操作，然后再返回给调用者或者继续执行 finally 块之后的代码。我们还会频繁地使用 finally 块来显式释放对象以避免资源泄漏。下面就是这样一个例子，它将所有的清理代码(关闭文件)都放在了一个 finally 块中：

```
class SomeType {
   void SomeMethod() {

      // 打开一个文件
      FileStream fs = new FileStream(@"C:\ReadMe.txt", FileMode.Open);
      try {
         // 显示用100除以文件中第一个字节所得的结果
         Console.WriteLine(100 / fs.ReadByte());
      }
      finally {
         // 将清理代码放在finally块中，从而确保不
         // 管是否出现异常(例如，第一个字节为0)
         // 文件都能得到关闭
         fs.Close();
      }
   }
}
```

确保清理代码的执行是如此重要，以致于许多编程语言都提供了一些构造来简化这种编码。例如，C#就提供了lock和using语句。(本书第19章将会详细解释using语句。)这些语句为开发人员提供了一种简单的语法，它们可以使编译器自动产生try块和finally块，其中finally块包含的就是清理代码。例如，下面的C#代码就利用了using语句。这段代码比上面的代码更短，但是两者编译后产生的结果却是一样的。

```
class SomeType {
   void SomeMethod() {

      // 打开一个文件
      using (FileStream fs =
         new FileStream(@"C:\ReadMe.txt", FileMode.Open)) {

         // 显示用100除以文件中第一个字节所得的结果
         Console.WriteLine(100 / fs.ReadByte());
      }
   }
}
```

18.7.2 避免捕获所有异常

异常处理中一个常见的错误就是catch块使用的太频繁、或者使用的不恰当。当我们捕获一个异常时，我们是在表明该异常是我们所能预料到的，我们理解它为什么出现，并且知道如何处理它。换句话说，我们是在为应用程序定义一种策略。但是，我经常会看到类似下面的代码：

```
catch (Exception) {
   ...
}
```

这段代码表明它可以预料到所有的异常类型，并且知道如何从中恢复。这显然是不可能的。如果我们设计的类型是一个类库的一部分，那么它绝对不应该捕获所有的异常，因为它不可能知道应用程序会如何处理这些异常。另外，这些类型也可能会频繁地通过委托或者虚方法来调用应用程序代码。如果应用程序代码抛出了一个异常，应用程序的另一部分很可能期望捕获该异常。这时我们应该让异常按照筛选器的筛选规则沿着调用堆栈向更高一层传递，直至找到能够处理它的应用程序代码。

大家可能想在代码中的一些特殊的位置放上一些catch块来捕获所有的异常，然后再从中顺利地恢复。添加一个catch块来捕获System.Exception，这的确是一个诱惑。为了演示不可以这么做，我们来考虑一下虚方法Equals(由System.Object定义)。如果两个对象的逻辑值不同，那么该方法将返回false。所以如果有代码试图比较Apple和Orange，Equals应该返回false。大家因此可能会像下面这样来实现Apple的Equals方法(注意这样的实现是错误的)：

```
sealed class Apple : Object {
   Color c = Color.Red;   // Apple的颜色

   public override Boolean Equals(Object o) {
      Boolean equal = false;      // 假设两个对象不相等

      try {
         // 转型为Apple
         Apple a = (Apple) o;

         // 比较'this'和a的字段
         // 如果有字段不相等，则跳出try块
         if (this.c != a.c) goto leave;

         // 如果所有的字段值都相等，则两个对象相等
         equal = true;
```

```
            // 离开try块。注意：C#没有提供leave关键字，
            // 所以我们只能用goto语句跳转到一个标签上
            leave:;
        }
        catch (Exception) {
        }
        return equal;
    }
}
```

仔细查看上面的代码，我们会发现在试图将 o 转型为 Apple 时可能会导致 CLR 抛出一个 InvalidCastException 异常。实际上，这可能是大家能够想到的该方法抛出的惟一一个异常。因为 InvalidCastException 是预料中的惟一一个异常，所以大家可能认为像上面代码中那样捕获 Exception 就可以了。即使还有其他情况导致抛出异常，大家也可能认为捕获 Exception 就足够了。毕竟，Equals 应该返回 true 或者 false。

然而，实际上我们不应该在这里捕获 Exception，这是因为任何时候系统都有可能抛出一个 StackOverflowException 或者 OutOfMemoryException。但在上面的代码中，Equals 将捕获到这两个异常，并简单地返回 false 给调用者。Equals 这么做会隐藏一些非常严重的问题，并使应用程序以一种不可预期的结果向前运行。这当然不是我们希望见到的情形。修复这段代码很简单，只要将捕获筛选器中的 Exception 换为 InvalidCastException 就可以了。InvalidCastException 是我们的代码知道如何从中恢复的惟一一个异常，所有其他的异常都不是 Equals 所预期的，应该让它们沿着方法的调用堆栈继续向上传递。

18.7.3 从异常中顺利地恢复

有时在调用一个方法时，我们已经预料到方法可能会抛出某些异常。因为这些异常是我们所能预期到的一些异常，所以我们希望能有一些代码去执行一些恢复工作。下面是一个伪码描述的例子：

```
public String CalculateSpreadsheetCell(Int32 row, Int32 column) {
    String result;
    try {
        result = /* 计算电子表格单元中的数值 */
    }
    catch (DivideByZeroException) {
        result = "Can't show value: Divide by zero";
    }
    return result;
}
```

这段伪码计算电子表格单元中的内容，并返回给调用者一个表示结果的字符串，这样调用者就可以将其显示在应用程序的窗口中。但是，某个单元格中的内容可能是另两个单元格的数值相除的结果。如果作为分母的单元格的数值为 0，那么 CLR 将抛出一个 DivideByZeroException 异常。在这种情况下，方法会捕获到这个异常，并返回一个特殊的字符串显示给用户。

当我们捕获某个特定的异常时，我们应该完全理解导致抛出异常的情况，并清楚有哪些异常类型继承自我们所捕获的异常。不要去捕获并处理 System.Exception，因为我们不可能知道 try 块中可能抛出的所有异常情况(尤其是考虑 OutOfMemoryException、OverflowException、StackOverflow Exception、ExecutionEngineException 等异常)。(译注：这里举 OverflowException 的例子是不恰当的，因为 OverflowException 是一个很普通的异常，它的抛出原因与捕获处理方式都是有章可循的。)

18.7.4　当异常无法修复时，回滚部分完成的操作

我们常常会遇到这样一些方法，它们需要调用其他几个方法来共同完成一个抽象操作。这些方法中有些可能会成功，有些则可能会失败。例如，某个方法要在两个账户中实现转账，它可能首先要将一笔钱添加到一个账户上，然后再从另一个账户上减去相同数目的金额。但是如果第一次操作成功，而第二次操作却由于某种原因失败了，那么为了保持账目平衡，我们还必须从第一个账户中减去先前添加的金额。

下面的例子可能更有意思：假设我们正在将一组对象序列化到一个磁盘文件中，但是在执行完第 10 个对象的序列化后，却抛出了一个异常。(可能是因为磁盘已满，或者下一个要序列化的对象没有应用 Serializable 定制特性。)这时，我们应该将这个异常留给调用代码处理，但是磁盘文件的状态怎么办呢？因为现在的文件包含了一个部分序列化的对象图，所以它实际上已经处于损坏状态。如果应用程序可以回滚(back out)这些部分完成的操作，使文件回到对象序列化之前所处的状态，那么整个事情就会好许多。下面是对上述内容的一个代码实现：

```
public void SerializeObjectGraph(FileStream fs,
    IFormatter formatter, Object rootObj) {

    // 保存文件当前位置
    Int64 beforeSerialization = fs.Position;
```

```
    try {
        // 尝试将对象图序列化到文件中
        formatter.Serialize(fs, rootObj);
    }
    catch { // 捕获所有与 CLS 兼容以及不兼容的异常
        // 一旦发生错误，则将文件恢复到一个良好的状态
        fs.Position = beforeSerialization;

        // 截断文件
        fs.SetLength(fs.Position);

        // 注意：前面的代码并不位于 finally 块中；因为
        // 只有在序列化失败时，我们才将文件流重新复位

        // 重新抛出同一个异常，让调用者清楚发生的事情
        throw;
    }
}
```

为了正确回滚部分完成的操作，我们的代码应该捕获所有的异常。是的，是捕获所有的异常，因为我们并不关心出现的是哪种错误，我们只需要我们的数据结构保持在一个一致的状态。在捕获并处理完异常后，我们不应该忽略该异常——相反，我们应该让调用代码知道发生了异常。我们可以通过重新抛出它来实现这一点。实际上，C#和许多其他语言都简化了这项任务。我们可以像上面的代码中演示的那样使用 throw 关键字，而且不要在其后指定任何异常对象。

注意上面示例代码中的 catch 块没有指定任何的异常类型，因为我们希望捕获所有与 CLS 兼容以及不兼容的异常。C#允许我们通过在 catch 后不指定任何异常类型来实现这一点，同时我们还要使用 throw 语句来重新抛出任何捕获到的对象。

18.7.5　隐藏实现细节

在某些情况下，大家可能会发现捕获异常后再抛出另外一个不同的异常会非常有用。看下面的例子：

```
public Int32 SomeMethod(Int32 x){
    try {
        return 100 / x;
    }
    catch (DivideByZeroException e) {
        throw new ArgumentOutOfRangeException("x", x, "x can't be 0", e);
    }
}
```

在上面的代码中，SomeMethod 接受一个 Int32 数值，并返回 100 除以该数的结果。在进入该方法后，代码可能会检查 x 是否为 0，如果是，则抛出一个 DivideByZeroException 异常。但是，每次都进行这样的检查会影响代码性能，因为代码本身有一个隐含假设，即 x 很少为 0。所以上面的方法实现首先假设 x 不为 0，并试图拿它去除 100。如果 x 碰巧为 0，那么将会捕获到一个 DivideByZeroException 异常，并重新抛出一个 ArgumentOutOfRangeException 异常。注意 DivideByZeroException 异常通过构造器的第 4 个参数被设为 ArgumentOutOfRangeException 异常的 InnerException 属性。

> **重要**　上面的讨论向大家展示了如何捕获一个异常，然后又抛出另外一个不同的异常。当我们使用这项技巧时，我们应该将原来的异常设为新异常的 InnerException 属性。上面的代码演示了其实现过程。但不幸的是，这段代码不能通过编译。原因在于许多 FCL 中的异常类型都没有提供接受 innerException 参数的构造器。这些构造器的缺失显然是一些 bug，微软已经许诺会在.NET 框架的后续版本中纠正它们。就个人而言，我发现这些 bug 非常令人头疼，而且也找不到另外的解决办法，因为 Exception 没有提供任何设置内部异常的方式。

这里和 18.7.3“从异常中顺利地恢复”一节中讨论的情况非常类似。当我们捕获某个特定的异常时，我们要完全理解导致抛出异常的情况，并清楚有哪些异常类型继承自我们所捕获的异常。

另外，类库开发人员不能捕获 System.Exception 等诸如此类的异常类型。那样做意味着将所有的异常类型都转换成了一个单一的异常类型。这会丢弃所有有意义的信息(异常的类型)，而且重新抛出的单一的异常类型对于实际发生的情况也所知无几。没有这些信息，调用堆栈中居于较高层次的代码就更难以捕获并处理那些具体的异常。(译注：注意这里的解释针对的是“捕获一个异常，然后对之进行某种处理和封装再抛出一个不同的异常”的情况，而不包括“捕获一个异常，然后用 throw 语句将之重新抛出”的情况。实际上，当一个具体的异常被一个更一般的异常捕获筛选器捕获到，然后又用 throw 语句重新抛出时，它的实际类型、也就是动态类型是不会改变的。)我们应该为调用堆栈中居于较高层次的代码提供机会，使它们不仅能够捕获 System.Exception 异常，而且还能够捕获其他一些用作基类型的异常。

基本上而言，捕获一个异常并重新抛出另一个异常的目的是为了提高方法的抽象含义。另外，我们抛出的新的异常类型应该是一个具体的异常(即那些不用作其他异常的基类型的异常)。

下面的伪码演示了一个 PhoneBook 类型定义的、根据姓名查找电话号码的方法：

```
class PhoneBook {
   String pathname;  // 地址簿文件路径名

   // 这里放其他方法

   public String GetPhoneNumber(String name) {
      String phone;
      FileStream fs = null;
      try {
         fs = new FileStream(pathname, FileMode.Open);

         // 这里的代码从 fs 中读取内容，直至找到 name
         phone = /* 找到的电话号码 */
      }
      catch (FileNotFoundException e) {
         // 抛出一个不同的异常，它包含 name，并将原来的
         // 异常设为它的内部异常
         throw new NameNotFoundException(name, e);
      }
      catch (IOException e) {
         // 抛出一个不同的异常，它包含 name，并将原来的
         // 异常设为它的内部异常
         throw new NameNotFoundException(name, e);
      }
      finally {
         if (fs != null) fs.Close();
      }
      return phone;
   }
}
```

该电话薄的数据是从一个文件(而非一个网络连接或者数据库)中获得的。但是，PhoneBook 类型的用户并不知道这一点。所以如果由于某种原因文件未找到，或者不能读取，调用者将会看到一个 FileNotFoundException 或者 IOException，而这些都不是调用者所预期的。换句话说，文件的存在和能够被读取的能力是方法隐含假设的一部分。但是，调用者无需知道这些情况。所以 GetPhoneNumber 方法会捕获这两个异常类型，并抛出一个新的 NameNotFoundException。

这样，调用者既可以从抛出的异常中知道违反了一个隐含假设，同时调用者又可以从 NameNotFoundException 类型中得到被违反的隐含假设的抽象视图。另外，将 NameNotFoundException 的内部异常设为 FileNotFoundException 或 IOException 非常重要，因为这样不会丢失导致异常的真正原因；清楚导致异常的原因对 PhoneBook 类型的开发人员来说非常有用。

现在，假设 PhoneBook 类型的实现过程与前面稍微有一些不同。假设该类型有一个公有属性 PhoneBookPathname，用户可以通过它来设置或者查询搜索电话号码的文件路径名。因为现在用户知道电话数据来自一个文件，所以我们应该修改 GetPhoneNumber 方法，使其不再捕获任何异常。相反，我们应该让抛出的所有异常都沿着方法的调用堆栈继续向上传递。注意，我们并没有改变 GetPhoneNumber 方法的任何参数，只是改变了 PhoneBook 类型面向用户的抽象方式。

再强调一遍，类库开发人员应该严格遵循这里讲述的所有设计原则，而应用程序开发人员则可以设定自己认为合适的策略，遵循本章给出的设计原则可能会有助于尽快发现代码中的问题、并修正它们，从而使应用程序更加健壮。但应用程序开发人员在仔细考虑之后也完全可以背离这些原则，并设定自己认为最好的策略。比如，应用程序在捕获异常方面就可以做的更积极一些。

18.8 FCL 中存在的一些问题

前面向大家提供了一些异常处理方面的建议，这些建议主要来自于和众多开发人员的交流以及我自己多年的编程经验。前一节中曾经提到过很多 FCL 中的异常类型都缺少一个设置内部异常的构造器，这仅仅是 FCL 中存在的一种 bug。实际上，FCL 中包含有许多与异常处理相关的 bug。本节的目的是让大家了解这些 bug 的存在，从而免得大家在遇到这些问题时再像我一样浪费大量的时间去查找原因。

如前所述，微软并没有遵循本章描述的许多原则。实际上，微软的许多代码甚至违反了微软自己的原则，这使得我们在使用 FCL 时会遇到很多困难。另外，FCL 文档也没有完整描述开发人员会遇到的所有异常、以及从一些异常中恢复的办法，这些都使得问题变得更加复杂。

FCL 的第一个问题是其中的很多代码都具有如下的构造：

```
catch { ... }

catch (Exception) { ... }

catch (ApplicationException) { ... }
```

前面曾经解释过，这些异常是不应该被一个类库所捕获并忽略的。

这里有一个例子：System.IO.Directory 和 System.IO.File 类型都有一个静态方法 Exists。该方法会根据传入的路径参数是否标识了用户磁盘驱动器上的一个目录/文件，来判断该返回 true 还是 false。如果这些方法内部出现问题，它们会捕获 Exception，然后将 false 返回给调用者。这样调用者将不会知道异常的发生是由于文件不存在，还是由于调用者没有足够的权限访问该目录或文件。另外，如果系统抛出了一个 StackOverflowException 或 OutOfMemoryException，方法也会返回 false！

第二个问题是，一些 FCL 代码会频繁地捕获一个异常，然后又抛出一个新的异常。前面讲过，如果新异常能够提供必要的信息，这样做可能比较有用，但是 FCL 通常会对应用程序开发人员隐藏实际发生的情况。

例如，System.Array 类型的 Sort 方法通过调用每个对象的 CompareTo 方法来进行对象数组的排序。如果 CompareTo 方法抛出了任何一个异常，Sort 方法都会捕获到它，并抛出一个新的 InvalidOperationException。这太可怕了！这意味着我们的代码可能抛出一个它永远无法捕获的异常。Array 的 Sort 方法没有理由这么做。顺便提一下，Array 的 BinarySearch 方法也会捕获任何的异常，并抛出一个新的 InvalidOperationException，这同样会隐藏很大的麻烦。

FCL 的第三个问题是其处理反射(本书第 20 章将予以详细讨论)的方式。如果我们获得了一个方法的 MethodInfo，并试图调用其 Invoke 方法，而真正被调用的方法又有可能会抛出异常。非常不幸，MethodInfo 的 Invoke 方法会捕获这其中的任何异常，并抛出一个新的 System.Reflection.TargetInvocationException。因此，我们将无法捕获到真正被调用的方法所抛出的异常。

第三个问题的另外一个例子是，FCL 代码会频繁地捕获一个异常，然后抛出一个新的 System.Exception。由于调用者必须先捕获 Exception，然后才能查明错误原因，这就有可能导致真正有用的信息被丢失。

下面是 FCL 代码中存在的又一个问题：System.Windows.Forms.DataGrid 控件有一个公有属性 CurrentCell。如果我们的代码在试图设置当前单元格的值的时候抛出了一个异常，DataGrid 控件会捕获所有继承自 Exception 的异常，然后显示一个 Windows 消息框！当我发现这个 bug 时，我简直不敢相信自己的眼睛！不仅我们无法捕获该异常，用户也会被这个连我们的应用程序都无法控制的消息框给搞懵的！

除了上述问题之外，我经常还会遇到其他一些问题：当我编写代码处理一些操作时，我发现有些代码不能按预期的那样运行，找了半天才发现是 FCL 中的代码忽略了我的异常。

我热切希望微软能够仔细检查FCL的源代码，并纠正这些错误，使它们遵循本章给出的设计原则。

经验是获知哪些FCL方法会导致问题的最好方式。解决这些问题的惟一方法就是定义一些FCL代码不能捕获的异常类型。

大家因此可能会认为在这种情况下应该定义并抛出一些和CLS不兼容的异常类型，因为很少有FCL代码会去捕获它们。但是，我并不推荐这样做，原因如下：

- 使用其他语言编程的开发人员可能捕获不到与CLS不兼容的异常。
- C#和许多其他语言都不允许我们抛出与CLS不兼容的异常。
- C#和许多其他语言虽然允许我们捕获与CLS不兼容的异常，但是它们不允许我们获得这些异常的任何信息。我们不能判断这些异常的类型是什么，也不能获得它们的堆栈踪迹以及辅助性的描述信息。

即使我们使用了与CLS不兼容的异常，我们也不能保证FCL不会捕获并忽略它们、或者抛出一个新的异常类型。除非微软修正这些FCL异常处理代码，否则我们不可能找到一个满意的解决方案。

18.9 性能考虑

开发人员社区经常会就异常处理的性能问题展开一些积极的争论。我个人的经验表明异常处理的好处远远超出了它所带来的任何性能损失。本节将阐述一些和异常处理相关的性能问题。

比较异常处理和常用的异常报告(HRESULT、特殊返回值代码、等等)之间的性能差别通常是很困难的。如果在每次方法调用时我们都要自己编写代码来检查它们的返回值，并将其返回给调用者，那么我们的应用程序性能将会受到严重的影响。就算不考虑性能，我们必须为此编写的额外代码和潜在的错误也会大幅度增加。异常处理相对来说则是一种更好的选择。

我们知道，非托管C++编译器一方面必须产生一些代码来追踪那些被成功构造的对象。另一方面，它还必须产生一些代码以便在捕获到异常后为每个成功构造的对象调用它们的析构器。编译器在承担这一责任的同时，也为我们的应用程序引入了许多额外的簿记代码，这无疑会对应用程序的代码尺寸和执行时间造成很大的负面影响。

但是，这些工作对托管编译器来说就简单多了，因为托管对象的内存是在托管堆中分配的，而托管堆受垃圾收集器的监控。如果一个对象被构造成功后抛出了一个异常，垃圾收集器会确保该对象的内存最终得到回收。编译器不需要产生任何簿记代码来追踪哪些对象被成功构造，并确保调用它们的析构器(尤其是在托管对象的销毁时刻为非确定的情况下)。与非托管 C++相比，这意味着编译器产生的代码更少，运行时要执行的代码也更少，应用程序的性能自然会更好。

多年来，我一直在各种不同的语言、不同的操作系统和不同的 CPU 结构下使用异常处理。每种情况的异常处理实现都有所不同，每种实现在性能方面都各有利弊。一些实现将异常处理构造直接编译到一个方法中。一些实现将与异常处理相关的信息存储在一个与方法关联的数据表中——只有在抛出一个异常时该表才能被访问。一些编译器不能内联包含异常处理器的方法。而另一些编译器在方法包含异常处理器的情况下无法将变量存放到寄存器中。

问题在于我们无法判断使用异常处理到底会给应用程序增加多大的额外开销。在托管世界里这会变得更加困难，因为我们的程序集代码可以运行在任何支持.NET 框架的平台上。同样的一段管理异常处理的代码，将它放在一台 x86 机器上、或一台 IA64 机器上、或.NET Compact Framework 上 JIT 编译运行，最终得到的代码性能会有很大的不同。

实际上，我已经使用了几个微软内部提供的 JIT 编译器对自己的代码进行了一些测试，结果显示的一些性能差别非常之大，令人吃惊。所以我们必须在各种期望的用户平台上测试我们的代码，并根据得到的结果进行相应的调整。再说一遍，我并不担心使用异常处理所带来的一些性能问题。如前所述，异常处理带来的好处远远超过了任何它所引入的负面影响。

如果大家希望了解异常处理对代码性能所产生的影响，则可以使用 PefMon.exe 或者 Windows NT 4、Windows 2000、Windows XP 和 Windows .NET 服务器产品家族附带的 System Monitor ActiveX 控件。图 18.1 显示了安装.NET 框架时所安装的与异常相关的性能计数器。

图 18.1 PerfMon.exe 中显示的.NET CLR 异常计数器

下面列出了每个计数器的含义：

- **# Of Exceps Thrown** 显示自应用程序启动以来抛出的异常总数目。这其中不但包括.NET 异常，还包括转换为.NET 异常的非托管异常。例如，非托管代码中的空指针引用将在托管代码中作为一个 System NullReferenceException 异常重新抛出。该计数器既包括已处理的异常，也包括未处理的异常。重新抛出的异常会被再次计数。
- **# Of Exceps Thrown/Sec** 显示每秒抛出的异常个数。这其中不但包括.NET 异常，还包括转换为.NET 异常的非托管异常。例如，非托管代码中的空指针引用将在托管代码中作为一个 System.NullReferenceException 异常重新抛出。该计数器既包括已处理的异常，也包括未处理异常。该计数器用于指示由于抛出大量(>100)异常而导致的潜在性能问题。该计数器显示的不是一个时间段上的平均值，而是最后两次采样值之差除以采样间隔得到的结果。
- **# Of Filters/Sec** 显示每秒执行的.NET 异常筛选器的数目。异常筛选器评估一个异常是否应该被处理。该计数器追踪异常筛选器执行评估的比率，它并不关心异常是否被处理。

和前面的计数器一样，该计数器显示的也不是一个时间段上的平均值，而是最后两次采样值之差除以采样间隔得到的结果。

- **# Of Finallys/Sec** 显示每秒执行的 finally 块的数目。不管 try 块是怎样退出的，finally 块都会确保执行。只有被执行的 finally 块才会被计数，通常代码路径上的 finally 块不会被计数。同样，该计数器显示的也不是一个时间段上的平均值，而是最后两次采样值之差除以采样间隔得到的结果。
- **Throw To Catch Depth/Sec** 显示每秒遍历的栈帧(stack frame)数目，遍历从抛出异常的栈帧开始，到处理异常的栈帧结束。当进入异常处理器时计数器被清零，所以嵌套异常将显示的是从一个异常处理器到另一个异常处理器之间的堆栈深度。同样，该计数器显示的也不是一个时间段上的平均值，而是最后两次采样值之差除以采样间隔得到的结果。

18.10 捕获筛选器

当一个异常被抛出时，CLR 会向上遍历整个调用堆栈，查看每个 catch 块的捕获筛选器——即 catch 关键字后指定的异常类型。下面的代码演示了一个带有三个 catch 块的 try 块。

```
public void SomeMethod() {
    try {
        // 这里执行一些操作
    }
    catch (NullReferenceException e) {
        // 处理一个空引用异常
    }
    catch (InvalidCastException e) {
        // 处理一个无效转型异常
    }
    catch {   // 在 C#中，该筛选器会捕获任何异常
        // 处理所有异常
    }
}
```

当编译上面的代码时，编译器会为 SomeMethod 方法中的每一个 catch 块产生一个“捕获筛选器 funclet”(译注：这里的 funclet 是用名词“function”和后缀“let”构造而得的一个词，在这里我们可以将其理解为一种微缩变型的函数)。当有异常被抛出时，CLR 会首先调用 NullReferenceException 捕获筛选器 funclet，并将抛出的异常对象传递给它。

捕获筛选器 funclet 检查传入的对象是否为一个 NullReferenceException、或者继承自 NullReferenceException 的类型。如果两者都不是，捕获筛选器 funclet 会返回一个特殊的值告诉 CLR 继续搜索。在上面的例子中，接下来遇到的是 InvalidCastException 捕获筛选器 funclet，最后是“捕获所有异常”的筛选器 funclet。如果没有“捕获所有异常”的 catch 块，CLR 将继续向上遍历整个调用堆栈。

如果捕获筛选器 funclet 能够识别被抛出的异常类型，那么它会返回一个特殊的值通知 CLR。CLR 首先执行更低层次的堆栈中所有必要的 finally 块来清理其中启动的操作，展开(unwind)调用堆栈。然后，CLR 才执行与抛出异常类型相匹配的 catch 块中的代码。

在 C#和许多其他面向.NET 框架的语言中，捕获筛选器就是一个数据类型。捕获筛选器 funclet 会判断抛出的异常类型是否和筛选器指定的类型相匹配。然而，CLR 还支持更复杂的捕获筛选器。Visual Basic、托管扩展 C++以及 IL 汇编语言是目前我所知道的支持更复杂的捕获筛选器的仅有的几门语言。下面的 Visual Basic 代码演示了一个更复杂的捕获筛选器：

```
Imports System

Public Module MainMod
   Function HundredDivX(x as Int32) As String
      Try
         x = 100 / x
         HundredDivX = x.ToString()

         Dim a as Object
         Console.WriteLine(a.ToString())

      Catch e as Exception When x = 0
         HundredDivX = "Problem: x was 0"

      Catch e as Exception
         HundredDivX = "Problem: I don't know what the problem is"

      End Try
   End Function

   Sub Main()
      Console.WriteLine(HundredDivX(0))
      Console.WriteLine(HundredDivX(2))
   End Sub
End Module
```

编译并运行上面的代码，我们会得到以下输出：

```
Problem: x was 0
Problem: I don't know what the problem is
```

下面是对这段代码的解释。Main 开始执行时，它会首先调用 HundredDivX 并为其传递参数 0。在 HundredDivX 内部，100 被 0 除，这将导致 CLR 抛出一个 OverflowException 异常。(抛出一个 OverflowException 异常，而不是一个 DivideByZeroException 异常是因为 Visual Basic 将 100 看做是一个 8 字节的实数，而不是一个整数。)(译注：这里的说法是不正确的，Visual Basic 实际上会将 100 看做是一个 4 字节的整数，调用 HundredDivX 并为其传递参数 0 时会导致 CLR 抛出一个 DivideByZeroException 异常。)

当异常被抛出时，CLR 会调用与第一个 Catch 块相关联的捕获筛选器 funclet(由编译器产生)。该捕获筛选器 funclet 首先检查 OverflowException 对象(译注：应该为 DivideByZeroException 对象)是否继承自 Exception，答案显然是肯定的。接着，捕获筛选器 funclet 使用 Visual Basic 的 When 语句来判断 x 是否为 0，答案同样也是肯定的。这时，捕获筛选器 funclet 会返回一个值告诉 CLR 它将处理该异常。如前所述，CLR 首先要执行更低层次的堆栈中所有必要的 Finally 块来展开(unwind)调用堆栈。但在本例中没有这样的 Finally 块。所以 CLR 会直接执行 Catch 块中的代码，将返回字符串设置为"Problem: x was 0"。最后线程先跳出 Catch 块，然后再跳出 Try 块，并从方法中返回。

现在执行线程又回到 Main 中，再次调用 HundredDivX，但这一次为其传递的参数为 2。在 HundredDivX 中，100 被 2 除，结果得到 x 为 50，这里没有抛出任何异常。但是程序接着又声明了一个 System.Object 引用变量 a，并将其初始化为 Nothing，然后又试图在其上调用 ToString 方法，这显然会导致 CLR 抛出一个 NullReferenceException 异常。这时，第一个捕获筛选器 funclet 会去检查抛出的异常类型，并发现第一个条件匹配。但是因为 x 现在为 50(而非为 0)，即第二个条件不匹配，所以捕获筛选器 funclet 会告诉 CLR 继续搜索。

现在 CLR 开始检查第二个捕获筛选器。因为 NullReferenceException 继承自 Exception，并且第二个捕获筛选器没有 When 语句，所以这个捕获筛选器 funclet 会返回一个特殊的值告诉 CLR 它将处理该异常。同样，CLR 首先要执行更低层次的堆栈中可能存在的 Finally 块，显然这里也没有。所以 CLR 会直接去执行 Catch 块中的代码，将返回字符串设置为"Problem: I don't know what the problem is"。接着线程先跳出 Catch 块，然后再跳出 Try 块，并最终从方法中返回。

从本节中，我们应该得到两条重要的启示：

- CLR 支持复杂的筛选器，但是许多语言却不支持。如果我们使用的语言不支持复杂筛选器，但是我们又希望使用它们，我们可以使用 Visual Basic、IL 汇编语言、或者其他支持复杂筛选器的语言来实现这部分代码。因为在.NET 框架中，各种语言编写的代码可以无缝地集成，所以这是一个可行的解决方案。

- 注意前面代码中所演示的 CLR 管理异常的方式。首先，CLR 会去查找接受被抛出异常的捕获筛选器。接着，它会调用更低层次的堆栈中的 Finally 块(如果存在的话)来展开调用堆栈。最后，CLR 才会执行匹配的 Catch 块中的代码。

18.11 未处理异常

前一节曾经解释过，当一个异常被抛出时，CLR 会搜索能够处理该异常的捕获筛选器。如果没有捕获筛选器接受该异常对象，那么将出现一个未处理异常(unhandled exception)。未处理异常表明一种应用程序未曾预料到的情况。

在开发应用程序时，我们应该为未处理异常设置一种策略。为应用程序的调试版本和发行版本分别设置两种不同的策略是比较常见的做法。通常在一个调试版本中，我们希望能够启动调试器，并将其连接到我们的应用程序上，这样我们就可以精确地定位出错的地方，并进而修正我们的代码。

对于发行版本，未处理异常通常是在用户使用应用程序的过程中出现的。用户可能没有调试应用程序的相关知识(或者源代码)，因此在这种情况下最好的解决方法是将未处理异常的有关信息记录下来，并尽可能地使应用程序顺利地从异常中恢复过来。对于客户端应用程序，这可能意味着先要尽力去保存用户的数据，然后再中断应用程序。对于服务器端应用程序，这可能意味着要中断当前用户的请求，并为下一次客户端请求做准备——通常情况下，服务器端应用程序中发生的未处理异常不应该中断服务器。

> **注意** 类库开发人员不应该为未处理异常设置任何策略。应用程序开发人员应该对定义和实现这种策略拥有完全的控制。

我个人很喜欢微软的 Office XP 在遇到未处理异常时的做法——虽然它们不是.NET 框架应用程序。在遇到未处理异常时，它们首先会保存用户目前正在编辑的文档，然后显示一个对话框通知用户发生了问题。图 18.2 显示了一个 PowerPoint 中出现的对话框。

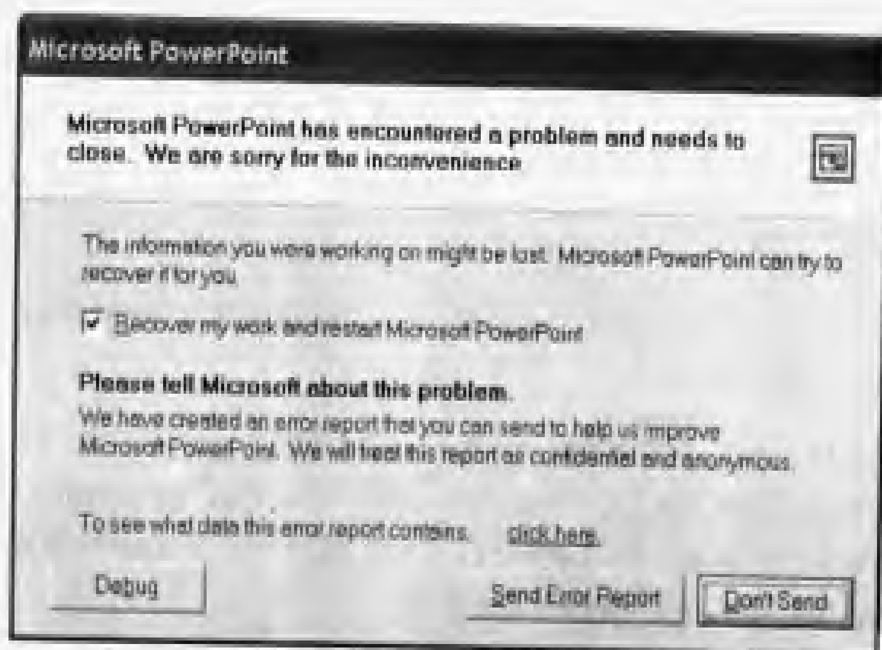


图 18.2 Microsoft PowerPoint 中出现的未处理异常对话框

其中 Send Error Report 按钮允许用户通过互联网给微软发送一份错误报告。Recover My Work And Restart Microsoft PowerPoint 复选框为用户提供了重新启动应用程序的选择(在点击 Send Error Report 或 Don't Send 按钮之后)。如果应用程序重新启动，它将自动重新加载先前正在编辑的文档。如果机器安装有调试器，Debug 按钮将出现在对话框中，它允许用户调试应用程序。本章稍后将讨论一个应用程序如何为未处理异常设立一种类似于 Office XP 的策略。

下面的例子描述了另一种解决未处理异常的策略：当客户端请求一个远程对象或者远程服务时，服务器端的一个线程(从一个线程池中)将被唤醒以处理该请求。服务器代码应该被放在一个 try 块中执行，该 try 块要有一个关联的 catch 块来捕获所有的异常。如果 catch 块捕获到一个异常，那么该异常有关的信息(包括堆栈踪迹)将被作为服务器的响应发送给客户端。这种策略的可行之处在于异常对象是可被序列化的。本章前面 18.6“定义自己的异常类”一节中曾经探讨过这个问题。

当客户端代码检测到有异常从服务器返回时，它将对异常对象执行反序列化，然后将其抛出。这将使得客户端代码能够捕获服务器抛出的异常——这一点非常的酷！整个过程有效地隐藏了应用程序域(AppDomain)、进程以及机器边界，实际上客户端代码甚至完全可以认为远程调用就发生在本地。

在考虑未处理异常时，我们应该清楚我们正在处理的为何种线程。总共有五种线程：

- **主线程** 执行控制台应用程序(CUI)、或 Windows 窗体应用程序(GUI)的 Main 方法的那个线程为一个托管主线程。
- **手工线程** 应用程序代码使用 System.Threading.Thread 对象显式创建的线程称为手工线程(manual thread)。
- **线程池线程** .NET 框架中的许多特性(feature)都利用了内建于 CLR 中的线程池(thread pool)。线程池线程典型地执行由 System.Threading.ThreadPool 类或者 System.Threading.Timer 类中的方法启动的代码。另外，提供 CLR 异步编程模型的方法(例如一个委托的 BeginInvoke 和 EndInvoke 方法)也使用的也是线程池线程。
- **终止化器线程** 在垃圾收集器判定对象为不可达时，托管堆有一个线程专门执行对象的 Finalize 方法。
- **非托管线程** 一些线程创建时无需知道任何 CLR 的知识。使用 P/Invoke 或者 COM 互操作，这些非托管线程可以跳转到 CLR 中执行托管代码。因为 CLR 本身没有创建这些线程，所以任何未处理异常都将在 CLR 之外抛出，非托管线程代码可以以自己的方式来处理这些未处理异常。CLR 与此毫无干涉。

对于上述五种线程，我们可以用类似下面的代码来实现自己的未处理异常策略：

```
using System;
using System.Diagnostics;
using System.Windows.Forms;

class App {
   static void Main() {

      // 在当前的应用程序域上注册 MgdUEFilter
      // 回调方法以使出现未处理异常时它会得到调用
      AppDomain.CurrentDomain.UnhandledException +=
         new UnhandledExceptionEventHandler(MgdUEPolicy);

      // 应用程序代码的其余部分
      try {
         // 模拟一个异常用作测试
         Object o = null;
```

```
            o.GetType();    // 抛出一个 NullReferenceException
        }
        finally {
            Console.WriteLine("In finally");
        }
    }

///////////////////////////////////////////////////////////////////////////////

    // 下面的方法在遇到一个未处理异常时会被调用
    static void MgdUEPolicy(object sender, UnhandledExceptionEventArgs e) {
        // 该字符串包含显示或者记录异常的信息
        String info;

        // 初始化字符串的内容
        Exception ex = e.ExceptionObject as Exception;
        if (ex != null) {
            // 抛出一个与 CLS 兼容的未处理异常，可以
            // 在这里访问 Exception 的属性(Message、
            // StackTrace、HelpLink、InnerException 等)
            info = ex.ToString();
        } else {
            // 抛出一个与 CLS 不兼容的未处理异常，
            // 可以在这里调用 Object 定义的方法
            // (ToString、GetType、等)
            info = String.Format(
                "Non-CLS-Compliant exception: Type={0},String={1}",
                e.ExceptionObject.GetType(), e.ExceptionObject.ToString());
        }

        #if DEBUG

        // 对于调试版的应用程序，启动调试器
        // 查看发生的异常，并修复它
        if (!e.IsTerminating) {
            // 出现在一个线程池线程或者一个终止化器线程
            // (译注：还包括手工线程)内的未处理异常
            Debugger.Launch();

        } else {
            // 出现在一个托管线程(译注：还包括非托管线程)
            // 中的未处理异常。默认情况下，CLR 将自动连接
            // 到调试器上，但我们可以用下面一行强制这样做
            Debugger.Launch();
        }

        #else
```

```
        // 对于发行版的应用程序，显示或者记录异常以
        // 使用户可以将其反馈给我们
        if (!e.IsTerminating) {
            // 出现在一个线程池线程或者一个终止化器线程
            // (译注：还包括手工线程)内的未处理异常。对
            // 于这些异常，我们可能只需要记录异常，而无
            // 需将问题显示给用户。但是每个应用程序都应
            // 选择对自己来说最有意义的方式

        } else {

            // 出现在一个托管线程(译注：还包括非托管线程)
            // 中的未处理异常。CLR将中断应用程序，我们应
            // 该显示或者记录该异常
            String msg = String.Format("{0} has encountered a problem and " +
                "needs to close. We are sorry for the inconvenience.\n\n" +
                "Please tell {1} about this problem.\n" +
                "We have created an error report that you can send to " +
                "help us improve {0}. " +
                "We will treat this report as confidential and anonymous.\n\n" +
                "Would you like to send the report?",
                "(AppName)", "(CompanyName)");

            if (MessageBox.Show(msg, "(AppName)",
                MessageBoxButtons.YesNo) == DialogResult.Yes) {
                // 用户选择将错误报告发送给我们
                // 将info和所有有助于解决问题的
                // 信息都发送给我们

                // 处于测试目的，这里仅仅显示info
                MessageBox.Show(info, "Error Report");
            }
        }
        #endif
    }
}
```

在上面的代码中，应用程序首先在初始化阶段(Main 方法内)构造了一个System.UnhandledExceptionEventHandler 委托来封装静态方法 MgdUEPolicy。该委托随后被登记到System.AppDomain 类型提供的 UnhandledException 事件上。这样不管何时某个线程中出现了未处理异常，CLR 都将会去调用 MgdUEPolicy 方法。但是，如果是在一个非托管线程中出现了一个由非托管代码抛出的未处理异常，CLR 将不会调用 MgdUEPolicy 方法。

对于发行版的应用程序来说，MgdUEPolicy 方法应该将与未处理异常有关的信息(包括堆栈踪迹)显示或者记录下来，以便将它们发送回开发该应用程序的公司中。

这其中可能包括通过互联网来传送异常信息(类似于 Office XP 中的做法)，从而避免用户在报告问题上再花费时间。开发人员接到报告后，便可以修复出现问题的代码，以使应用程序在下一个版本中能够正确地预期这样的异常，并做出合适的反应。对于调试版的应用程序来说，在遇到未处理异常时，调试器应该启动并将自己连接在进程上，这样开发人员便可以判断出现异常的原因并及时修正代码。

回调方法 MgdUEPolicy 被认为是一个捕获筛选器，该方法的执行说明没有更深的捕获筛选器(译注：这里“更深”的意思是指堆栈与堆栈之间的相对位置，“更深的捕获筛选器”是指“更深的堆栈”中的捕获筛选器；而“更深的堆栈”等同于本章前面所述的“更低层次的堆栈”)再接受异常，因此还没有任何的 finally 块被执行过。(译注：这里所说的 finally 块是有范围的，这个范围从异常抛出的位置开始，到最顶层的调用堆栈结束。同样我们要从堆栈结构的角度，而不要从代码路径的角度来理解这个范围。)另外要注意该回调方法的其中一个参数为一个 System.UnhandledExceptionEventArgs 对象。该对象有两个公有只读属性，即 ExceptionObject(类型为 System.Object)和 IsTerminating(类型为 System.Boolean)。ExceptionObject 表示抛出的异常对象。注意 ExceptionObject 的类型为 Object，而不是 Exception，原因是抛出的对象可能与 CLS 不兼容。

IsTerminating 属性告诉我们 CLR 是否会因为这个未处理异常而中断应用程序。通常情况下，对于手工线程、线程池线程、终止化器线程，CLR 会忽略任何的未处理异常，然后或者中断线程、或者将线程返回到线程池中、或者转而调用下一个对象的 Finalize 方法。如果是上面这三种线程中出现了未处理异常，那么 IsTerminating 属性将返回 false。如果是应用程序主线程或者一个非托管线程中出现了未处理异常，那么 IsTerminating 属性将返回 true。

如果大家希望在一个非中断(nonterminating)线程(译注：即手工线程、线程池线程、终止化器线程)中出现未处理异常时能够调试自己的应用程序，那么应该在 MgdUEPolicy 方法中调用 System.Diagnostics.Debugger 类型的静态方法 Launch。上面的示例代码也演示了这一做法。

对于主线程或者非托管线程，系统或者将一个调试器连接到应用程序域上，或者终止当前进程。这也是 IsTerminating 属性返回 true 的原因。

18.11.1 发生未处理异常时的 CLR 行为控制

当一个托管线程中出现未处理异常时，CLR 会检测某些设置以确定其是否应该启动调试器。具体而言 CLR 会检测下面的注册表子键中的 DbgJITDebugLaunchSetting 值来进行判断：HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework。如果存在该值，它的值必须是表 18.2 所列出的其中之一。

表 18.2　DbgJITDebugLaunchSetting 的可选值

值	描　述
0	显示一个对话框询问用户是否愿意调试进程。如果用户选择否，CLR将触发AppDomain的UnhandledException事件。如果异常出现在主线程、或者非托管线程内，CLR随后会中断进程以及其内所有的应用程序域。如果AppDomain的UnhandledException事件没有登记任何回调方法，并且如果进程是一个CUI应用程序，CLR将会把堆栈跟踪显示到控制台上。如果用户选择调试应用程序，CLR会启动一个调试器，并将其连接到当前的应用程序域上。CLR通过查看同一注册表子键下的DbgManagedDebugger值来确定启动调试器的命令行
1	不向用户显示任何对话框。CLR会触发AppDomain的UnhandledException事件。如果异常出现在主线程、或者非托管线程内，CLR随后会中断进程以及其内所有的应用程序域。如果AppDomain的UnhandledException事件没有登记任何回调方法，并且如果进程是一个CUI应用程序，CLR将会把堆栈踪迹显示到控制台上
2	不向用户显示任何对话框。相反，CLR只是启动调试器并将其连接到应用程序上

默认情况下，对于应用程序的主线程或非托管线程，CLR 会中断应用程序，或者启动一个调试器。但对于手工线程、线程池线程、或者终止化器线程，CLR 只是忽略它们，并允许线程继续运行——CLR 并不会中断当前进程。

为了帮助调试，大家可能希望在任何线程中出现未处理异常时都能得到通知。要实现这一点，我们可以将 DbgJITDebugLaunchSetting 注册表值的最高三个字节设置为 0xFFFFFF。同时将最低位的一个字节设为表 18.2 列出的 0、1、或者 2。

注意　我个人认为只有在 AppDomain 的 UnhandledException 事件没有登记任何回调方法时 CLR 才应该去检查注册表值。目前的问题是只要出现未处理异常，CLR 都会去检测 DbgJITDebugLaunchSetting 注册表值的最低位字节。如果该字节为 0 或者 2，那么 CLR 将显示给用户一个对话框、或者直接启动调试器——即使应用程序登记了 AppDomain 的 UnhandledException 事件也不例外。我希望将来能够看到只有在 AppDomain 的 UnhandledException 事件没有登记任何回调方法时、CLR 才应该去考虑未处理异常。

18.11.2 未处理异常与 Windows 窗体

在 Windows 窗体应用程序中，System.Windows.Forms.Application 类的静态方法 Run 负责线程的消息循环。该循环将窗体消息分发给 System.Windows.Froms.NativeWindow 类型提供的一个私有方法。该私有方法在内部创建了一个 try/catch 块，并在 try 块内调用了 System.Windows.Froms.Form 类的受保护方法 WndProc。

如果一个继承自 System.Exception 的未处理异常出现在窗体消息的处理过程中，那么 catch 块将会调用该窗体类型的虚方法 OnThreadException，并将异常对象传递给它。System.Windows.Forms.Control 类型实现的 OnThreadException 又会调用 Application 的 OnThreadException 方法(译注：实际上 OnThreadException 并非 System.Windows.Forms.Control 类型的方法，相反它是 System.Windows.Forms.NativeWindow 类型中定义的一个方法，只是 Control 类型中有一个 NativeWindow 类型的实例字段)。默认情况下，该方法将显示一个类似于图 18.3 所示的对话框。

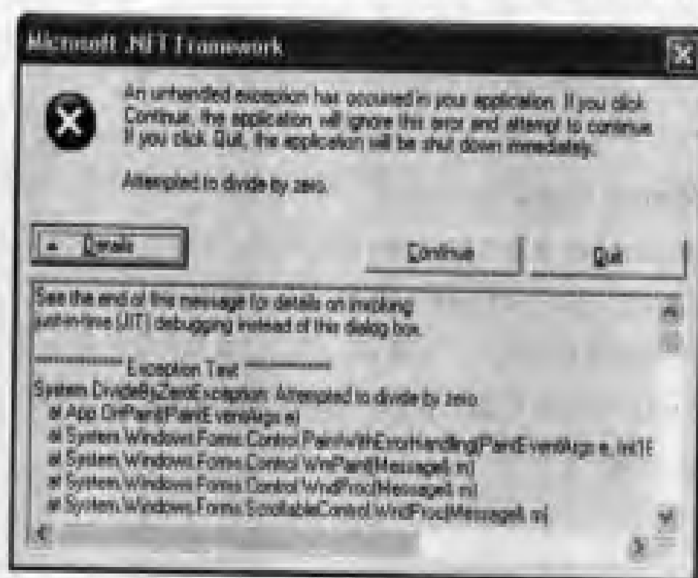


图 18.3 一个窗口过程中的未处理异常导致 Windows 窗体显示该对话框

当一个窗口过程(window procedure)中出现与 CLS 兼容的未处理异常时，该对话框会显示给用户，通知用户发生了未处理异常，并让用户选择是忽略异常继续运行、还是退出应用程序。如果用户选择了忽略异常，应用程序将会继续运行，但程序在这时可能已经崩溃了，再继续运行下去可能会出现无法预期的行为。如果应用程序是在处理一个数据文件，那么用户应该将整个工作保存到一个新文件中。

随后用户应该退出应用程序并重新启动。一旦重新运行，用户应该加载新的文件，并检查它是否已经损坏。如果已经损坏，那么任何新的工作都将丢失。但用户仍可回到原来的文件中继续工作。如果新文件完好无损，那么用户就可以继续编辑，并在某个时刻将原来的文件删除，或者留作备份。

我们可以通过定义一个和 System.Threading.ThreadExceptionEventHandler 委托相匹配的方法，并将其登记到 Application 类型的静态事件 ThreadException 上来覆盖这个内建的对话框。

注意 我个人认为 ThreadException 事件没能模仿 AppDomain 的 UnhandledException 事件名称和行为是比较遗憾的。换句话说，Application 的这个事件应该被称作 UnhandledException 而不是 ThreadException，并且也应该使用 UnhandledExceptionEventHandler 委托类型。

大家可能已经意识到了 Windows 窗体只处理与 CLS 兼容的异常，与 CLS 不兼容的异常会跳出线程的消息循环，并沿着调用堆栈向更高一层传递。因此，如果我们希望能够同时显示或记录与 CLS 兼容和不兼容的异常，我们必须定义两个回调方法，并将一个登记在 Application 类型的 ThreadException 事件上，将另一个登记在 AppDomain 类型的 UnhandledException 事件上。

如前所述，NativeWindow 的内部方法会捕获所有与 CLS 兼容的异常，并显示一个对话框、或者调用我们登记在 ThreadException 事件上的回调方法。但是，某些情况会改变这种默认行为。首先，如果有调试器被连接到我们的 Windows 窗体应用程序上，那么窗口过程中抛出的任何未处理异常都将不会被捕获，而会沿着调用堆栈向更高一层传递。其次，如果我们安装了 JIT 调试器，并且应用程序的 XML 配置文件(.config 文件)中指定了 jitDebugging 配置设置，那么异常也不会被捕获，同样会沿着调用堆栈向更高一层传递。

18.11.3 未处理异常与 ASP.NET Web 窗体

ASP.NET 在自己的 try 块中执行所有的 Web 窗体代码。如果我们的代码抛出了一个未处理异常，ASP.NET 将会捕获到该异常，并决定如何处理。ASP.NET 提供了几种机制允许我们在出现未处理异常时接受到一个通知。首先，我们可以定义一个回调方法使其在某个特定 Web 页面中出现未处理异常时能够被调用。该回调方法由 System.Web.UI.TemplateControl 类提供的 Error 事件来登记，其中 TemplateControl 类为 System.Web.UI.Page 和 System.Web.UI.UserControl 的基类。

除了可以接受某个特定 Web 页面中出现的未处理异常通知外，我们还可以登记一个回调方法来接受一个 Web 窗体应用程序内任何页面中出现的未处理异常通知。提供登记这种应用程序级回调方法的是 System.Web.HTTPApplication 类的 Error 事件。我们一般会将这样的代码加到 Global.asax 文件中。

ASP.NET 还提供了一种追踪选项，它允许我们将未处理异常的堆栈踪迹输出到一个 Web 页面上，从而帮助我们检测问题并修正代码。有关 ASP.NET 与异常的更多细节，可参见.NET 框架 SDK 文档。

18.11.4　未处理异常与 ASP.NET XML Web 服务

对于 ASP.NET XML Web 服务来讲，有关未处理异常的事情非常简单。当我们的 XML Web 服务方法抛出了一个未处理异常时，ASP.NET 会捕获到该异常，并抛出一个新的 System.Web.Services.Protocals.SoapException 对象。SoapException 对象然后被序列化到 XML 中，表示一个 SOAP 错误。表示 SOAP 错误的 XML 可以被任何用作 XML Web 服务客户端的机器所分析和理解。这使得 XML Web 服务中客户机/服务器之间的互操作成为可能。

如果客户端是一个安装了.NET 框架的机器，那么表示 SOAP 错误的 XML 将被反序列化为一个 SoapException 对象，并在客户端线程中抛出。客户端代码可以捕获到该异常，并以任何适当的方式来处理它们。

18.12　异常堆栈踪迹

如前所述，System.Exception 类型提供了一个公有只读属性 StackTrace。在异常筛选器或者 catch 块内，我们可以通过读取该属性来获取异常的堆栈踪迹。异常的堆栈踪迹指出了异常经过的路径中所发生的事件，它对于我们检测异常原因、进而修正错误代码来说非常有用。本节讨论和堆栈踪迹相关的一些问题，这些问题有时候并不十分明显。

Exception 类型的 StackTrace 属性非常神奇。当我们访问该属性时，我们实际上调用到了 CLR 内部的代码。该属性并不是简单地返回一个字符串。当我们构造一个新的继承自 Exception 类型的对象时，StackTrace 属性将被初始化为 null。如果我们读取该属性，我们得到的将不是堆栈踪迹，而是一个 null。

当一个异常被抛出时，CLR 会在内部记录 throw 指令出现的位置。当有捕获筛选器接受该异常时，CLR 又会记录异常被捕获的位置。

如果我们在catch块内访问被抛出异常对象的StackTrace属性，实现该属性的代码将调用到CLR内部，而后者将创建一个字符串来标识从异常被抛出开始到异常被捕获为止之间所有的方法。注意捕获筛选器不可能访问到堆栈信息，因为直到有捕获筛选器接受异常后，它才被创建。

重要 当我们抛出一个异常时，CLR会重新设置异常的起始点。也就是说，CLR只记录最近一次异常抛出的位置。下面的代码抛出了它所捕获的同一个异常对象，从而导致CLR重新设置该异常的起始点：

```
void SomeMethod() {
    try { … }
    catch (Exception e) {
        …
        throw e;   // CLR认为这里就是异常的起始点
    }
}
```

相反，如果我们重新抛出一个异常对象，CLR将不会重新设置其堆栈的起始点。下面的代码重新抛出了它所捕获的异常，但不会导致CLR重新设置异常的起始点：

```
void SomeMethod() {
    try { … }
    catch (Exception e) {
        …
        throw;      // CLR不会重新设置异常的起始点
    }
}
```

实际上，上述两段代码的惟一区别就在于CLR如何确定异常被抛出的起始点。

位于异常捕获点之上的调用堆栈中的任何方法都不会被包含在 StackTrace 属性返回的字符串中。如果我们希望得到从线程顶端到异常处理器之间所有的堆栈踪迹，我们可以将 System.Environment 的静态属性 StackTrace 与异常对象的 StackTrace 属性合并在一起来实现。(译注：实际上由于 System.Environment.StackTrace 返回的结果总会出现 StackTrace 属性调用内部的堆栈信息，所以最好的办法是用 System.Diagnostics.StackTrace 类型来实现。另外，要得到“从线程顶端到异常处理器之间所有的堆栈踪迹”，没有必要再组合异常对象的 StackTrace 属性。)

注意 FCL 还提供有一个 System.Diagnostics.StackTrace 类型，该类型中定义的一些属性和方法允许我们通过编程来操作一个堆栈踪迹以及组成堆栈踪迹的栈帧。我们可以使用几种不同的构造器来构造一个 StackTrace 对象。一些构造器构造的 StackTrace 对象表示的栈帧从线程的顶端开始、到对象被构造的位置结束。另一些构造器使用一个 Exception 对象来初始化 StackTrace 对象的栈帧。System.Environment 的静态属性 StackTrace 也和 StackTrace 类型密切相关，下面是其内部实现：首先调用 StackTrace 类型中参数为 Boolean 的那个构造器、并为其传递参数值 true；然后根据新构造的 StackTrace 对象的栈帧创建一个字符串，最后将该字符串返回给 StackTrace 属性。

如果 CLR 能够为我们的程序集找到调试符号，那么 Exception 的 StackTrace 属性或 Environment 的静态属性 StackTrace 将包括源代码文件路径名称以及代码行数，这些信息对于调试非常有用。遗憾的是，System.Diagnostics.StackTrace 类型的 ToString 方法不包括源代码文件路径名称和代码行数。希望这个 bug 能够在.NET 框架的后续版本中得到修复。

在我们获取一个堆栈踪迹时，我们可能会发现调用堆栈中的某些方法没有出现在堆栈踪迹内。造成这种情况的原因在于 JIT 编译器可能对一些方法进行了内联处理，这样做的目的是避免调用某些方法以及从中返回时产生的额外负担。许多编译器(包括 C#编译器)都提供有一个/debug 命令行开关。当该命令行开关打开时，编译器会在生成的程序集中嵌入信息告诉 JIT 编译器不要内联程序集中的任何方法，这样开发人员在调试代码时得到的堆栈踪迹将更加完整、也更有意义。

注意 JIT编译器会检查应用于程序集上的System.Diagnostics.DebuggableAttribute定制特性。我们的编译器通常会自动应用这一特性。如果DebuggableAttribute构造器的isJITOptimizer Disabled参数为true，JIT编译器将不会内联程序集中的任何方法。使用C#的/debug命令行开关会将该参数设为true。另外，通过在一个方法上应用System.Runtime.CompilerServices.MethodImpl Attribute定制特性，我们可以同时禁止JIT编译器在应用程序的调试版和发行版中对该方法进行内联处理，下面的代码演示了这一做法：

```
using System.Runtime.CompilerServices;
...
class SomeType {

    [MethodImpl(MethodImplOptions.NoInlining)]
    public void SomeMethod() {
        ...
    }
}
```

18.12.1 远程堆栈踪迹

我们知道，如果客户端向服务器发出了一个请求，但是服务器代码却抛出了一个异常，那么该异常对象将被封送处理(marshal)传回客户端，并在客户端线程中重新抛出。这种能力非常的酷，因为客户端代码可以像对待客户端抛出的异常一样来对待服务器端抛出的异常。但是，堆栈踪迹呢？

实际上，完整的堆栈踪迹也被传回到了客户端。如果我们在客户端去查看堆栈踪迹，那么得到的结果将包括从异常抛出的位置(位于服务器端)到异常被捕获的位置(位于客户端)之间所有的栈帧。这也是一种很棒的能力，因为不管问题是出在客户端还是服务器端，构建分布式应用程序的开发人员都可以很容易地检测出问题的所在，从而及时地修复它们。

下面是服务器端的堆栈踪迹能够被封送传回客户端的原理。当一个Exception对象被序列化时，其堆栈踪迹信息将被序列化为一个字符串。当该异常对象被反序列化时，新构造的Exception对象将会把该字符串保存在一个私有字段中。自此开始，当我们请求新异常对象的StackTrace属性时，其返回的字符串将不仅包括从新异常对象被捕获的位置到新异常对象被抛出的位置之间所有的堆栈信息，还要再加上先前保存的那个私有字段所维护的字符串。

换句话说，从 StackTrace 属性返回的字符串将包含从服务器端抛出异常的位置开始、到客户端捕获异常的位置结束之间所有完整的堆栈踪迹。

18.13 异常调试

微软的 Visual Studio .NET 调试器为异常提供了特殊的支持。为了获得这些支持，我们可以在 Visual Studio .NET 中选择【调试.异常】菜单项，我们将会看到如图 18.4 所示的对话框。

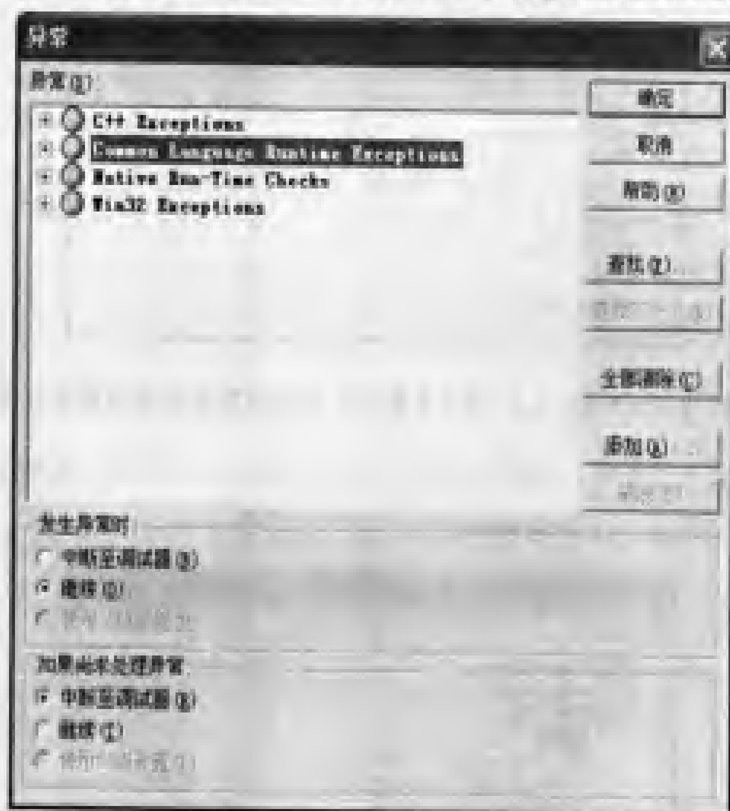


图 18.4 Visual Studio .NET【异常】对话框中显示的几种不同的异常

该对话框向我们展示了 Visual Studio 可以识别的几种不同的异常：

- **C++ Exceptions** 用于非托管 C++异常。
- **Common Language Runtime Exceptions** 用于托管异常，为本章的重点。
- **Native Run-Time Checks** 用于微软的非托管 Visual C++编译器提供的一个新的特性。这些异常只有在使用/RTC 编译器开关编译代码时才有用。
- **Win32 Exceptions** 用于 Windows 非托管 32 位异常代码。

对于 Common Language Runtime Exceptions，展开对话框中的对应分支，我们会看到 Visual Studio 调试器能够识别的命名空间集合，如图 18.5 所示。

图 18.5 Visual Studio .NET【异常】对话框根据命名空间来显示异常

如果我们展开一个命名空间，我们将会看到该命名空间中所有继承自 System.Exception 的类型。例如，图 18.6 就向我们展示了打开 System 命名空间时的情形。

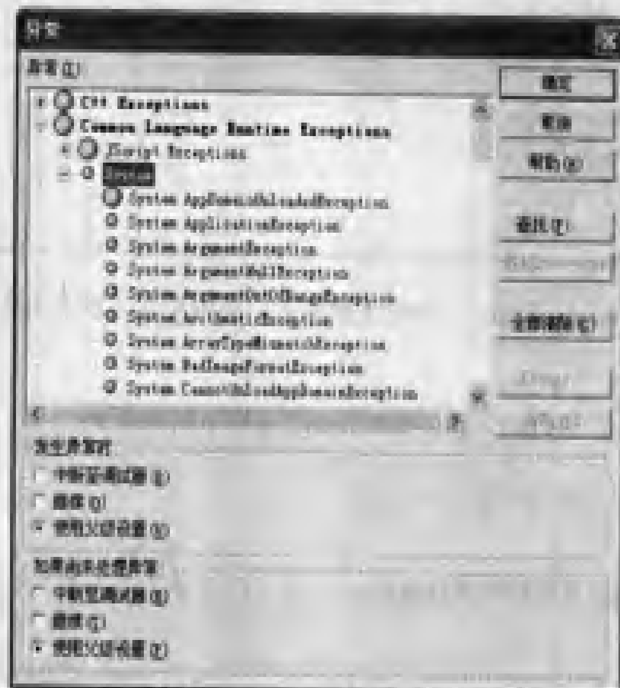


图 18.6 Visual Studio .NET【异常】对话框显示的 System 命名空间中定义的异常

在选择了一个异常类型之后，我们可以告诉调试器发生异常时它应该做何反应：

- **中断至调试器**　该选项告诉调试器只要一发生异常就通知我们。CLR 不会试图遍历调用堆栈去搜索任何捕获筛选器。该选项对于我们调试捕获异常与处理异常的代码非常有用。
- **继续**　该选项告诉调试器发生异常时不要通知我们。CLR 会遍历调用堆栈搜索能够接受所抛出异常的捕获筛选器。如果有 catch 块处理了该异常，调试器将不会通知我们该异常的出现。该选项是一个最常用的选择，因为异常被处理意味着应用程序能够预料并解决这种情况。应用程序在这种情况下会继续正常运行。如果没有异常筛选器接受所抛出的异常对象，那么将出现一个未处理异常。在这种情况下，即使我们选择了【继续】选项，调试器也会通知我们这个未处理异常。
- **使用父级设置**　该选项告诉调试器使用和父级节点相同的设置。我个人认为这种设置没有什么意义。因为我们很少会为一个命名空间中所有的异常类型都同时选择【中断至调试器】或【继续】。微软在这里应该做的是显示异常类型的层次结构，而非命名空间的层次结构。如果显示的是异常类型的层次结构，这样的设置才有意义。比如，我们可以为 ArgumentException 选择一种设置(例如【中断至调试器】)，那么其所有派生类型(ArgumentNullException、ArgumentOutOfRangeException 以及 DuplicateWaitObjectException)都将继承这种设置。这样当任何相关的异常类型被抛出时，我们都能中断至调试器。

在图 18.6 中对话框的底部，我们可以告诉调试器在遇到未处理异常时它应该做何反应：

- **中断至调试器**　该选项告诉调试器出现未处理异常时通知我们，并允许我们调试应用程序代码。对于托管应用程序，这是目前最有用的选项。实际上，这是我所选择的惟一选项。
- **继续**　对于托管应用程序，该选项告诉调试器中断应用程序。换句话说，调试器不会通知我们出现了未处理异常，应用程序会立即中断运行。在调试托管应用程序时，该选项是无用的，因为我们调试应用程序是为了修复未处理异常，而不是忽略它们。如果我们正在调试的是一段脚本代码(比如 IE 浏览器，或者其他 HTML 宿主中的脚本代码)，那么脚本代码会停止运行，宿主会收到错误报告，但宿主仍会继续运行。

- **使用父级设置**　该选项告诉调试器使用和父级节点相同的设置。同样，因为显示的是命名空间的层次结构，而非类型的层次结构，所以该选项不是很有用。

如果我们定义了自己的异常类型，我们可以将它添加到该对话框中。这可以通过选择 Common Language Runtime Exceptions 节点，并点击【添加】按钮来完成。点击按钮后将出现如图 18.7 所示的对话框。

图 18.7　让 Visual Studio .NET 识别我们自己的异常

在该对话框中，我们可以输入异常的完全限定名。注意我们输入的类型并不一定要为一个继承自 System.Exception 的类型，该对话框也支持与 CLS 不兼容的异常类型。如果我们有两个或多个位于不同程序集中的同名类型，那么将没有办法区分它们。幸运的是，这种情况很少发生。

如果我们的程序集中定义了几个异常类型，我们只能一次添加一个。我希望将来能够看到该对话框会在我们选择一个程序集后，它便能自动将其内定义的所有异常类型都导入到 Visual Studio 调试器中。另外，该对话框也应该允许我们通过程序集来区分异常类型，从而避免不同程序集中的类型同名问题。

最后，如果该对话框也能够允许我们单独地选择那些与 CLS 不兼容的异常类型就更好了。这样，我们就可以添加自己定义的与 CLS 不兼容的异常类型。然而，与 CLS 不兼容的异常类型是不鼓励使用的，所以这并非一个必须的特性。

18.13.1　告诉 Visual Studio 调试何种代码

当使用 Visual Studio 调试应用程序时，我们必须告诉调试器我们希望调试何种代码。当我们将 Visual Studio 调试器连接到一个进程上时，它会显示如图 18.8 所示的对话框。

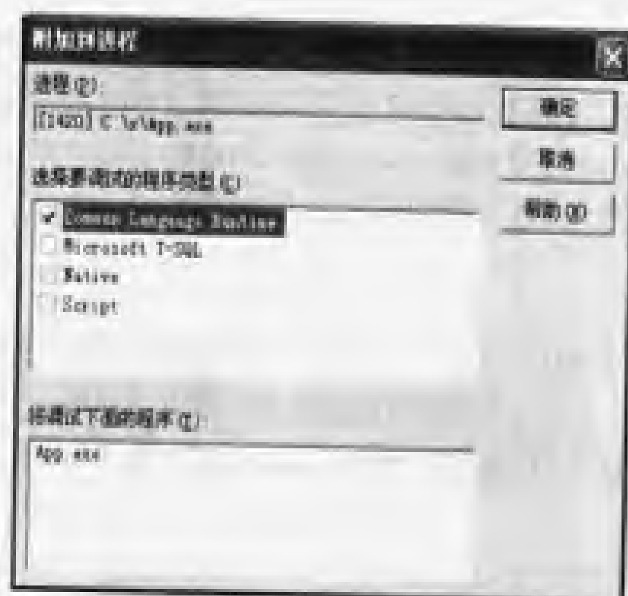


图 18.8 将 Visual Studio .NET 调试器连接到一个进程上

下面是调试各种代码的含义：

- **Common Language Runtime** 该选项允许我们调试托管代码。调试器会显示托管符号和堆栈踪迹。如果没有选择该选项，调试器将不会为托管代码显示任何符号，这将使得托管代码的单步调试变得非常麻烦。
- **Microsoft T-SQL** 该选项允许我们调试 SQL Server 数据库中的 T-SQL 存储过程。
- **Native** 该选项允许我们调试非托管代码，调试器会显示非托管符号和堆栈踪迹。如果我们的托管应用程序使用了非托管代码，但是我们又不需要调试非托管代码，那么我们应该确定不要选择该选项。如果没有选择该选项：单步调试代码的托管部分将会快许多，而且当非托管线程遇到断点时也不会挂起。如果一个线程正在执行非托管代码，那么它将继续运行。

  如果选择了该选项，只有在 Windows XP 和 Windows .NET Server 平台上才有可能将调试器和进程分开。如果没有选择该选项，则在任何 Windows 平台上都可以将调试器和进程分开。
- **Script** 该选项允许我们调试在一个宿主(如 IE 浏览器)中执行的脚本代码。

只有在我们调试托管代码和非托管代码互操作(P/Invoke 和 COM 互操作)的部分时，我们才应该同时选择 **Common Language Runtime** 和 **Native**。同时选择这两项会导致单步调试的速度非常慢。另外，不管运行的是哪个版本的 Windows，这种情况下的调试器都不会和进程分开。

我们应该认识到当单步调试发生在托管代码和非托管代码之间转换的部分时，调试器有时可能会失去控制。换句话说，当我们试图单步调试托管代码和非托管代码之间的转换部分时，线程可能会直接开始运行(也可能挂起)。大多数时候，在这些转换之间的跳转都会正常工作，但是偶尔也会失败。失败的原因在于 CLR 开发组认为性能要优先于“调试的精确度”。

对于一个 Visual Studio 项目，我们也可以通过项目属性设置来告诉 Visual Studio 我们希望调试何种代码。图 18.9 演示了一个 C#控制台应用程序项目的属性页。因为该项目为一个 C#项目，所以 Visual Studio 会假设我们总是希望调试托管代码。使用该对话框，我们也可以选择非托管调试和/或 SQL 调试。

图 18.9 为项目选择希望调试的代码种类

19

自动内存管理(垃圾收集)

本章讨论有关托管对象的一些核心问题，它们包括新对象的创建、生存期的管理、以及内存资源的回收。换言之，本章将向大家解释 Microsoft .NET 框架中垃圾收集器的工作原理，以及与之相关的各种性能问题。

19.1 垃圾收集平台基本原理解析

每个程序都要使用这样或那样的资源，比如文件、内存缓冲、屏幕空间、网络连接、数据库资源，等等。实际上，在一个面向对象的环境里，每一个类型都标识着某些为程序所用的资源。要想使用这些资源，我们必须为相应的类型实例分配一定的内存空间。下面是访问一个资源所需要的几个步骤。

1. 调用中间语言(IL)中的 newobj指令，为表示某个特定资源的类型实例分配一定的内存空间。当我们在C#或者Visual Basic以及其他一些编程语言中调用new操作符时，编译器将产生newobj指令。

2. 初始化上一步所得的内存，设置资源的初始状态，从而使其可以为程序所用。一个类型的实例构造器负责做这样的初始化工作。

3. 通过访问类型成员来使用资源，这根据需要会有一些反复。

4. 销毁资源状态，执行清理工作。这将在本章19.4 节中予以探讨。

5. 释放内存。这一步由垃圾收集器全权负责。(译注：注意这里的内存指的是分配在托管堆上的引用类型实例所占的内存资源。除了托管堆中的内存，系统运行时还有一类内存，即值类型实例所占的内存，它们位于当前运行线程的堆栈上，垃圾收集器并不负责这些内存资源的回收。当值类型实例变量所在的方法执行结束时，它们的内存将随着堆栈空间的消亡而自动消失，也就无所谓回收。)

这种看起来简单的模式却是导致许多编程错误的主要原因之一。想想看，有多少次我们忘记了释放无用的内存？又有多少次我们试图访问已经被释放的内存，恐怕谁也说不清楚。

因为这两类 bug 发生的时间和次序都难以预料，所以它们对于应用程序来说比其他大多数 bug 的危害都要大得多。对于其他大多数 bug 来讲，程序的异常行为一经发现，一般问题都能很快得到解决。但是这两类 bug 的直接后果是资源泄漏(内存消耗)和对象损毁(状态不稳定)，这使得应用程序的行为变得不可预期，发生泄漏和损毁的次数也不可预期。实际上，有许多工具(如微软的 Windows 任务管理器、系统监视器 ActiveX 控件，以及来自 Compuware 公司的 NuMega BoundsChecker 和 Rational 公司的 Purify)就是专门用来帮助开发人员查找此类 bug 的。

正确无误的资源管理通常是一件比较困难和单调的工作，它们会极大地分散开发人员解决实际问题的注意力。如果有一种机制能够简化这种容易遗漏的内存管理任务，那将是一件令人愉快的事。值得庆幸的是，确实存在这样的机制，这就是垃圾收集(garbage collection)。

垃圾收集完全将开发人员从繁杂的内存管理工作中解脱了出来，而这在以前需要开发人员追踪每一块内存的使用情况，并知晓何时释放它们。但是垃圾收集器对内存中的类型表示着何种资源却一无所知，这意味着垃圾收集器并不清楚怎样执行前面列表中的第 4 步，即“销毁资源状态，执行清理工作”。为了使资源得到正确的清理，开发人员必须自己编写执行这部分工作的代码。这些代码一般被放在 Finalize、Dispose 以及 Close 方法中，本章稍后将会谈到它们。其实在后面大家将会看到，垃圾收集器在这个问题上也能提供一些帮助，从而允许我们在许多情形下可以跳过上面第 4 步的工作。

另外，大多数类型，例如 Int32、Point、Rectangle、String、ArrayList 以及 SerializationInfo，表示的资源并不需要任何特殊的清理操作。例如，一个 Point 资源完全可以通过销毁对象内存中的 x 字段和 y 字段来完成清理工作。

另一方面，对于一个表示(或者说封装)着非托管(操作系统)资源(例如文件、数据库连接、套接字、互斥体、位图、图标，等等)的类型，在其对象被销毁时，就必须执行一些清理代码。

本章稍后将向大家解释怎样正确定义那些需要明确清理资源的类型，以及怎样正确使用这些类型。我们目前首先来探讨内存分配和资源初始化问题。

通用语言运行时(CLR)要求所有的内存资源(译注：这里仅限于分配给引用类型实例的内存资源)都从一个称作**托管堆**(managed heap)的地方分配而得。除了不需要从托管堆中释放对象外——当应用程序不再需要它们时，对象会被自动释放——托管堆和 C 语言运行时中的堆非常相似。这自然会提出一个问题，“托管堆是如何知道应用程序何时不再使用某个对象的？”这个问题稍后回答。

目前应用于实践中的垃圾收集算法有好几种。每一种算法都是针对某一特定环境而量身定做的最优化方案。本章中，我们将把注意力集中在.NET 框架中 CLR 提供的垃圾收集算法上。我们先从最基本的概念开始。

当应用程序进程完成初始化后，CLR 将保留(reserve)一块连续的地址空间，这段空间最初并不对应任何的物理内存(backing storage)。(译注：在进程的可用地址空间上为其分配内存的行为称作“保留”，进程因此分配而得的内存空间称作“保留地址空间”。由于保留地址空间是一段虚拟地址空间，所以要真正使用它，还必须为其“提交”物理内存。有关物理内存与虚拟内存、保留与提交的更详细的知识可参见 Jeffrey Richter 先生的 Programming Applications for Microsoft Windows 第 4 版。)该地址空间即为托管堆。托管堆上维护着一个指针，我们暂且称之为 NextObjPtr。该指针标识着下一个新建对象分配时在托管堆中所处的位置。刚开始的时候，NextObjPtr 被设为 CLR 保留地址空间的基地址。

中间语言(IL)指令 newobj 负责创建新的对象。如前所述，许多语言(包括 C#和 Visual Basic)都提供有一个 new 操作符，该操作符会使编译器在相应方法的 IL 代码中产生一个 newobj 指令。在代码运行时，newobj 指令将导致 CLR 执行以下几步操作：

1. 计算类型所有字段(以及其基类所有的字段)(译注：这里所说的字段应为类型的实例字段)所需要的字节总数。

2. 在前面所得字节总数的基础上再加上对象额外的附加成员所需的字节数。每个对象包括两个附加字段：一个方法表指针和一个SyncBlockIndex。在32位的系统中，这两个字段各占32位，合起来将为每个对象增加8个字节。在64位的系统中，每个字段将占用64位空间，两个合起来将为每个对象增加16个字节。

3. CLR检查保留区域中的空间是否满足分配新对象所需的字节数——如果需要则提交(commit)物理内存。如果满足，对象将被分配在NextObjPtr指针所指示的地方。接着，类型的实例构造器被调用(NextObjPtr指针会被传递给this参数)，IL指令newobj(或者说new操作符)返回为其分配的内存地址。就在newobj指令返回新对象的地址之前，NextObjPtr指针会越过新对象所处的内存区域，并指示出下一个新建对象在托管堆中的地址。

图19.1演示了一个包括3个对象的托管堆：A、B和C。如果再分配新的对象，它将被放在NextObjPtr指针所指示的位置(紧接着对象C之后)。

图 19.1 包含 3 个对象的托管堆

作为对比，我们来看一下 C 语言运行时中的堆分配内存时的情况。C 语言运行时中的堆在为对象分配内存时首先需要遍历一个链表数据结构，一旦找到了一个足够大的内存块，该内存块就会被拆分开来，同时链表相应节点上的指针也会得到适当的调整。但是对于托管堆来说，分配对象仅仅意味着在指针上增加一个数值——这显然要比操作链表的做法快许多。实际上，在托管堆中分配对象几乎可以达到和在线程堆栈中分配对象一样快的速度！另外，大多数的堆(如 C 语言运行时中的堆)都是在所找到的自由空间中为对象分配内存。因此，如果我们连续地创建了几个对象，它们将很有可能被分散在地址空间的各个角落。但是在托管堆中，连续分配的对象可以保证它们在内存中也是连续的。

在许多应用程序中，同一时间分配的对象彼此之间大多都有着较强的联系，它们经常会在同一时间被访问。例如，通常我们会在分配一个 FileStream 对象之后紧接着再分配一个 BinaryWriter 对象。然后，当应用程序使用 BinaryWriter 对象时，BinaryWriter 对象内部也会使用到 FileStream 对象。在一个垃圾收集环境中，对象在内存中的连续分配会由于引用的本地化而获得一些性能方面的提升。具体而言，这意味着应用程序的进程工作集将变得更小，方法中使用的对象也更有可能全部驻留在 CPU 缓存中。这样以来，应用程序只需使用 CPU 缓存中的数据(不会出现因缓存遗漏而导致的强制 RAM 访问)就可以执行大多数的操作，访问对象的速度自然会变得更快。(译注：进程的工作集指进程在运行时被频繁地访问的内存页面集合。如果进程的工作集比较大，进程在运行时将出现频繁的页面换入或换出现象。这种页面的换入或换出会发生在硬盘和内存之间，也会发生在内存和 CPU 缓存之间，这里主要指后一种情况。)

到目前为止，看起来好像托管堆在实现的简单性和速度方面要远优于 C 语言运行时中的堆。但是，在为此兴奋之前我们还需要清楚一个细节，即托管堆之所以能够获得这些好处是因为它实际上做了一个相当大胆的假设——那就是应用程序的地址空间和存储空间是无限的。显然，这种假设是荒谬的。托管堆必须应用某种机制来允许它做这样的假设。这种机制就是垃圾收集器。下面是其工作原理。

当应用程序调用 new 操作符创建对象时，托管堆中可能没有足够的地址空间来分配该对象。托管堆通过将对象所需要的字节总数添加到 NextObjPtr 指针表示的地址上来检测这种情况。如果得到的结果超出了托管堆的地址空间范围，那么托管堆将被认为已经充满，这时就需要执行垃圾收集。

> **重要**　上面的描述有些过于简单。实际上，垃圾收集在第 0 代对象充满时就出现了。一些垃圾收集器使用一种称作代龄(generation)的机制，该机制的惟一目的就是提高垃圾收集的性能。其基本思想是将应用程序生存期中新创建的对象认为是较新的代的一部分，而将早先创建的对象认为是较老的代的一部分。将对象划分为各个代龄使得垃圾收集器能够将执行对象限定在某个特定的代龄中，从而避免每次都对托管堆中所有的对象执行垃圾收集。本章稍后会对垃圾收集中代龄这一机制进行详细的探讨。在此之前，大家可以先简单地认为在托管堆空间充满的时候才会出现垃圾收集。

19.2　垃圾收集算法

垃圾收集器通过检查在托管堆中是否有应用程序不再使用的对象来回收内存。如果有这样的对象，它们占用的内存将可以被回收。(如果托管堆中没有可用的内存，new 操作符将会抛出一个 OutOfMemoryException 异常。)那么垃圾收集器是怎样知道应用程序是否正在使用一个对象呢？这不是一个三言两语就能说的清楚的问题。

每个应用程序都有一组根(root)。一个根是一个存储位置，其中包含着一个指向引用类型的内存指针。该指针或者指向一个托管堆中的对象，或者被设为 null。例如，所有的全局引用类型变量或静态引用类型变量都被认为是根。另外，一个线程堆栈上所有引用类型的本地变量或者参数变量也被认为是一个根。最后，在一个方法内，指向引用类型对象的 CPU 寄存器也被认为是一个根。

当 JIT 编译器编译一个方法的 IL 代码时，除了产生本地 CPU 代码外，JIT 编译器还会创建一个内部的表。从逻辑上来讲，该表中的每一个条目都标识着一个方法的本地 CPU 指令的字节偏移范围，以及该范围中一组包含根的内存地址(或者 CPU 寄存器)。表 19.1 形象地描述了一个这样的内部表。

表 19.1 一个 JIT 编译器生成的表，其中展示了本地代码偏移和方法中的根的映射关系

起始字节偏移	结尾字节偏移	根
0x00000000	0x00000020	this, arg1, arg2, ECX, EDX
0x00000021	0x00000122	this, arg2, fs, EBX
0x00000123	0x00000145	fs

如果在 0x00000021 和 0x00000122 之间的代码执行时开始了垃圾收集，那么垃圾收集器将知道参数 this、参数 arg2、本地变量 fs 以及寄存器 EBX 都是根，它们引用的托管堆中的对象将不会被认为是可收集的垃圾对象。除此之外，垃圾收集器还可以遍历线程的调用堆栈，通过检测其中每一个方法的内部表来确定所有调用方法中的根。最后，垃圾收集器使用其他一些手段来获得存储在全局引用类型变量和静态引用类型变量中保存的根。

注意 在表 19.1 中，方法的 arg1 参数在偏移为 0x00000020 处的指令执行完毕后就不再被引用了。这意味着 arg1 引用的对象在该指令执行后的任何时刻都可以被执行垃圾收集(假设应用程序中没有其他的根再引用该对象)。换句话说，只要一个对象不再可达，它就是垃圾收集的候选对象——CLR 并不保证对象在一个方法的整个生存期内都能一直存活。

但是，当应用程序运行在一个调试器中，或者当程序集包含有 System.Diagnostics.DebuggableAttribute 特性、且其构造器的参数 isJITOptimizerDisabled 被设为 true 时，JIT 编译器将会把所有变量（值类型和引用类型）的生存期扩展到它们的作用范围末端——通常为方法的结尾之处。（顺便提一句，微软的 C#编译器提供了一个/debug 命令行开关选项，它可以为程序集添加 DebuggableAttribute 特性，并将其参数 isJITOptimizerDisabled 设为 true。）该扩展会阻止垃圾收集器当代码在引用类型对象的作用范围内运行时对它们执行垃圾收集，这在做代码调试时将十分有用。想想看，如果我们调用了一个对象的方法时得到了错误的结果，但随后却看不到该对象，这无论如何都是一件让人感到奇怪的事情。

当垃圾收集器开始执行时，它首先假设托管堆中所有的对象都是可收集的垃圾。换句话说，垃圾收集器假设应用程序中没有一个根引用着托管堆中的对象。然后，垃圾收集器遍历所有的根，构造出一个包含所有可达对象的图。例如，垃圾收集器可能会定位出一个引用着托管对象的全局变量。图19.2展示了一个分配有几个对象的托管堆，其中对象A、C、D和F为应用程序的根所直接引用。所有这些对象都是可达对象图的一部分。当对象D被添加到该图中时，垃圾收集器注意到它还引用着对象H，于是对象H也被添加到该图中。垃圾收集器就这样以递归的方式来遍历应用程序中所有的可达对象。

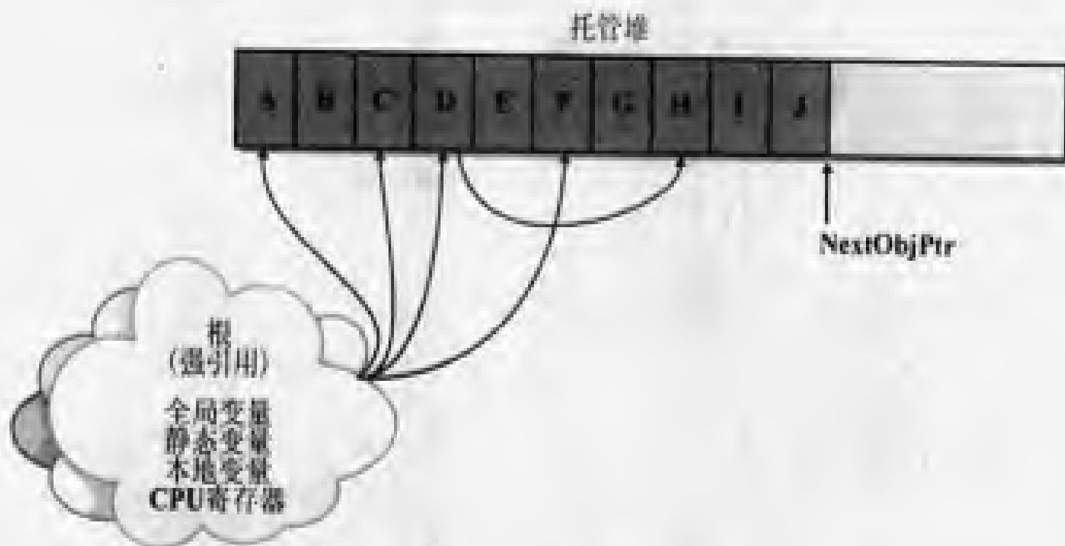


图19.2　垃圾收集执行前的托管堆

一旦该部分的可达对象图完成以后，垃圾收集器将检查下一个根，并遍历其引用的对象。当垃圾收集器在对象之间进行遍历时，如果它试图将一个先前已经添加过的对象再一次添加到可达对象图中时，它会停止沿着该对象标识的路径方向上的遍历活动。这种行为有两个目的。首先，这可以避免垃圾收集器对一些对象执行多次遍历，这在客观上提高了性能。其次，如果对象之间出现了循环引用，这可以避免遍历陷入无限循环。

垃圾收集器一旦检查完所有的根，其得到的可达对象图将包含所有从应用程序的根可以访问的对象。任何不在该图中的对象都将是应用程序不可访问的对象，因此也是可以被执行垃圾收集的对象。垃圾收集器接着线性地遍历托管堆以寻找包含可收集垃圾对象的连续区块(这些区块现在被认为是自由空间)。一些容量较小的内存块将被垃圾收集器忽略不计。

如果找到了较大的连续区块，垃圾收集器将会把内存中的一些非垃圾对象搬移到这些连续区块中(使用大家非常熟悉的 memcpy 函数)以压缩托管堆。显然，搬移内存中的对象将使所有指向这些对象的指针变得无效。所以垃圾收集器必须修改应用程序的根以使它们指向这些对象更新后的位置。另外，如果任何对象包含有指向这些对象的指针，那么垃圾收集器也会负责矫正它们。在托管堆中的内存被压缩之后，托管堆上的 NextObjPtr 指针将被设为指向最后一个非垃圾对象之后。图 19.3 演示了垃圾收集执行后的托管堆。

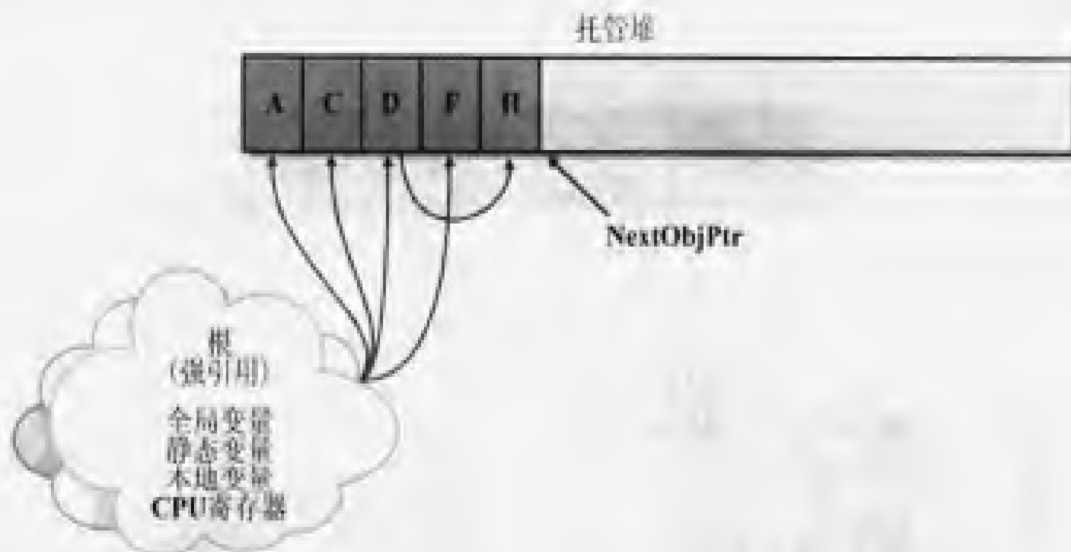


图 19.3 垃圾收集执行后的托管堆

如我们所见，垃圾收集会给应用程序带来不小的性能损伤，这也是使用托管堆时主要的负面影响。但是，我们要清楚垃圾收集只有在第 0 代对象充满时才会出现。在这之前，托管堆的性能要比 C 语言运行时中的堆的性能高许多。最后，CLR 的垃圾收集器还提供了一些特殊的优化设计可以大幅度地提高垃圾收集的性能。本章稍后将在 19.7 和 19.9 两节中讨论这些优化设计。

作为应用程序开发人员，大家应该从上面的讨论中得出以下两点重要的认识。首先，我们不必再自己实现代码来管理应用程序中对象的生存期。其次，本章开始描述的两种 bug 将不复存在。因为任何不可从应用程序的根中访问的对象都会在某个时刻被收集，所以应用程序将不可能再发生内存泄漏的情况。另外，应用程序也不可能再访问已经被释放的对象。因为如果对象可达，它将不可能被释放；而如果对象不可达，应用程序必将无法访问到它。

下面的代码演示了垃圾收集器是怎样分配和管理对象的。

```
class App {
    static void Main() {

        // 在托管堆中创建一个 ArrayList 对象，a 现在就是一个根
        ArrayList a = new ArrayList();

        // 在托管堆中创建 10,000 个对象
        for (Int32 x = 0; x < 10000; x++) {
            a.Add(new Object());    // 对象被创建在托管堆中
        }
        // 现在 a 是一个根(位于线程的堆栈上)。所以 a 是一
        // 个可达对象，a 引用的 10,000 个对象也是可达对象
        Console.WriteLine(a.Count);

        // 在 a.Count 返回后，a 便不再被 Main 中的代码所引用，
        // 因此也就不再是一个根。如果另一个线程在 a.Count
        // 的结果被传递给 WriteLine 之前启动了垃圾收集，那
        // 么这 10001 个对象的内存将被回收。
        // (译注：这 10001 个对象包括 a 引用的一个对象和其内
        // 引用的 10000 个对象。变量 x 虽然在此后的代码中也不
        // 再被引用，但由于它是一个值类型，并不存在于托管堆
        // 中，所以它不受垃圾收集器的管理，它在 Main 方法执
        // 行完毕后会随着堆栈的消失而自动被系统回收。)
        Console.WriteLine("End of method");
    }
}
```

注意　如果垃圾收集的功能是如此强大，大家可能会奇怪它为什么没有被 ANSI C++所采用。这是因为垃圾收集器必须能够识别出应用程序的根，并且还要找到所有的对象指针。非托管 C++的问题在于它允许我们将一个指针从一个类型转换为另一个类型，而我们无从知道它真正引用的对象是什么。但在 CLR 中，托管堆总能知道一个对象的实际类型，从而使用其元数据信息来判断一个对象的哪些成员引用着其他的对象。

19.3　终止化操作

到目前为止，大家应该已经对垃圾收集和托管堆的情况(包括垃圾收集器怎样回收一个对象的内存)有了一个基本的了解。大多数类型仅需要内存即可进行操作。

例如，Int32、Point、Rectangle、String以及ArrayList等这些类型只需要操作内存中的字节数据。但是，有些类型要发挥作用仅有内存是不够的。

例如，System.IO.FileStream类型就需要打开一个文件，并保存文件的句柄。然后该类型的Read和Write方法才能使用该句柄来操作文件。类似地，System.Threading.Mutex类型也需要打开一个Windows互斥体内核对象，并保存其句柄以供随后Mutex的方法被调用时使用。

任何封装了非托管资源的类型，例如文件、网络链接、套接字、互斥体等，都必须支持一种称作终止化(finalization)的操作。终止化操作允许一种资源在它所占用的内存被回收之前首先执行一些清理工作。要提供终止化操作，我们必须为类型实现一个名为Finalize的方法。当垃圾收集器判定一个对象为可收集的垃圾时，它便会调用该对象的Finalize方法(如果存在的话)。Finalize方法的实现通常便是调用CloseHandle函数(译注：CloseHandle为一个Win32函数，其主要用于关闭一个打开的对象句柄)，该函数接受非托管资源的句柄作为参数。因为FileStream类型定义了一个Finalize方法(译注：这里准确地讲应该为“重写了System.Object的Finalize方法”，因为System.Object中已经定义了一个Finalize虚方法，任何需要释放非托管资源的类型都要重写这个Finalize方法。如果一个类型以及其所有的基类型都没有重写Object的Finalize方法，那么垃圾收集器会认为该类型不需要终止化操作，它不会调用Object的Finalize方法。本章后面会多次提到“某某类型定义了一个Finalize方法”，如未做特殊说明，大家应该将其理解为“某某类型或者是其某个基类型重写了System.Object的Finalize方法”)，所以每一个FileStream对象都会保证在其内存被回收时能够释放它所占用的非托管资源。如果一个封装了非托管资源的类型没有定义Finalize方法，那么这些非托管资源将得不到关闭，从而会导致某种程度的资源泄漏(译注：这里的前提是我们没有显式关闭对象所封装的非托管资源)。直到进程结束，这些非托管资源才会被操作系统回收。

下面的OSHandle类型演示了如何定义一个封装着非托管资源的类型。当垃圾收集器判定一个OSHandle对象是可收集的垃圾时，它会调用其上的Finalize方法，Finalize方法在内部调用了Win32函数CloseHandle，从而确保非托管资源得到了释放。OSHandle类可以用于任何使用CloseHandle函数释放的非托管资源。如果大家处理的非托管资源需要其他不同的清理函数，则可以在下面Finalize方法实现的基础上做相应的改动。

```
public sealed class OSHandle {

   // 下面的字段保存着一个非托管资源的Win32句柄
   private IntPtr handle;

   // 下面的构造器用于初始化handle句柄
   public OSHandle(IntPtr handle) {
      this.handle = handle;
   }
```

```
    // 当垃圾收集执行时，下面的 Finalize 方法将被调用，
    // 它将关闭非托管资源句柄
    protected override void Finalize() {
        try {
            CloseHandle(handle);
        }
        finally {
            base.Finalize();
        }
    }

    // 下面的公有方法用于返回所封装的 handle 句柄
    public IntPtr ToHandle() { return handle; }

    // 下面的公有隐式转型操作符也用于返回所封装的 handle 句柄
    public static implicit operator IntPtr(OSHandle osHandle) {
        return osHandle.ToHandle();
    }

    // 下面的私有方法用于释放非托管资源
    [System.Runtime.InteropServices.DllImport("Kernel32")]
    private extern static Boolean CloseHandle(IntPtr handle);
}
```

当 OSHandle 对象被执行垃圾收集时，垃圾收集器会调用它的 Finalize 方法。该方法首先执行必要的清理操作，然后调用基类型的 Finalize 方法以使基类型也能够有机会释放其内的非托管资源。对基类型的 Finalize 方法的调用被放在一个 finally 块中，这可以确保即使在 OSHandle 的清理代码抛出异常的情况下它也能够得到调用。在上面的例子中，System.Object 的 Finalize 方法将被调用。由于 Object 的 Finalize 方法并没有任何实现代码，所以我们可以去掉上面代码中的异常处理和对 base.Finalize 的调用，这可以在不失正确性的前提下提高代码的性能。

然而，C#编译器实际上并不会编译上面的代码。C#编译器组发现许多开发人员对 Finalize 方法的实现都不够正确。具体而言，许多开发人员都会忘记在 Finalize 方法中对异常情况进行处理、以及调用基类型的 Finalize 方法。为了减轻开发人员的负担，C#为定义 Finalize 方法提供了特殊的语法。下面的 C#代码所做的工作实际上和上面的代码是等同的，只是这段代码会成功通过编译，因为它使用了 C#提供的特殊语法来定义 Finalize 方法。

```
public sealed class OSHandle {

    // 下面的字段保存着一个非托管资源的 Win32 句柄
    private IntPtr handle;

    // 下面的构造器用于初始化 handle 句柄
    public OSHandle(IntPtr handle) {
        this.handle = handle;
    }

    // 当垃圾收集执行时，下面的析构器(Finalize)方法将被
    // 调用，它将关闭非托管资源句柄
    ~OSHandle() {
        CloseHandle(handle);
    }

    // 下面的公有方法用于返回所封装的 handle 句柄
    public IntPtr ToHandle() { return handle; }

    // 下面的公有隐式转型操作符也用于返回所封装的 handle 句柄
    public static implicit operator IntPtr(OSHandle osHandle) {
        return osHandle.ToHandle();
    }

    // 下面的私有方法用于释放非托管资源
    [System.Runtime.InteropServices.DllImport("Kernel32")]
    private extern static Boolean CloseHandle(IntPtr handle);
}
```

编译上面的代码，并用 ILDasm.exe 查看生成的程序集，我们将会看到 C#编译器实际上在托管模块的元数据中产生了一个名为 Finalize 的方法。如果我们再查看 Finalize 方法的 IL 代码，我们会看到 C#编译器将 CloseHandle 函数调用放在了一个 try 块内，同时又在和该 try 块相匹配的 finally 块中添加了一个 base.Finalize 调用。

要创建一个 OSHandle 的对象实例，我们首先必须调用一个 Win32 函数来返回一个非托管资源的句柄，这样的函数有 CreateFile、CreateMutex、CreateSemaphore、CreateEvent、socket、CreateFileMapping，等等。然后，我们便可以使用 C#的 new 操作符来构造一个 OSHandle 的实例(将上面得到的非托管资源句柄作为构造器参数)。

这样当未来某个时刻垃圾收集器判定该对象为可收集的垃圾时，它会看到该类型定义有一个 Finalize 方法，于是它便会调用该方法，从而允许 CloseHandle 函数来关闭其中的非托管资源。在 Finalize 方法返回之后的某个时刻，该 OSHandle 对象在托管堆中所占的内存才会被回收。

重要 如果大家熟悉C++，大家会注意到C#定义Finalize方法的语法非常类似于C++中定义析构器的语法。实际上C#语言规范中甚至就将该方法称为析构器。但是，Finalize方法的工作原理和非托管C++中析构器的工作原理完全不同。

我个人认为C#编译器组选择模仿C++析构器的语法，并将该方法称为析构器是一个错误的决定。因为这会搞乱那些从C++背景转向C#和.NET框架的开发人员。这些开发人员可能会错误地认为使用C#的析构器语法意味着类型实例就可以被确定性地析构（就像C++中的一样）。但是CLR不支持确定性的析构，因此C#也不能提供这种机制。实际上，没有哪个支持CLR的编程语言可以提供这样的能力。即使我们使用托管扩展C++来实现一个托管类型，定义一个"析构器"也会导致编译器产生一个只有在垃圾收集执行时才能被调用的Finalize方法。

不要被一个语言的析构器语法所迷惑——Finalize方法只有在垃圾收集执行时才会被调用。它不会在一个方法退出、或者对象超出作用范围时被调用。

在设计一个类型时，我们应该尽可能地避免使用Finalize方法，原因如下：

- 终止化对象(译注：终止化对象是指那些类型本身，或者其基类型重写了System.Object的Finalize方法的对象。反之，非终止化对象是指那些类型本身及其所有的基类型都没有重写System.Object的Finalize方法的对象)的代龄会被提升，这会增加内存的压力，并会在垃圾收集器判定对象为可收集的垃圾时阻止回收对象的内存。另外，所有被终止化对象直接引用或者间接引用的对象的代龄也将被提升。(本章稍后将讨论对象的代龄和代龄的提升等相关问题。)
- 终止化对象的分配花费的时间较长，因为指向它们的指针必须被放在终止化链表上(这将在19.3.2节中阐述)。
- 强制垃圾收集器执行Finalize方法会极大地损伤应用程序的性能。记住，每个对象都会被终止化。所以，如果我们有一个含有10 000个对象的数组，那么每个对象的Finalize方法都必须被调用。

- 一个终止化对象可能会引用其他的对象(包括终止化对象和非终止化对象)，从而不必要地延长它们的生存期。实际上，我们可以考虑将一个类型拆分成两个不同的类型，一个是包含Finalize方法的轻量级类型，其中不要引用任何其他的对象(就像前面展示的OSHandle类型一样)。另一个是不包含Finalize方法的类型，其中可以引用任何其他对象。

- 我们并不能控制Finalize方法何时执行。对象可能会一直占有着非托管资源，直到出现垃圾收集。

- CLR不对Finalize方法的执行顺序做任何保证。例如，假设一个对象中包含着一个指向另一个对象(下面称为内部对象)的指针。现在，垃圾收集器检测到两个对象都是可被收集的垃圾，假设内部对象的Finalize方法首先被调用。让我们再假设外部对象的Finalize方法要访问内部对象，并调用其上的方法。但是内部对象已经执行了终止化操作，因此结果将变得不可预期。出于这种原因，微软强烈建议大家不要在Finalize方法中访问任何内部成员对象。本章稍后谈论的Dispose模式会为我们提供另一种清理资源的方式，它就不会受到这里谈到的约束。

如果我们决定为自己的类型实现Finalize方法，我们要确保其中的代码能够尽可能快地执行，而避免那些有可能阻塞Finalize方法的行为，包括任何线程同步操作。另外，如果有任何异常在Finalize方法中未经捕获而逃脱，那么CLR会忽略它，并继续调用其他对象的Finalize方法。

注意 到目前为止，实现一个Finalize方法最常见的原因便是释放对象所占有的非托管资源。实际上，在终止化操作中，我们应该避免编写代码访问其他的托管对象或者托管静态方法。避免访问其他托管对象的原因是这些对象的类型也可能实现了Finalize方法，而它们有可能首先被调用，从而将这些对象置于一个不可预期的状态。避免调用托管静态方法的原因是这些方法内部可能会访问已经执行了终止化操作的对象，从而也会把这些方法置于一种不可预期的状态。

Finalize 方法也可以用于其他目的。下面演示的类会在每次执行垃圾收集时使计算机发出“嘟嘟”的响声。

```
public sealed class GCBeep {
    ~GCBeep() {
        // 进入终止化操作，使计算机发出响声
        MessageBeep(-1);

        // 如果应用程序域(AppDomain)不是正在执行卸载，
        // 那么创建一个新的对象以备下一次垃圾收集时执行
        // 终止化，本章下一节将讨论 IsFinalizingForUnload
        if (!AppDomain.CurrentDomain.IsFinalizingForUnload())
            new GCBeep();
    }

    [System.Runtime.InteropServices.DllImport("User32.dll")]
    private extern static Boolean MessageBeep(Int32 uType);
}
```

要使用上面的类，我们只需要构造一个 GCBeep 类的实例。然后不管何时出现垃圾收集，GCBeep 对象的 Finalize 方法都会被调用，Finalize 方法内部在调用 MessageBeep 之后会构造一个新的 GCBeep 对象。这样当下一次垃圾收集出现时，新构造的 GCBeep 对象的 Finalize 方法又会被调用。下面的例子演示了 GCBeep 类的使用方法。

```
class App {
    static void Main() {
        // 构造一个 GCBeep 对象以使每次执行
        // 垃圾收集时计算机都能发出一次响声
        new GCBeep();

        // 构造多个含有 100 个字节的对象
        for (Int32 x = 0; x < 10000; x++) {
            Console.WriteLine(x);
            Byte[] b = new Byte[100];
        }
    }
}
```

另外需要注意的是即使一个类型的实例构造器抛出了异常，它的 Finalize 方法也会被调用。所以我们实现的 Finalize 方法不应想当然地认为对象会处在一个良好的、一致的状态。下面的代码演示了这一点。

```
class TempFile {
    String filename = null;
    public FileStream fs;

    public TempFile(String filename) {
        // 下面一行代码可能会抛出异常
        fs = new FileStream(filename, FileMode.Create);

        // 保存文件名称
        this.filename = filename;
    }

    ~TempFile() {
        // 因为我们并不能确信 filename 在构造器中是否完成了初始
        // 化，所以这里正确的做法是首先测试 filename 是否为 null
        if (filename != null)
            File.Delete(filename);
    }
}
```

作为选择，我们也可以用如下的代码来实现同样的功能。

```
class TempFile {
    String filename;
    public FileStream fs;

    public TempFile(String filename) {
        try {
            // 下面一行代码可能会抛出异常
            fs = new FileStream(filename, FileMode.Create);

            // 保存文件名称
            this.filename = filename;
        }
        catch {
            // 如果出现了问题，告诉垃圾收集器不要调用 Finalize
            // 方法。本章稍后会讨论 SuppressFinalize 方法
            GC.SuppressFinalize(this);

            // 告诉调用者发生了异常
            throw;
        }
    }

    ~TempFile() {
        // 没有必要再用 if 语句，因为这段代码只会在构造器运行
        // 成功的情况下才会执行
        File.Delete(filename);
    }
}
```

19.3.1 调用 Finalize 方法的条件

有4种事件会导致一个对象的 Finalize 方法被调用：

- **第0代对象充满** 该事件是目前导致 Finalize 方法被调用的最常见的一种方式。该事件通常在应用程序代码运行过程中分配新对象的时候发生。
- **显式调用 System.GC 的静态方法 Collect** 我们的代码可以显式地请求 CLR 执行垃圾收集。虽然微软强烈建议不要这样做，但某些时候强制执行垃圾收集对应用程序来说还是有意义的。(本章稍后将探讨这一点。)
- **CLR 卸载应用程序域** 当一个应用程序域(AppDomain)卸载时，CLR 会认为该应用程序域中不存在任何根，因此会调用该应用程序域中创建的所有对象(译注：准确地讲应该是"所有的终止化对象")上的 Finalize 方法。本书第20章将讨论应用程序域。
- **CLR 被关闭** 当一个进程正常中断时，它会试图关闭 CLR。这时 CLR 会认为该进程中不存在任何根，因此会调用托管堆中所有对象(译注：同样应该是"所有的终止化对象")上的 Finalize 方法。

CLR 使用一个特殊的专用线程来调用 Finalize 方法。对于上面列出的第1种、第2种和第3种事件来说，如果有 Finalize 方法进入了一个无限循环，那么这个特殊的线程将被阻塞，其他的 Finalize 方法将得不到调用。这种情况非常糟糕，因为应用程序将不能够再回收其他终止化对象占用的内存——只要应用程序还在运行，它就存在泄漏内存的可能。

对于第4种事件来说，每个 Finalize 方法会有大约2秒钟的运行时间。如果一个 Finalize 方法没有在2秒钟内返回，那么 CLR 将中断该进程——其他的 Finalize 方法将得不到调用。另外，如果调用所有对象的 Finalize 方法超过了40秒钟，那么 CLR 也会中断该进程。

> **注意** 这些为超时所设定的值在本书写作的时候是正确的，但是微软可能会在将来改变它们。

Finalize 方法中的代码也可能会构造新的对象。如果这发生在 CLR 关闭期间，CLR 会继续收集对象，并调用它们的 Finalize 方法，直到不再存在对象、或者它们的执行时间超过40秒钟为止。

回顾本章前面的GCBeep类型。如果一个GCBeep对象由于第1种或第2种事件被执行了终止化，那么将有一个新的GCBeep对象被构造。因为应用程序会继续运行，后面还会有更多的垃圾收集出现，所以这种做法没什么问题。但是，如果一个GCBeep对象由于第3种或第4种事件被执行了终止化，那么将不会有新的GCBeep对象被构造，因为这时应用程序域正在执行卸载、或者CLR正在关闭。如果在这时构造了这些新的对象，那么CLR将面临许多无用的工作，因为它还要调用它们的Finalize方法。

为了阻止构造新的GCBeep对象，GCBeep的Finalize方法中调用了AppDomain的IsFinalizingForUnload方法。如果GCBeep对象的Finalize方法是由于应用程序域卸载而被调用的，那么该方法将返回true。这种解决方案对于应用程序域卸载是可行的，但如果是CLR正在关闭呢？

为了获知CLR是否正在关闭，微软为System.Environment类添加了一个HasShutdownStarted只读属性。GCBeep的Finalize方法应该读取该属性，如果其返回的值为true，那么就不应该再构造新的对象。不幸的是，GCBeep的Finalize方法并没有这样做，为什么呢？这是因为该属性根本无法访问！微软的开发人员在这里犯了一个错误，他们把HasShutdownStarted属性实现成了一个实例属性，但是Environment类却只有一个私有构造器，因此我们没有办法创建它的实例，从而也就没有办法访问该属性——这实在是太糟糕了！希望微软能够在.NET框架类库(FCL)的后续版本中修复这一bug。

目前我们还没有办法解决这一问题。GCBeep的Finalize方法仍然会在CLR关闭的时候构造新的对象。新构造的对象继续被执行终止化，再构造新的对象……直到过了40秒钟后，CLR才会放弃这样的操作并中断进程。

19.3.2 终止化操作的内部机理

从表面上来看，终止化操作好像很简单。我们创建一个对象，当该对象被执行垃圾收集时，垃圾收集器会自动调用它的Finalize方法。但是一旦深入研究下去，大家会发现终止化操作远非这么简单。

当应用程序创建一个新对象时，new操作符会为对象从托管堆上分配内存。如果该对象的类型定义了Finalize方法，那么在该类型的实例构造器被调用之前，指向该对象的一个指针将被放到一个称作终止化链表(finalization list)的数据结构里面。终止化链表是一个由垃圾收集器控制的内部数据结构。链表上的每一个条目都引用着一个对象，这实际上是在告诉垃圾收集器在回收这些对象的内存之前要首先调用它们的Finalize方法。

图 19.4 展示了一个托管堆，其中包含着几个对象。它们中有些是从应用程序的根可达的对象，有些则不是。当对象 C、E、F、I 和 J 被创建时，系统会检测到这些对象的类型定义了 Finalize 方法，于是便会将指向这些对象的指针添加到终止化链表中。

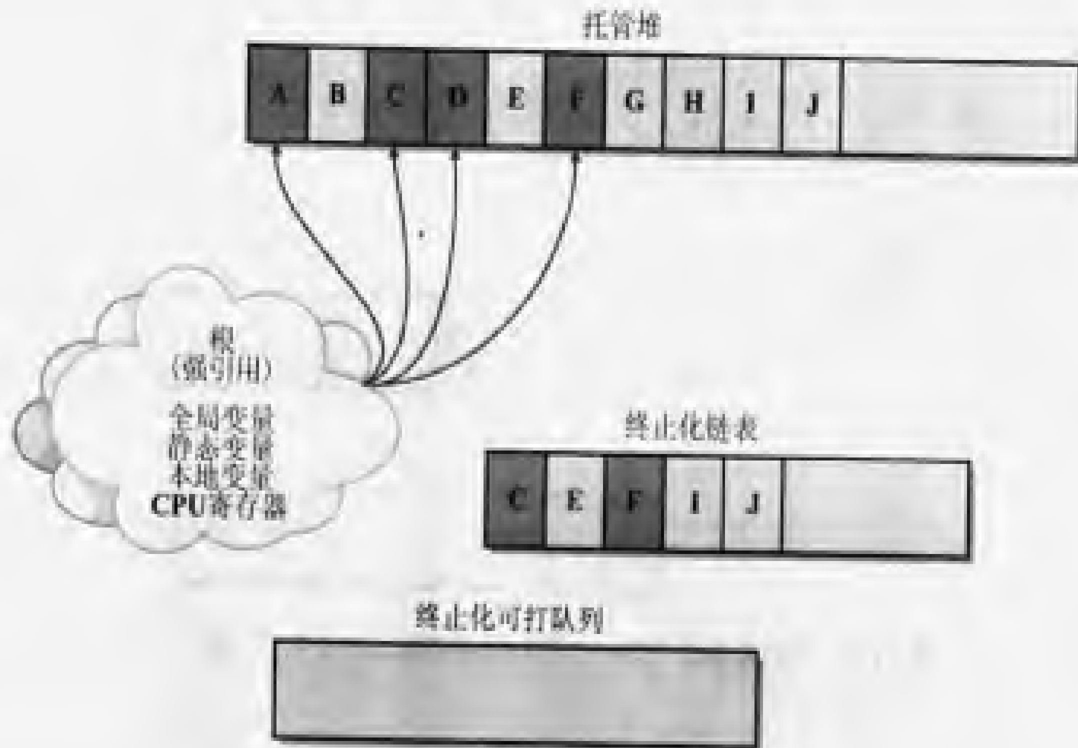


图 19.4　终止化链表上含有一些指针

> **注意**　虽然 System.Object 定义了一个 Finalize 方法，但是 CLR 会忽略它。也就是说，当构造一个类型的实例时，如果该类型的 Finalize 方法是从 System.Object 继承的，那么该对象将不被认为是终止化对象。一个对象要成为终止化对象，那么在它的类型及其基类型中（除 Object 之外），必须至少有一个重写 Object 的 Finalize 方法。

当垃圾收集开始时，对象 B、E、G、H、I 和 J 将被判定为可收集的垃圾。垃圾收集器然后扫描终止化链表以查找其中是否有指向这些对象的指针。当找到这样的指针时，它们会被从终止化链表上移除，并添加到一个称作终止化可达队列(freachable queue，其英文名称中的 freachable 可念作“F-reachable”)的数据结构上。终止化可达队列是另一个由垃圾收集器控制的内部数据结构。在终止化可达队列中出现的对象表示该对象的 Finalize 方法即将被调用。当垃圾收集执行完毕后，托管堆的情况将如图 19.5 所示。

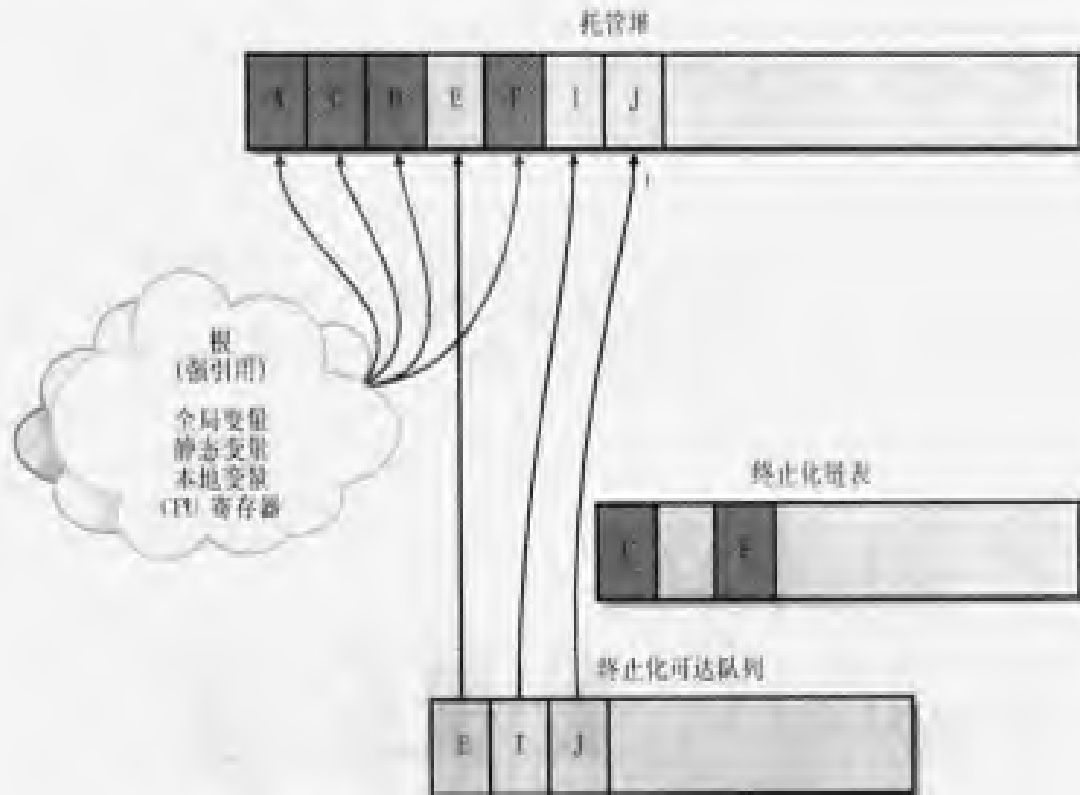


图 19.5 终止化链表中的一些指针已经转移到终止化可达队列中

在上面的图中，我们看到对象 B、G、和 H 占用的内存已经被回收了，因为它们没有 Finalize 方法需要调用。但是对象 E、I 和 J 占用的内存却不能被回收，因为它们的 Finalize 方法还没有被调用。

CLR 中有一个特殊的高优先级的线程专门用于调用 Finalize 方法。使用专用的线程可以避免潜在的线程同步问题。如果使用的是应用程序的一个线程，线程同步问题就有可能造成一些麻烦。当终止化可达队列为空时(通常的情况)，该线程将处于睡眠状态。但当终止化可达队列中有条目出现时，该线程将被唤醒，开始把每个条目从终止化可达队列中移除，并调用每个对象的 Finalize 方法。因为该线程特殊的工作方式，我们在 Finalize 方法中的代码不应该对正在执行的线程做任何假设。例如，我们应该避免在 Finalize 方法中访问线程本地存储数据(Thread Local Storage，简称 TLS)。

终止化链表和终止化可达队列之间的交互非常有意思。首先，我们来了解一下终止化可达队列这一名称的由来。其中“freachable”的“f”显然代表“终止化”(finalization)，其含义为终止化可达队列中每一个条目的 Finalize 方法都应该被调用。而“reachable”部分又意味着对象是可达的。换一种方式来看，我们可以把终止化可达队列视作和全局变量、静态变量一样的根。所以如果一个对象位于终止化可达队列上，那么该对象将是一个可达对象，因此也就不是一个可被收集的垃圾对象。

简而言之，如果一个对象是不可达的，那么垃圾收集器将把它视作可收集的垃圾。当垃圾收集器将一个对象从终止化链表转移到终止化可达队列上时，该对象将不再被认为是可收集的垃圾对象，它的内存也就不可能被回收。到此为止，垃圾收集器已经完成了垃圾对象的鉴别工作。一些原先被认为是垃圾的对象现在又被认为不是垃圾，从某种意义上讲，对象又“复苏”了。当第一次垃圾收集执行完毕后，特殊的 CLR 线程将会清空终止化可达队列中的对象，同时执行其中每个对象的 Finalize 方法。

等下一次垃圾收集执行时，它会看到终止化对象已经成为真正的垃圾对象，因为应用程序的根不再指向它，终止化可达队列也不再指向它。这时对象的内存才会被回收。在整个过程中，我们需要认识到的最重要的一点就是终止化对象需要执行两次垃圾收集才能释放掉它所占用的内存。实际上，由于对象的代龄可能会被提升，所以释放一个对象占用的内存所需要的垃圾收集的次数可能会超过两次，本章稍后将谈论这一问题。图 19.6 展示了第 2 次垃圾收集执行后托管堆的情况。

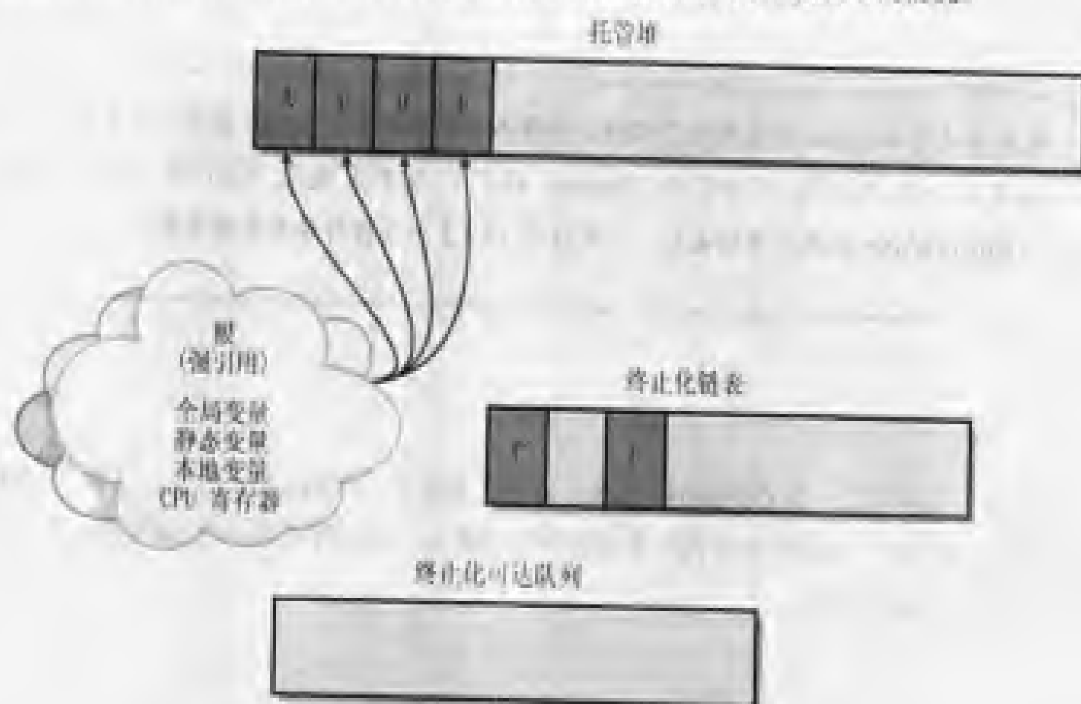


图 19.6 第 2 次垃圾收集执行后托管堆的情况

19.4 Dispose 模式：强制对象清理资源

Finalize 方法非常有用，因为它可以确保托管对象在释放内存的同时不会泄漏非托管资源。但是 Finalize 方法的问题在于我们并不能确定它会在何时被调用，而且由于它并不是一个公有方法，我们也不能显式地调用它。

当我们使用互斥体和位图这样的非托管资源时，能够显式地释放或者关闭对象通常是很有用的。使用文件和数据库连接更是如此。例如，我们可能希望打开一个数据库连接，查询一些记录，然后再关闭该数据库连接——我们并不希望让该数据库连接一直打开直到下一次垃圾收集出现，因为下一次垃圾收集完全有可能在我们获取数据库记录后的几个小时、甚至几天后才出现。

要提供显式释放或者关闭对象的能力，一个类型通常要实现一种被称为 Dispose 的模式。Dispose 模式定义了开发人员在实现类型的显式资源清理功能时所要遵循的一些约定。如果一个类型实现了 Dispose 模式，使用该类型的开发人员将能够知道当对象不再被使用时如何显式地释放掉它所占用的资源。

> **注意** 所有定义了 Finalize 方法的类型都应该实现本节所描述的 Dispose 模式以给用户更多的控制权。但是，一个类型也可以实现 Dispose 模式，而不用定义 Finalize 方法。例如，System.IO.BinaryWriter 就是这样的类型。本章后面 19.4.3 一节将解释其中的原因。

前面曾向大家演示了一个 OSHandle 类型，其中就实现了一个 Finalize 方法。这样当 OSHandle 对象被执行垃圾收集时，其封装的非托管资源将自动被关闭。但使用 OSHandle 对象的开发人员却无法显式关闭其中的非托管资源。

下面的代码演示了一个新版的 OSHandle 实现，其中就应用了 Dispose 模式。

```
using System;

// 实现 IDisposable 接口以表明该类提供了 Dispose 模式
public sealed class OSHandle : IDisposable {

   // 下面的字段保存着一个非托管资源的 Win32 句柄
   private IntPtr handle;
```

```
// 下面的构造器用于初始化 handle 句柄
public OSHandle(IntPtr handle) {
   this.handle = handle;
}

// 当垃圾收集执行时，下面的 Finalize 方法将被调用，
// 它将关闭非托管资源句柄
~OSHandle() {
   Dispose(false);
}

// 下面的公有方法可以被显式调用来关闭非托管
// 资源句柄
public void Dispose() {
   // 因为对象的资源被显式清理，所以在这里阻止
   // 垃圾收集器调用 Finalize 方法
   GC.SuppressFinalize(this);

   // 进行实际的资源清理工作
   Dispose(true);
}

// 下面的公有方法可以用来替换 Dispose 方法
public void Close() {
   Dispose();
}

// 下面的方法用于进行实际的清理工作。Finalize、Dispose
// 和 Close 都要调用该方法。因为该类是一个密封类，所以
// 该方法被定义为私有方法。如果该类不是一个密封类，
// 那么该方法应该是一个受保护方法
private void Dispose(Boolean disposing) {
   // 同步那些同时调用 Dispose/Close 方法的线程
   lock (this) {
      if (disposing) {
         // 对象正在被显式释放/关闭，而非执
         // 行终止化。因此在该 if 语句中访问
         // 那些引用其他对象的字段是安全的，
         // 因为这些对象的 Finalize 方法还没有
         // 被调用

         // 对于 OSHandle 来说这里没什么可做的
      }
      // 对象正在被释放/关闭、或者被执行终止化
      if (IsValid) {
         // 如果 handle 有效，那么关闭非托管资源
```

```
                // 注意：将 CloseHandle 替换成大家自己
                // 关闭非托管资源的函数
                CloseHandle(handle);

                // 将 handle 字段设置为某个标记值用来防
                // 止多次调用 CloseHandle
                handle = InvalidHandle;
            }
        }
    }

    // 下面的公有属性用于返回一个无效的句柄值。注意：使
    // 该属性返回一个表示正在使用的非托管资源无效的值
    public IntPtr InvalidHandle { get { return IntPtr.Zero; } }

    // 下面的公有方法用于返回所封装的 handle 句柄
    public IntPtr ToHandle() { return handle; }

    // 下面的公有隐式转型操作符也用于返回所封装的 handle 句柄
    public static implicit operator IntPtr(OSHandle osHandle) {
        return osHandle.ToHandle();
    }

    // 下面的公有属性用于表示所封装的句柄是否有效
    public Boolean IsValid { get { return (handle != InvalidHandle); } }
    public Boolean IsInvalid { get { return !IsValid; } }

    // 下面的私有方法用于释放非托管资源
    [System.Runtime.InteropServices.DllImport("Kernel32")]
    private extern static Boolean CloseHandle(IntPtr handle);
}
```

上面的 OSHandle 实现看起来有些琐碎，但实际却不然，其中有许多代码是用来支持 Dispose 模式的。虽然该类只封装了一个简单的非托管资源，但是它却几乎可以用于任何场合。如果大家需要设计一个封装有非托管资源的类型，完全可以将上面的代码拷贝到自己的项目中并做适当的修改即可使用。实际上，如果微软能够将上面的 OSHandle 类引入到 FCL 中就更好了。

下面对新版的 OSHandle 实现代码做一解释。首先，OSHandle 类实现了 System.IDisposable 接口。该接口在 FCL 中定义如下：

```
public interface IDisposable {
    void Dispose();
}
```

任何实现了该接口的类型都是在声明自己遵循了 Dispose 模式。简而言之，这意味着该类型提供了一个公有、无参的 Dispose 方法。我们可以通过显式调用 Dispose 方法来释放对象所封装的非托管资源。注意调用该方法并不会释放对象在托管堆中占用的内存，释放对象内存的工作仍由垃圾收集器负责，而且释放的时间仍不确定。

> **注意**　大家可能注意到了 OSHandle 类型还提供了一个公有的 Close 方法。该方法只是调用了一下 Dispose 方法。一些遵循 Dispose 模式的类型为方便起见大都提供了 Close 方法，但这对于 Dispose 模式来说并非必须。例如，System.IO.FileStream 类在遵循 Dispose 模式的同时也提供了一个 Close 方法，这是因为很多开发人员都认为"关闭(close)"一个文件要比"释放(dispose)"一个文件讲起来更自然一些。但是，System.Threading.Timer 类就没有提供 Close 方法，虽然它也实现了 Dispose 模式。

无参的 Dispose 方法和 Close 方法都应该是公有的非虚方法。但是，因为 Dispose 方法是在实现一个接口的方法，所以它默认情况下是一个虚方法。所以最好的做法是将 Dispose 方法定义为密封方法。幸运的是，当我们在 C#中实现一个接口方法时，编译器将默认该虚方法为公有密封方法。但是如果大家使用的是另外不同的编程语言，则应该确保 Dispose 方法被定义为密封方法。

现在我们有了 3 种方法可以用来清理 OSHandle 对象封装的非托管资源：第 1 种是显式调用 Dispose 方法，第 2 种是显式调用 Close 方法，第 3 种是让垃圾收集器调用 Finalize 方法。不管选用哪一种方法，清理代码都是一样的，所以我们将其放在一个单独的、私有非虚 Dispose 方法中，该方法接受一个 Boolean 类型的参数 disposing。

带 Boolean 参数的 Dispose 方法中包含着实际进行非托管资源清理工作的代码。在 OSHandle 例子中，该方法只是简单地调用了一下 CloseHandle 函数，并将 handle 字段设置为一个无效的句柄。将 handle 字段设置为一个无效的句柄可以确保它所表示的句柄不至于被多次关闭，因为无参的 Dispose 方法和 Close 方法都是公有方法，应用程序有可能多次调用它们。另外，由于多个线程可能会同时调用这些方法，所以带 Boolean 参数的 Dispose 方法中使用了 C#的 lock 语句来确保该方法中的代码是线程安全的。

当一个 OSHandle 对象的 Finalize 方法被调用时，Dispose 方法的 disposing 参数将被设为 false。这将告诉 Dispose 方法它不应该执行任何引用其他托管对象(译注：这里的托管对象准确地讲应该为"终止化对象")的代码。

假设 CLR 正在关闭，而在一个 Finalize 方法中我们却试图往一个 FileStream 中写入数据。由于 FileStream 的 Finalize 方法可能已经被调用了，所以这种操作就有可能失败。

另一方面，当调用无参的 Dispose 方法和 Close 方法时，disposing 参数将被设为 true。这表示对象正在被显式执行资源清理，而非由垃圾收集器执行终止化。在这种情况下，Dispose 方法就可以执行引用其他对象(例如 FileStream)的代码。因为这时应用程序的逻辑是由我们控制，所以我们知道 FileStream 对象是否仍然还打开。

顺便提一句，如果 OSHandle 类型没有标识 sealed，那么带 Boolean 参数的 Dispose 方法应该被实现为一个受保护的虚方法，而不是一个私有的非虚方法。任何继承自 OSHandle 的类都可以重写该方法，但是不要去重写无参的 Dispose 方法和 Close 方法，也不要去重写 Finalize 方法，这 3 个方法应该直接被继承到派生类型中。

重要 我们需要清楚使用 Dispose 模式时可能出现的一些版本问题。如果第 1 个版本中的基类型没有实现 IDisposable 接口，那么它将不可以在后续的版本中再实现该接口。如果该基类型在后续的版本中添加了 IDisposable 接口，那么所有的派生类型将不会知道去调用基类型中的 Dispose 方法，因此基类型中封装的非托管资源将不能得到正确的清理。类似地，如果第 1 个版本中的基类型实现了 IDisposable 接口，那么它将不可以在后续的版本中再删除该接口，因为派生类型会试图去调用一个在基类型中不存在的方法。(译注：这显然是接口实现本身所固有的困境，而非 Dispose 模式所专有，只是因为 Dispose 模式负有释放资源这样的重大责任而对这个问题比较敏感而已。)

实际上，如果派生类型本身没有资源清理的工作要做的话，那么它也不必再做任何与 Dispose 模式相关的事情。但是，如果派生类型需要进行一些资源清理工作的话，我们只需重写带 Boolean 参数的 Dispose 虚方法即可。在重写该方法时，我们首先要实现自身的资源清理逻辑，然后再去调用基类的带 Boolean 参数的 Dispose 方法。看下面的例子(假设 OSHandle 不是密封类)：

```
class SomeType : OSHandle {

    // 下面的字段保存着一个非托管资源的 Win32 句柄
    private IntPtr handle;

    protected override void Dispose(Boolean disposing) {
        // 同步那些同时调用 Dispose/Close 方法的线程
        lock (this) {
```

```
            try {
                if (disposing) {
                    // 对象正在被显式释放/关闭，而非执行终止化
                    // 因此在该 if 语句中访问那些引用其他对象的
                    // 字段是安全的，因为这些对象的 Finalize,
                    // 方法还没有被调用。
                    // 对于 SomeType 类来说这里没什么可做的
                }
                // 对象正在被释放/关闭、或者被执行终止化
                if (IsValid) {
                    // 如果 handle 是有效的，那么关闭非托管资
                    // 源。注意：将 CloseHandle 替换成大家自
                    // 己关闭非托管资源的函数
                    CloseHandle(handle);
                    // 将 handle 字段设置为某个标记值用来防止
                    // 多次调用 CloseHandle
                    handle = InvalidHandle;
                }
            }
            finally {
                // 让基类执行自己的清理工作
                base.Dispose(disposing);
            }
        }
    }
}
```

另一个值得注意的地方是我们在无参的 Dispose 方法中调用了 GC 的静态方法 SuppressFinalize。这是因为如果使用 OSHandle 对象的代码显式调用了 Dispose 方法或 Close 方法，那么该对象的 Finalize 方法就不应该再被执行。如果再执行 Finalize 方法，CloseHandle 函数将被多次调用。(译注：显然这不是调用 SuppressFinalize 方法的原因，因为带 Boolean 参数的 Dispose 方法中后一个 if 判断语句已经很好地解决了这个问题。调用 SuppressFinalize 方法的主要原因是它避免了终止化对象给垃圾收集器带来的沉重负担。)调用 GC 的 SuppressFinalize 方法会打开与 this 所引用的对象相关的一个位标记。当该位标记被打开时，CLR 将不会再把对象的指针从终止化链表转移到终止化可达队列上，从而阻止对象的 Finalize 方法被调用。当垃圾收集器判定该对象是可收集的垃圾时，它的内存会被立即回收。

19.4.1 使用实现了 Dispose 模式的类型

既然已经知道了怎样为一个类型实现 Dispose 模式，下面就让我们来看一下怎样使用实现了 Dispose 模式的类型。这里不再讨论前面的 OSHandle 类，而是讨论更为常见的 System.IO.FileStream 类。利用 FileStream 类，我们可以打开一个文件，从中读取字节或者向其写入字节，并最终将其关闭。

FileStream 类的内部实现和 OSHandle 类有些相似，不同的是它的构造器中调用了 Win32 函数 CreateFile，并将函数返回的结果保存在一个私有句柄字段中。FileStream 类还提供有几个额外的属性(例如 Length、Position、CanRead、Handle 等)和方法(例如 Read、Write、Flush 等)。

假设我们希望编写代码来创建一个临时文件，并向其中写入一些字节，然后再删除该文件。我们开始可能会像下面这样编写代码。

```
using System;
using System.IO;

class App {
    static void Main() {
        // 创建要写入临时文件的字节
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // 创建临时文件
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);

        // 将字节写入临时文件
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // 删除临时文件
        File.Delete("Temp.dat");    // 这里会抛出一个 IOException
    }
}
```

不幸的是，如果我们编译并运行上面的代码，它可能会工作，但大多数时候并不能。问题在于 File 的静态方法 Delete 在请求 Windows 删除一个打开的文件，Delete 会因此抛出一个 System.IO.IOException，同时还附带有一个异常的消息文本“该进程无法访问文件“Temp.dat”，因为该文件正由另一进程使用”。

但是我们要注意在有些情况下，文件实际上可能会被删除。这是因为如果在 Write 调用之后 Delete 调用之前垃圾收集线程由于某种情况而被启动，那么 FileStream 对象的 Finalize 方法将被调用，这时文件就会被关闭，随后的 Delete 操作也就可以正常运行。然而产生这种情况的可能性非常小，上面的代码在绝大多数情况下都会失败。

幸运的是，FileStream 类实现了 Dispose 模式，它允许我们显式地关闭文件。下面是修改后的代码。

```
using System;
using System.IO;

class App {
    static void Main() {
        // 创建要写入临时文件的字节
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // 创建临时文件
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);

        // 将字节写入临时文件
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // 在写入字节后显式关闭文件
         ((IDisposable) fs).Dispose();

        // 删除临时文件
        File.Delete("Temp.dat");    // 这里总会正常运行
    }
}
```

这段代码和前面的代码惟一的不同在于其中调用了 FileStream 的 Dispose 方法。Dispose 方法调用带 Boolean 参数的 Dispose 方法，带 Boolean 参数的 Dispose 方法又调用 CloseHandle 函数，Windows 于是关闭文件。这样，当 File 的 Delete 方法被调用时，Windows 会发现该文件已经关闭，因此文件删除会成功进行。

注意 FileStream 对象仍然存在于托管堆中，所以我们仍然可以调用其上的方法。最后，垃圾收集器会运行，并将该 FileStream 对象判定为可收集的垃圾。这时，垃圾收集器本来应该调用 FileStream 对象上的 Finalize 方法，但是因为 Dispose 方法调用了 GC 的 SuppressFinalize 方法，所以 Finalize 方法不会再被调用，对象的内存将直接被回收。

注意　前面的代码在调用 Dispose 方法之前首先将 fs 变量转型成了一个 IDisposable 接口。大多数实现了 Dispose 模式的类型并不需要这样的转型，但是 FileStream 对象则需要这样做，这是因为微软的开发人员将 Dispose 方法实现成了一个显式接口方法(本书第 15 章对此有描述)。我个人认为这是一种令人遗憾的做法，因为这只能使事情更复杂，也不会增添任何价值。一般情况下，只有在具有多个同名方法时，我们才应该使用显式接口方法实现。因为 Dispose 和 Close 方法的名称并不相同，所以它们都应该被实现为公有方法，从而避免调用时的转型操作。

幸运的是，FileStream 类还提供了一个公有的 Close 方法，借助该方法我们可以用更简便的代码来实现同样的功能。

```
using System;
using System.IO;

class App {
   static void Main() {

      // 建要写入临时文件的字节
      Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

      // 创建临时文件
      FileStream fs = new FileStream("Temp.dat", FileMode.Create);

      // 将字节写入临时文件
      fs.Write(bytesToWrite, 0, bytesToWrite.Length);

      // 在写入字节后显式关闭文件
      fs.Close();

      // 删除临时文件
      File.Delete("Temp.dat");    // 这里总会正常运行
   }
}
```

> **注意** 记住 Close 方法并不是 Dispose 模式正式定义的一部分，有些类型提供了 Close 方法，而有些类型则没有。

需要注意的是调用 Dispose 或 Close 方法只是在一个确定的时刻对对象占用的非托管资源执行清理操作而已，它们并不会控制托管堆中对象所使用的内存的生存期。这意味着我们在调用过 Dispose 或 Close 方法之后，仍然可以调用对象上的方法。下面的代码在文件关闭之后又调用了 FileStream 的 Write 方法试图向文件中写入更多的字节。显然，这样的操作不可能成功。当代码执行时，该 Write 方法调用将抛出一个 System.ObjectDisposedException 异常，同时还附带有一个异常的消息文本“无法访问已关闭的文件”。

```
using System;
using System.IO;

class App {
    static void Main() {

        // 创建要写入临时文件的字节
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // 创建临时文件
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);

        // 将字节写入临时文件
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // 在写入字节后显式关闭文件
        fs.Close();

        // 试图在文件关闭之后写入字节。下面一行
        // 会抛出一个 ObjectDisposedException 异常
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);

        // 删除临时文件
        File.Delete("Temp.dat");    // 这里总会正常运行
    }
}
```

注意这里不会出现内存崩溃的情况，因为 FileStream 对象的内存仍然存在。只是由于对象被执行了资源清理之后，我们便不能再成功执行某些方法而已。

> **重要** 在为我们自己的类型实现 Dispose 模式时，如果对象占用的非托管资源被执行了清理，那么所有的方法调用都要抛出一个 System.ObjectDisposedException 异常(译注：要所有的方法调用都抛出 ObjectDisposedException 异常是不合适的，只有那些在对象占用的非托管资源有效的情况下才能成功执行的方法才应该这么做)。但是 Dispose 和 Close 方法在多次调用的情况下则不应该抛出 ObjectDisposedException 异常，遇到这种情况它们应该不执行任何操作而直接返回(就像前面 OSHandle 类型中实现的那样)。

重要 一般情况下，我个人建议大家不要使用 Dispose 或者 Close 方法。理由是 CLR 的垃圾收集器已经实现的很好了，我们完全可以将工作交给它来做。垃圾收集器知道一个对象何时不再被应用程序代码所访问，直到在做出这样的判定之后它才会考虑收集对象。而当应用程序代码调用 Dispose 或 Close 方法时，它实际上是在表明自己知道应用程序已经不再需要使用该对象了。对于许多应用程序来说，这往往是不可能的。

例如，我们的代码构造了一个对象，然后又将其传递给了另外一个方法，而该方法可能会将该对象引用保存在自己的某个内部字段中(成为一个根)。我们的代码可能并不知道这些情况。如果我们的代码调用了该对象的 Dispose 或者 Close 方法，那么当其他的代码再试图操作该对象时，系统将很有可能抛出 ObjectDisposedException 异常。

但是，在需要显式清理对象占用的非托管资源(例如试图删除一个打开的文件)、或者确信操作是安全的情况下，我仍然建议大家调用 Dispose 或者 Close 方法，因为这可以阻止由于 Finalize 方法运行所导致的对象代龄的提升，从而提高应用程序的性能。

19.4.2 C#的 using 语句

前面的示例代码向大家展示了怎样显式调用一个类型的 Dispose 或者 Close 方法。如果大家决定显式调用这两个方法，强烈建议把它们放在一个异常处理的 finally 块中。因为这样可以保证它们被执行。因此，上面的示例代码更好的写法如下：

```
using System;
using System.IO;

class App {
    static void Main() {
        // 创建要写入临时文件的字节
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // 创建临时文件
        FileStream fs = null;
        try {
            fs = new FileStream("Temp.dat", FileMode.Create);
```

```
                // 将字节写入临时文件
                fs.Write(bytesToWrite, 0, bytesToWrite.Length);
            }
            finally {
                // 在写入字节后显式关闭文件
                if (fs != null)
                  ((IDisposable) fs).Dispose();
            }

            // 删除临时文件
            File.Delete("Temp.dat");    // 这里总会正常运行
        }
    }
```

像上面这样添加异常处理代码是必要。如果大家使用的是 C#，则还可以利用其中的 using 语句，该语句为我们提供了一种简化的语法来产生和上述代码相同的结果。看下面的代码：

```
using System;
using System.IO;

class App {
    static void Main() {
        // 创建要写入临时文件的字节
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };

        // 创建临时文件
        using (FileStream fs =
            new FileStream("Temp.dat", FileMode.Create)) {

            // 将字节写入临时文件
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);
        }

        // 删除临时文件
        File.Delete("Temp.dat");    // 这里总会正常运行
    }
}
```

我们首先在 using 语句内初始化一个对象，并将其引用保存在一个变量中。然后我们在和 using 语句匹配的大括号内访问该变量。当我们编译这段代码时，编译器会自动产生一个 try 块和一个 finally 块。在 finally 块中，编译器会产生代码将变量转型为一个 IDisposable 接口，并调用其上的 Dispose 方法。很明显，**using 语句只能用于那些实现了 IDisposable 接口的类型。**

注意　C#的 using 语句支持初始化多个变量的能力，前提是这些变量的类型相同。另外，using 语句还支持使用一个已经初始化的变量。有关该话题的更多信息，请参见《C#程序员参考》。

我目前还没有见过其他哪一门编程语言为实现 Dispose 模式的类型提供了如此方便的语法。在这些编程语言中，大家还必须手动添加 try 和 finally 异常处理代码。

重要　如前面一节的末尾所指出的那样，我们只应该在需要资源清理的地方调用 Dispose 或者 Close 方法。因此，我们也应该谨慎地使用 C#的 using 语句。我看到许多开发人员经常随意地使用 C#的 using 语句，但后来他们才发现自己过早地执行了资源清理，从而导致应用程序的其他部分抛出 ObjectDisposedException 异常。

19.4.3　一个有趣的依赖问题

System.IO.FileStream 类型允许用户打开一个文件进行读写操作。为了提高性能，该类型的实现使用了一个内存缓冲。只有在内存缓冲充满时，FileStream 对象才会将缓冲区中的数据填充到文件中。FileStream 类型只支持字节的读写操作。如果我们希望操作更复杂的数据类型(例如 Int32、Double、String 等)，则可以使用 System.IO.BinaryWriter 类型，下面的代码演示了可能的做法：

```
FileStream fs = new FileStream("DataFile.dat", FileMode.Create);
BinaryWriter bw = new BinaryWriter(fs);
bw.Write("Hi there");

// 下面对 Close 的调用是我们应该做的
bw.Close()
// 注意：调用 BinaryWriter.Close 会同时关闭其使用的 FileStream
// 对象。FileStream 对象在这种情况下不必显式关闭。
```

注意 BinaryWriter 的构造器接受一个 FileStream 对象作为参数。BinaryWriter 对象内部会保存该 FileStream 对象的一个引用。当我们向一个 BinaryWriter 对象写入数据时，它会将数据缓存在自己的内存缓冲区中。当其内存缓冲区充满时，BinaryWriter 对象才会将数据写入 FileStream 对象。

当我们通过 BinaryWriter 对象进行数据写入的操作执行完毕时，我们应该调用其上的 Dispose 方法或 Close 方法。(由于 BinaryWriter 实现了 Dispose 模式，所以我们也可以使用 C#的 using 语句。)这两个方法的行为相同，都会导致 BinaryWriter 对象将其内存缓冲区中的数据填充到 FileStream 对象中，同时关闭该 FileStream 对象。当 FileStream 对象被关闭时，它会首先将自己缓冲区中的内容填充到磁盘文件中，然后才会调用 CloseHandle 函数。

> **注意** 我们不必显式调用 FileStream 对象的 Dispose 或 Close 方法，因为 BinaryWriter 已经为我们做了这件事。如果我们调用了 Dispose 或 Close 方法，FileStream 对象会发现它已经被执行了资源清理，Dispose 或 Close 方法将不执行任何操作而直接返回。

那么如果我们没有显式调用 BinaryWriter 对象的 Dispose 或 Close 方法会出现什么情况呢？我们知道垃圾收集器能够正确地检测出对象是否已成为可收集的垃圾，如果是，垃圾收集器将对它们执行终止化操作。但是垃圾收集器并不能保证多个对象上的 Finalize 方法的执行顺序，所以如果首先被执行终止化操作的是 FileStream 对象的话，它将会关闭文件。然后，当 BinaryWriter 对象被执行终止化时，它将试图向一个已关闭的文件中写入数据，这自然会抛出异常。但是，如果首先被执行终止化的是 BinaryWriter 对象的话，其中的数据将会被安全地写入到文件中。

那么微软是如何解决这个问题的呢？使垃圾收集器以特定的顺序来执行对象终止化操作显然不可能，因为对象之间可能包含着对彼此的引用，垃圾收集器根本无法正确地推断出对它们执行终止化的顺序。微软的解决方法是不让 BinaryWriter 类重写 Finalize 方法。这意味着如果我们忘记了显式关闭 BinaryWriter 对象，其内存缓冲区中的数据必然会丢失。微软希望开发人员能够认识到这一点，并在自己的代码中显式调用 Dispose 或 Close 方法。

19.5 弱 引 用

我们知道，当一个根指向一个对象时，该对象不可能被执行垃圾收集，因为应用程序代码还可以访问该对象。在这种情况下，我们通常说存在一个该对象的强引用(strong reference)。垃圾收集器还支持弱引用(weak reference)的概念。弱引用允许垃圾收集器收集对象，同时也允许应用程序访问该对象，结果是哪一个要取决于时间。

如果只有对象的弱引用存在，那么在垃圾收集执行后，该对象的内存将被回收，应用程序再试图访问对象时会告失败。另一方面，要访问一个弱引用对象，应用程序必须获取该对象的一个强引用。如果应用程序在对象被执行垃圾收集之前获得了它的强引用，那么垃圾收集器将不能对该对象执行垃圾收集，因为这时存在一个该对象的强引用。

是否对上面的叙述感到有些困惑？我们来看一段代码，这会帮助大家搞清楚弱引用的真正含义。

```
void SomeMethod() {
    // 创建一个新对象的强引用
    Object o = new Object();

    // 创建一个短弱引用对象。
    // 该对象负责追踪对象 o 的生存期
    WeakReference wr = new WeakReference(o);

    o = null;           // 移除对象的强引用

    o = wr.Target;

    if (o == null) {
        // 出现过垃圾收集，对象的内存已经被回收
    } else {
        // 未出现过垃圾收集，所以我们可以成功地
        // 使用变量 o 来访问对象
    }
}
```

那么我们为什么需要弱引用呢？我们经常会用到这样一些数据结构，它们很容易创建但是却需要大量的内存。例如，我们可能会有一个应用程序需要知道用户硬盘中所有的目录和文件。我们可以很容易地构造一个树来反映这些信息，当应用程序运行时，它就可以引用内存中的树，而不必再访问用户的硬盘。这显然会极大地提高应用程序的性能。

但问题在于这个树可能会非常庞大，需要许多内存。如果用户转而访问应用程序的其余部分，那么这个树可能变得不再必要，但却仍然浪费着许多内存。我们可能会放弃对这个树的根对象的引用，但是如果用户又切换回到应用程序的第一部分，那么我们又需要重新构造这个树。弱引用使得我们可以方便、高效地处理这种情况。

当用户离开应用程序的第一部分时，我们可以创建一个弱引用对象来引用这个树的根对象，而放弃其所有的强引用。如果应用程序其他组件的内存负载比较低，垃圾收集器将不会回收这个树的内存。

当用户重新切换回应用程序的第一部分时，应用程序可以尝试获得对这个树的根对象的强引用。如果成功，应用程序将不再需要遍历用户的硬盘。

System.WeakReference 类型提供有两个公有构造器：

```
public WeakReference(Object target);
public WeakReference(Object target, Boolean trackResurrection);
```

参数 target 表示 WeakReference 要追踪的对象。参数 trackResurrection 表示 WeakReference 是否要追踪对象复苏，换句话说它表示在对象的 Finalize 方法被调用之后，WeakReference 是否还应该追踪对象。通常情况下，我们为参数 trackResurrection 传递的值为 false，实际上第一个构造器在内部就是这样调用第二个构造器的。

为方便起见，我们将不追踪对象复苏的 WeakReference 称作一个短弱引用(short weak reference)，而将追踪对象复苏的 WeakReference 称作长弱引用(long weak reference)。如果一个对象的类型没有重写 Finalize 方法，那么短弱引用和长弱引用的行为是一样的。强烈建议大家不要使用长弱引用，因为长弱引用在一个对象被执行终止化后仍允许我们使该对象重新复苏，而这会导致对象状态的不可预期。

一旦我们创建了一个对象的弱引用，我们通常要将该对象的强引用设置为 null。如果有任何该对象的强引用存在，垃圾收集器都不会对该对象执行垃圾收集。

为了再次使用对象，我们必须将弱引用转换为一个强引用，这可以通过查询 WeakReference 对象的 Target 属性，并将结果赋值给应用程序的一个根来完成。如果 Target 属性的返回值为 null，那么对象已经被执行了垃圾收集。如果 Target 属性的返回值不为 null，那么我们将得到对象的一个强引用，应用程序代码也就可以继续操作该对象。只要存在对象的强引用，它就不可能被执行垃圾收集。

19.5.1 弱引用的内部机理

从前面的讨论中我们可以看出 WeakReference 对象和其他对象的行为有很大差别。通常情况下，如果我们的应用程序有一个根引用了一个对象，而该对象又引用了另一个对象，那么这两个对象都将是可达对象，二者都不可能被执行垃圾收集。但是，如果我们的应用程序有一个根引用了一个 WeakReference 对象，那么被该 WeakReference 对象引用的对象将不再被认为是可达对象，并且可以被执行垃圾收集。

为了完全理解弱引用的工作原理，我们再来深入探究一下托管堆。托管堆中包含有两个内部数据结构专门用来管理弱引用，即短弱引用表和长弱引用表。这两个表中只是包含着一些指针，它们引用着托管堆中的对象。

刚开始的时候，这两个表都为空。当我们创建一个 WeakReference 对象时，垃圾收集器并不会从托管堆中为其分配内存。相反，它会在两个弱引用表中选择一个(短弱引用使用短弱引用表，长弱引用使用长弱引用表)，并在其中寻找一个空白插槽。

一旦找到一个空白插槽，该插槽的值将被设为我们希望追踪的对象的地址——也就是传递给 WeakReference 构造器的那个对象指针，new 操作符返回的值就是相应弱引用表中插槽的地址。显然，这两个弱引用表不会被认为是应用程序的根，否则垃圾收集器将不能收集它们中的指针引用的对象。

下面是垃圾收集器运行时发生的一系列事情：

1. 垃圾收集器构造一个包含所有可达对象的图。前面对此已经做过详细的介绍。
2. 垃圾收集器扫描短弱引用表。如果该表中有指针引用的对象不是前面构造的可达对象图的一部分，那么该指针标识的将是一个不可达对象，短弱引用表中对应插槽的值将被设为null。
3. 垃圾收集器扫描终止化链表。如果该链表中有指针引用的对象不是前面构造的可达对象图的一部分，那么该指针标识的将是一个不可达对象，它将被从终止化链表转移到终止化可达队列上。这时，对象又成为可达对象图的一部分。
4. 垃圾收集器扫描长弱引用表。如果该表中有指针引用的对象不是可达对象图(该图现在包括终止化可达队列中引用的对象)的一部分，那么该指针标识的将是一个不可达对象，长弱引用表中对应插槽的值将被设为null。
5. 垃圾收集器压缩内存，填充不可达对象空出的位置。注意有时候如果垃圾收集器判定不可达对象空出的内存碎片数量不值得耗费时间去压缩，那么它将不会执行这一步。

一旦大家理解了垃圾收集器背后的工作原理，理解弱引用也就很容易了。查询 WeakReference 的 Target 属性会导致系统返回相应弱引用表中插槽的值。如果返回的插槽中的值为 null，那就证明对象已经被执行了垃圾收集。

短弱引用并不追踪对象复苏。这意味着只要垃圾收集器判定对象成为不可达对象，它就会把短弱引用表中对应的指针设为 null。如果对象的类型重写了 Finalize 方法，那么这时该方法还没有被调用，所以对象应该仍然存在。如果应用程序访问 WeakReference 对象的 Target 属性，返回值将为 null，虽然这时对象仍然存在。

长弱引用追踪对象复苏。这意味着只有在垃圾收集器认为对象的存储空间可以被回收时，它才会将长弱引用表中对应的指针设为 null，这时的对象也不可能再重新复苏。如果对象的类型重写了 Finalize 方法，那么这时该方法应该已经被调用。

19.6 对象复苏

相信很多人都对终止化操作很感兴趣。实际上，终止化的内容远不止前面所描述的那些。大家可能已经注意到了这样一个现象，当一个需要终止化的对象被认为“死亡”时，垃圾收集器可以强制使该对象获得“重生”，因为只有这样才能调用它的 Finalize 方法。在它的 Finalize 方法被调用之后，它才算真正地“死亡”了。总的来说，一个需要终止化的对象会经历“死亡”、“重生”、然后再“死亡”的过程。这个有趣的现象被称为复苏(resurrection)。顾名思义，复苏就是使一个对象“死而复生”。

准备调用一个对象的 Finalize 方法的行为就是一种复苏的形式。当垃圾收集器将对象的一个引用放到终止化可达队列中时，对象就成为一个可达对象，可以说获得了“重生”。但在对象的 Finalize 方法被调用之后，再没有任何根指向对象，这时对象才算真正地“死亡”了。但是如果我们在一个对象的 Finalize 方法中将该对象的指针再放入一个全局变量或静态变量中又会出现什么情况呢？看下面的代码。

```
class SomeType {
    ~SomeType() {
        Application.ObjHolder = this;
    }
}

class Application {
    public static Object ObjHolder;     // 默认值为 null
    ...
}
```

在上面的代码中，当 SomeType 对象的 Finalize 方法被执行时，该对象的一个引用将被放入一个根中，从而使其又成为应用程序中的一个可达对象。对象重新复苏以后，垃圾收集器不会再将其认为是可收集的垃圾，应用程序也可以自由地使用该对象。但是我们必须记住该对象已经被执行了终止化，所以使用它可能会导致不可预期的结果。另外需要注意的是如果 SomeType 中包含有引用其他对象的字段(直接或者间接)的话，所有被引用的对象都将重新复苏，因为它们现在也都是可达对象。但是，我们需要注意这些重新复苏的对象中的一些可能也已经被执行了终止化。

> **注意** 我们定义的任何类型都可能在某一点因重新复苏而脱离我们的控制。换句话说一个对象在其 Finalize 方法被调用之后，它的成员仍然可能会被访问。在一个理想的环境中，我们可能会考虑添加代码来检查一个对象的 Finalize 方法是否已经执行，然后根据检查结果来对成员访问做相应的处理(比如抛出适当的异常)。但是在实践中我们必须为此编写许多繁冗的代码，我怀疑很多开发人员是否对每一个类型都能设计的如此周到。

如果有代码将 Application.ObjHolder 设置为 null，那么对象就又成为不可达对象。最后，垃圾收集器将对象判定为可收集的垃圾，并回收其在托管堆中的内存。因为终止化链表上不再有对象的指针，所以其 Finalize 方法将不会再被调用。

对象复苏听起来很酷，但是能够很好地使用它却并不容易，我们应该尽可能地避免使用这项技术。当开发人员使用对象复苏时，他们通常希望对象在每次“死亡”时都能够顺利地清理自己的非托管资源。为了做到这一点，GC 类提供了一个名为 ReRegisterForFinalize 的静态方法，该方法接受一个类型为 System.Object 的参数。下面是前面代码改进后的一个版本。

```
class SomeType {
    ~SomeType() {
        Application.ObjHolder = this;
        GC.ReRegisterForFinalize(this);
    }
}
```

当 Finalize 方法被调用时，它首先用一个应用程序的根来引用对象，从而使其重新复苏。然后，Finalize 方法又调用了静态方法 ReRegisterForFinalize，将指定对象(this)的地址添加到终止化链表的末端。当未来某个时刻垃圾收集器检测到对象不可达时，它会将对象的指针转移到终止化可达队列上，对象的 Finalize 方法将再一次被调用。

上面的例子向大家展示了如何创建一个不断复苏、永不“死亡”的对象——但是我们通常不会希望对象有这样的行为，更常见的做法是在 Finalize 方法内部根据一定的条件来使对象重新复苏。

> **注意** 我们要确保每次对象复苏时 ReRegisterForFinalize 方法都只能被调用一次，否则对象的 Finalize 方法将会被多次调用。因为每调用 ReRegisterForFinalize 方法一次，它就会在终止化链表上添加一个条目。当对象被判定为可收集的垃圾时，终止化链表上所有这些条目都将被转移到终止化可达队列上，从而导致对象的 Finalize 方法被多次调用。

19.6.1 利用复苏设计一个对象池

下面的示例可以向大家展示对象复苏的应用价值。假设我们希望创建一个 Expensive 对象池。由于这些对象非常耗费资源，创建它们需要花费很多时间。处于性能考虑，我们打算在应用程序启动之时就能创建一组 Expensive 对象，然后在应用程序的整个生存期中重复地使用它们。

下面是管理 Expensive 对象池的示例代码。

```
using System;
using System.Collections;

// 下面这个类的实例构造起来比较耗费资源
class Expensive {

    // 下面的静态 Stack 中包含着
    // 对象池中可用对象的引用
    static Stack pool = new Stack();

    // 下面的静态方法返回从对象池中取得的对象
    public static Expensive GetObjectFromPool() {
        // 从对象池中获取一个对象，并将其引用从对象池中删除
        return (Expensive) pool.Pop();
    }

    // 当应用程序关闭时，调用下面的静态方法销毁对象池
    public static void ShutdownThePool() {

        // 阻止终止化对象将自己添加到对象池中
        pool = null;
    }
```

```
    // 下面的构造器创建一个对象并将其添加到对象池中
    public Expensive() {
        // 构造对象要花费比较长的时间
        ...

        // 在对象构造完之后，将其添加到对象池中
        pool.Push(this);
    }

    // 当应用程序不再需要对象时，下面的 Finalize 方法将被调用
    ~Expensive() {

        // 如果应用程序没有关闭，就将对象重新
        // 添加到对象池中
        if (pool != null) {

            // 调用 ReRegisterForFinalize 方法以使对象能够以
            // 正确的状态加入到对象池中
            GC.ReRegisterForFinalize(this);

            // 将对象重新加入到对象池中
            pool.Push(this);
        }
    }
}

class App {
    static void Main() {
        // 构造一组 Expensive 对象填充对象池
        for (Int32 i = 0; i < 10; i++)
            new Expensive();
        ...

        // 当我们需要对象时，就从对象池中获取
        Expensive e = Expensive.GetObjectFromPool();
        // 应用程序下面就可以使用 e 了
        ...

        // 要关闭应用程序，首先关闭对象池
        Expensive.ShutdownThePool();
    }
}
```

Expensive 中定义了一个类型为 System.Collections.Stack 的私有静态字段 pool，该字段负责管理对象池中可用的对象。在 Main 中，我们用一个循环构造了 10 个 Expensive 对象。当每个对象被构造时，它会将自己添加到对象池中。Expensive 的静态字段 pool 本身是一个应用程序的根，它引用着一组 Expensive 对象，这些对象一经创建便不可能被执行垃圾收集。

当应用程序需要使用 Expensive 对象时，它就调用 Expensive 的静态方法 GetObjectFromPool。该方法从对象池中返回一个对象引用，同时将其从对象池中删除。应用程序自此便可以使用 Expensive 对象了。

随着时间的推移，应用程序将不再持有前面返回的 Expensive 对象引用。如果在这之后出现了垃圾收集，那么没有被任何根所引用的 Expensive 对象将被执行终止化。当 Expensive 对象的 Finalize 方法被调用时，它会将自己再次加入到对象池中，从而使其重新复苏，阻止垃圾收集器回收其内存。另外，Finalize 方法还会调用 GC 的 ReRegisterForFinalize 方法。在未来的某个时刻，该 Expensive 对象还会再次被应用程序所获取，并在随后的某个时刻再次成为不可达对象，垃圾收集器也会再次被启动。由于我们前面调用了 ReRegisterForFinalize 方法，所以 Finalize 方法还会再次被调用，从而再次将对象加入到对象池中。

在 Main 退出之前，它会调用 Expensive 的静态方法 ShutdownThePool，该方法会将 pool 字段设为 null。当应用程序关闭时，CLR 会为在托管堆中的所有对象调用 Finalize 方法。这时我们不应该再去调用 ReRegisterForFinalize 方法，因为这样做会导致一个无限循环(CLR 会在 40 秒后强制中断进程)。因此我们会在 Expensive 的 Finalize 方法中检测 pool 字段。如果该字段为 null，那么 Expensive 对象将不会再被添加到终止化链表上，同时也不会被添加到对象池中，在这之后，Expensive 对象的内存将可以被回收。

如我们所见，对象复苏为实现对象池提供了一种简单有效的方式。

19.7　对象的代龄

如本章开始所言，代龄是旨在提高垃圾收集器性能的一种机制。一个基于代龄的垃圾收集器(又称季节性垃圾收集器，ephemeral garbage collector，本书不拟使用这一术语)有以下几点假设：

- 对象越新，其生存期越短。
- 对象越老，其生存期越长。
- 对托管堆的一部分执行垃圾收集要比对整个托管堆执行垃圾收集速度更快。

这些假设经过了大量的研究，已经在很多现存的应用程序上得到了验证，它们影响着垃圾收集器的实现。本节将解释垃圾收集器中代龄的工作机制。

在托管堆初始化时，其中不包括任何对象。这时添加到托管堆中的对象被称为第 0 代对象。简单地说，第 0 代对象就是那些新构造的对象，垃圾收集器还没有对它们执行过任何检查。图 19.7 向我们展示了一个新启动的应用程序中托管堆的情况，其中分配有 5 个对象(A 到 E)。经过一段时间后，对象 C 和对象 E 将变为不可达对象。

图 19.7 一个刚经过初始化的托管堆，其中包含着一些对象，这时所有的对象都处于第 0 代，垃圾收集还没有执行过

当 CLR 初始化时，它会为第 0 代对象选择一个阈值容量，假定为 256 KB。(实际的容量可能会与此不同。)当分配新对象导致第 0 代对象超过了为其设定的阈值容量时，垃圾收集器就必须启动了。假设从对象 A 到对象 E 总共占用了 256 KB，那么当对象 F 被分配时，垃圾收集器就会启动。垃圾收集器判定对象 C 和 E 为垃圾对象，因此会压缩对象 D 使其邻接于对象 B。在此次垃圾收集中存活下来的对象(对象 A、B、和 D)将被认为是第 1 代对象，它们经过了一轮垃圾收集检查。这时托管堆的情况如图 19.8 所示。

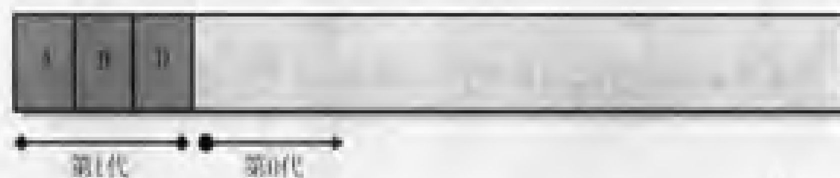


图 19.8 经过一次垃圾收集之后，第 0 代对象中的存活者被提升为第 1 代对象，第 0 代对象暂时空缺

在第一次垃圾收集执行后，第 0 代对象暂时空缺。但很快，随着应用程序的运行，又有新的对象被分配而成为第 0 代对象。图 19.9 中展示的对象 F 到对象 K 就是这样的情况。应用程序继续运行，对象 B、H 和 J 又成为不可达对象，它们的内存也要在某一点被回收。

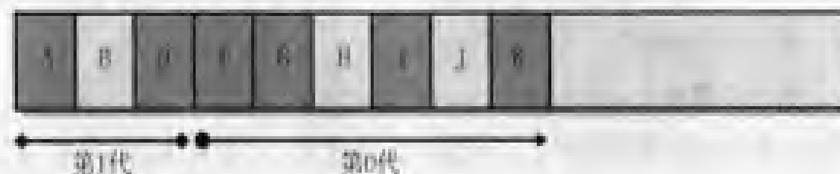


图 19.9 第 0 代中又分配了新的对象，第 1 代中某些对象已经成为垃圾对象

现在假设应用程序又试图分配对象 L，这将再一次使第 0 代对象超过它的阈值容量，于是又必须开始执行第二次垃圾收集。在这之前，垃圾收集器必须判定要收集哪些代的对象。前面曾说过，CLR 初始化的时候会为第 0 代对象选择一个阈值容量。实际上，它也会为第 1 代对象选择一个阈值容量。我们假设其为第 1 代对象选择的阈值容量为 2 MB。

当第二次垃圾收集开始执行时，它也会查看第 1 代对象占用了多少内存。在本例中，由于第 1 代对象占用的内存远少于 2 MB，所以垃圾收集器只会去检查第 0 代对象。在继续讨论之前，我们再来回顾一下一个基于代龄的垃圾收集器所做的几点假设。其第一个假设是新创建的对象的生存期比较短。所以第 0 代对象中成为垃圾对象的数量会比较多，因此对第 0 代对象执行垃圾收集将有可能回收比较多的内存，忽略掉第 1 代对象会提高垃圾收集的速度。

然而忽略第 1 代对象的意义不仅仅在于不对它们执行垃圾收集。实际上，更重要的是垃圾收集器不用再遍历整个托管堆中的所有对象了，这对垃圾收集器的性能提升具有重大的意义。如果一个根或者一个对象引用了一个代龄较大的对象，那么垃圾收集器就可以忽略一些这样的内部引用，从而减少构造可达对象图所需的时间。说忽略代龄较大的对象中的一些内部引用，而不是全部内部引用是因为一个代龄较大的对象有可能引用一个新的对象。为了确保这些位于代龄较大的对象中的一些新对象也能被检查到，垃圾收集器使用 JIT 编译器内部的一种机制来在对象的内部引用字段改变时设置一个相应的位标记。这使得垃圾收集器可以知道自从上次垃圾收集执行以来，哪些代龄较大的对象已经被应用程序所改写。这样垃圾收集器只需要检查它们是否引用了第 0 代对象就可以了。

> **注意**　微软的性能测试显示在 Pentium 200-MHZ 的机器上对第 0 代对象执行一次垃圾收集所花费的时间不超过 1 毫秒。微软的目标是使垃圾收集所花费的时间不超过一个普通的内存页面错误所花费的时间。

一个基于代龄的垃圾收集器所做的第二个假设是生存期比较长的对象将倾向于继续存活，所以第 1 代对象继续成为可达对象的可能性比较大。因此，如果垃圾收集器检查第 1 代对象，很有可能找不到很多垃圾对象，能够回收的内存也就有限。这样以来，收集第 1 代对象也就很有可能会浪费大量的时间。所以如果有垃圾对象位于第 1 代，那么它仍然会呆在那里。在第二次垃圾收集执行后，托管堆的情况将如图 19.10 所示。

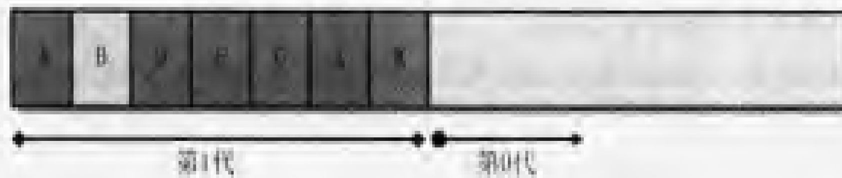


图 19.10 经过两次垃圾收集之后，第 0 代对象中的存活者被提升为第 1 代（第 1 代内存总量有所增长），第 0 代暂时空缺

如我们所见，所有第 0 代对象中的存活者现在又成了第 1 代的一部分。因为垃圾收集器没有检查第 1 代，所以对象 B 的内存并没有被回收，虽然它在第二次垃圾收集执行时已经成为不可达对象。同样，在第二次垃圾收集执行后，第 0 代对象暂时空缺，只有等着分配新的对象。接着，应用程序继续运行，并分配对象 L 到对象 O。过一段时间后，对象 G、L 和 M 又成为不可达对象。图 19.11 展示了这时托管堆的情况。

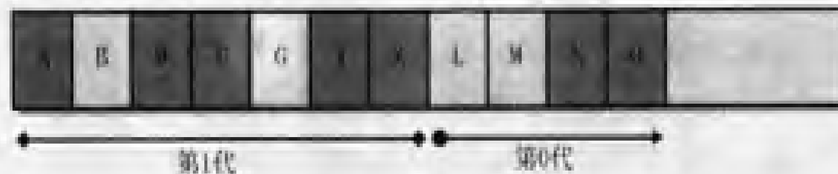


图 19.11 新对象被分配成为第 0 代，第 1 代中有了更多的垃圾

假设分配对象 P 又导致了第 0 代对象超过它的阈值容量，于是垃圾收集又被启动。因为第 1 代中所有对象的内存总量仍小于 2 MB，所以垃圾收集器仍会决定只收集第 0 代对象，而忽略第 1 代中的不可达对象(对象 B 和 G)。在这次垃圾收集执行完毕后，托管堆的情况将如图 19.12 所示。

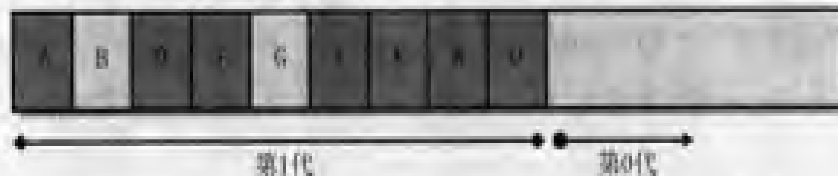


图 19.12 经过三次垃圾收集之后，第 0 代对象中的存活者被提升为第 1 代（第 1 代内存总量再次增长），第 0 代暂时空缺

在图 19.12 中，我们看到第 1 代对象在缓慢增长。现在，让我们假设第 1 代对象的增长导致其内存总量到达了 2 MB 这一阈值容量。这时，应用程序继续运行(因为垃圾收集刚刚完成)，并分配对象 P 到对象 S，这使得第 0 代也达到它的阈值容量。这时托管堆的情况将如图 19.13 所示。

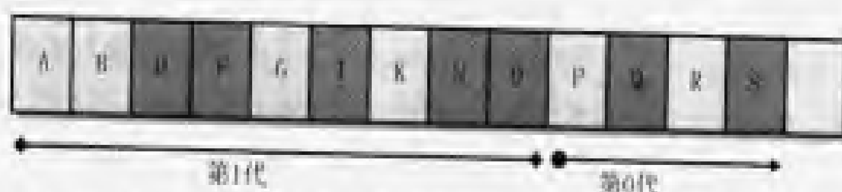


图 19.13　新对象被分配成为第 0 代，第 1 代中有了更多的垃圾

当应用程序试图分配对象 T 时，因为第 0 代对象已经充满(即达到设定的阈值容量)，所以必须执行垃圾收集。但是这一次垃圾收集器会发现第 1 代对象的内存总量已经超过了 2 MB 这一阈值——因为在前面几次对第 0 代对象的收集中，许多第 1 代对象已经成为了垃圾对象。所以这次垃圾收集器会同时收集第 0 代和第 1 代中所有的垃圾对象。在这次垃圾收集执行完毕后，托管堆的情况将如图 19.14 所示。

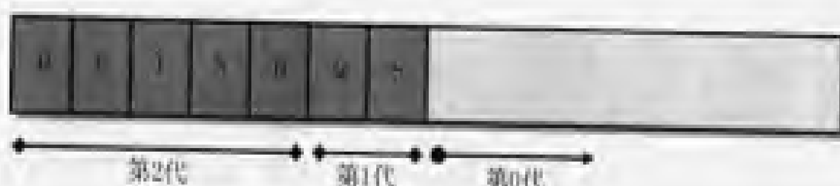


图 19.14　经过四次垃圾收集之后，第 1 代中的存活者被提升为第 2 代，第 0 代中的存活者被提升为第 1 代，第 0 代暂时空缺

和前面一样，在这次垃圾收集执行完毕后，所有第 0 代中的存活者将被提升为第 1 代，而第 1 代中的存活者将被提升为第 2 代，第 0 代对象则暂时空缺。其中第 2 代中的对象至少经过了两次垃圾收集的检查。在产生第 2 代对象之前，系统有可能已经执行了多次垃圾收集，但只有在第 1 代对象的内存总量达到它的阈值容量时，垃圾收集器才会检查第 1 代对象，而在这之前系统可能已经对第 0 代对象执行了好几次垃圾收集。

CLR 的托管堆只支持 3 个代龄：第 0 代、第 1 代和第 2 代，没有第 3 代。当 CLR 初始化时，它会为这三代选择 3 个阈值容量。如前所述，第 0 代的阈值容量大约为 256 KB，第 1 代的阈值容量大约为 2 MB，第 2 代的阈值容量大约为 10 MB。选择阈值容量是为了提高系统性能，阈值容量越大，垃圾收集执行的频率也就越低。性能提升源于下面的初始假设：新对象的存活时间比较短，而老对象则倾向于继续存活。

另外，CLR 的垃圾收集器还是一个自调节的垃圾收集器。这意味着垃圾收集器会在执行垃圾收集的过程中学习应用程序的行为。例如，如果我们的应用程序构造了许多对象，并在很短的时间里使用它们，那么垃圾收集器很有可能在第 0 代就能回收掉许多内存。实际上，所有第 0 代的对象都有可能被回收掉。

如果垃圾收集器发现在第 0 代对象被收集以后存活下来的对象很少，那么它可能会决定将第 0 代的阈值容量从 256 KB 减少到 128 KB。这将使得垃圾收集执行的频率变得更高，但是每次需要做的收集工作将更少，这样以来应用程序进程的工作集便不会增长的过大。实际上，如果第 0 代中所有的对象都是垃圾对象的话，垃圾收集器将不必压缩任何内存，它可以直接将 NextObjPtr 指针移到第 0 代对象的开始之处来完成垃圾收集——这恐怕是回收内存最快的方法了。

> **注意** 垃圾收集器在 ASP.NET Web 窗体和 XML Web 服务应用程序中工作的情况相当出色。对于 ASP.NET 应用程序来说，当客户请求到达时，会有许多新的对象被构造，这些对象会根据客户的行为执行某些操作，然后将结果返回给客户。之后，所有用来响应客户请求的对象都将成为垃圾对象。换句话说，每一次 ASP.NET 应用程序请求都会导致产生许多垃圾对象。因为这些对象几乎在它们被创建之后立即就成为了不可达对象，所以每次垃圾收集能够回收掉许多内存。这将使进程的工作集保持在非常低的状态，垃圾收集器的性能自然也更好。

另一方面，如果垃圾收集器收集了第 0 代对象之后看到还有很多对象存活(也就是说回收的内存并不多)的话，垃圾收集器将把第 0 代的阈值容量向上调整，比如调整到 512 KB。这样，垃圾收集执行的频率将降低，但是每一次回收的内存将比较多。

上面仅仅讨论了每次垃圾收集执行后动态调整第 0 代阈值容量的情况，但实际上，垃圾收集器也使用类似的启发式算法来调整第 1 代和第 2 代的阈值容量。当位于这些代龄中的对象被执行垃圾收集时，垃圾收集器会查看回收的内存总量和存活的对象总量。基于这些结果，垃圾收集器会调整各个代的阈值容量，从而达到提高应用程序性能的目的。

19.8 编程控制垃圾收集器

System.GC 类型为我们的应用程序提供了直接控制垃圾收集器的一些方法。例如我们可以通过读取 GC.MaxGeneration 属性来查询托管堆支持的最大代龄。该属性目前的返回值为 2。

我们也可以通过调用下面两个静态方法中的任何一个来强制执行垃圾收集：

```
void GC.Collect(Int32 Generation)
void GC.Collect()
```

其中第一个方法允许我们指定收集某一特定代龄的对象。我们可以为其传递在 0 和 GC.MaxGeneration 之间(包括 GC.MaxGeneration)的任何整数。传递 0 将导致第 0 代对象被执行垃圾收集，传递 1 将导致第 0 代和第 1 代对象被执行垃圾收集，传递 2 将导致第 0 代、第 1 代和第 2 代对象被执行垃圾收集。无参的 Collect 方法将强制对所有代龄的对象执行垃圾收集，它等同于下面的方法调用：

```
GC.Collect(GC.MaxGeneration);
```

大多数情况下，我们应该避免调用这些 Collect 方法，而让垃圾收集器根据自己的判断来执行，并根据应用程序的实际行为调整各个代龄的阈值容量。但是，如果我们正在编写的是 CUI 或 GUI 应用程序，我们的应用程序代码将“拥有”整个进程以及进程中的 CLR。对于这些应用程序，我们可能会希望在某些时候能够强制执行垃圾收集。

例如，在用户保存了一个数据文件之后，我们可能希望能够强制执行一次完全的垃圾收集(收集所有代龄的对象)。我们也可能希望 Web 浏览器在每次页面卸载后也能执行一次完全的垃圾收集。我们还可能希望应用程序在执行一些较费时的操作时能强制执行一次垃圾收集。这里的基本思想是当应用程序执行较为耗时的操作时，垃圾收集的执行时间可以被掩盖起来。在上面 3 个应用场景中，由于应用程序的其他一些操作会占用大量的时间，所以用户将感觉不到垃圾收集的存在。

GC 类型还为我们提供了一个 WaitForPendingFinalizers 方法。该方法会挂起调用线程，直到处理终止化可达队列的线程清空了该队列，并完成每个对象的 Finalize 方法调用为止。在大多数应用程序中，我们一般没有必要调用该方法。但是，我个人却看到过如下的代码：

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

上面的代码首先强制执行一次垃圾收集。当第一轮垃圾收集完成后，不需要终止化的那些对象的内存将被回收。但是终止化对象的内存还没有被回收。在第一次 Collect 调用返回后，一个特殊的、专门用于终止化的线程将采用异步的方式来调用所有终止化对象的 Finalize 方法。WaitForPendingFinalizers 方法将使应用程序线程处于睡眠状态，直到所有的 Finalize 方法调用完成为止。当 WaitForPendingFinalizers 方法返回时，所有的终止化对象将成为真正的可收集垃圾。这时，第二次 Collect 调用将强制执行第二轮垃圾收集，所有终止化对象的内存将在这一轮垃圾收集中被完全回收。

最后，GC 还提供有两个静态方法供我们确定一个对象所处的代龄：

```
Int32 GetGeneration(Object obj)
Int32 GetGeneration(WeakReference wr)
```

第 1 个版本的 GetGeneration 接受一个对象引用作为参数，第 2 个版本的 GetGeneration 接受一个 WeakReference 引用作为参数。两个方法的返回值在 0 和 GC.MaxGeneration 之间。

下面的代码可以帮助大家理解代龄的工作原理，其中也演示了如何使用上面提到的一些 GC 的方法。

```
using System;

class GenObj {
    ~GenObj() {
        Console.WriteLine("In Finalize method");
    }
}

class App {
    static void Main() {
        Console.WriteLine("Maximum generations: " + GC.MaxGeneration);

        // 在托管堆中创建一个新的 GenObj 对象
        Object o = new GenObj();

        // 因为该对象为新创建的对象，所以它的代龄应该为 0
        Console.WriteLine("Gen " + GC.GetGeneration(o));     // 0

        // 执行垃圾收集提升对象的代龄
        GC.Collect();
        Console.WriteLine("Gen " + GC.GetGeneration(o));     // 1
```

```
        GC.Collect();
        Console.WriteLine("Gen " + GC.GetGeneration(o));     // 2

        GC.Collect();
        Console.WriteLine("Gen " + GC.GetGeneration(o));     // 2 (最大值)

        o = null;       // 销毁对象的强引用

        Console.WriteLine("Collecting Gen 0");
        GC.Collect(0);                          // 收集第 0 代对象
        GC.WaitForPendingFinalizers();          // 不会调用 Finalize

        Console.WriteLine("Collecting Gen 1");
        GC.Collect(1);                          // 收集第 1 代对象
        GC.WaitForPendingFinalizers();          // 不会调用 Finalize

        Console.WriteLine("Collecting Gen 2");
        GC.Collect(2);                          // 等同于 Collect()
        GC.WaitForPendingFinalizers();          // 调用 Finalize
    }
}
```

编译并运行上面的代码，我们将会看到以下输出：

```
Maximum generations: 2
Gen 0
Gen 1
Gen 2
Gen 2
Collecting Gen 0
Collecting Gen 1
Collecting Gen 2
In Finalize method
```

19.9 其他一些与垃圾收集器性能相关的问题

本章前面在讨论垃圾收集算法时做了一个大胆的假设，即应用程序中只有一个线程在运行。但在现实世界中，经常会出现多个线程同时访问托管堆的情况(至少会有多个线程同时操作托管堆中分配的对象)。如果因为某个线程的执行而导致了垃圾收集的运行，那么其他线程将不能再访问任何对象(包括各线程自己堆栈上的对象引用)，因为垃圾收集器随时都有可能搬移这些对象、改变它们的内存地址。

因此当垃圾收集器开始运行时，所有执行托管代码的线程都必须被挂起。CLR 使用几种不同的机制来在执行垃圾收集时保证安全地挂起线程。存在多种机制的目的是为了尽可能地让线程保持运行，从而减少不必要的开销。本书不打算过多地深入探讨这里的一些细节，大家只需清楚微软在这方面确实已经做了大量的工作。随着时间的推移，这些机制将会得到持续的改进，垃圾收集器也会变得更加高效。

当 CLR 开始执行垃圾收集时，它会立即挂起进程中所有已经执行过托管代码的线程。接着，CLR 会检查每个线程的指令指针以判断线程执行到了哪里。然后，CLR 将指令指针地址和 JIT 编译器产生的表做比较，从而找出线程正在执行的代码。

如果线程的指令指针位于某个表中标识的偏移之处，那么该线程将被认为是到达了一个安全点(safe point)。一个安全点就是可以在垃圾收集执行完毕之前一直挂起线程的地方。如果线程的指令指针没有位于某个内部方法表中标识的偏移之处，那么该线程就没有到达一个安全点，CLR 也就不能执行垃圾收集。在这种情况下，CLR 会劫持(hijack)该线程——也就是改变线程的堆栈，以使其返回地址指向 CLR 内部实现的一个特殊函数。然后，该线程将被允许继续执行。当目前执行的方法返回时，CLR 内部的特殊函数将执行，从而挂起该线程。

然而，线程有时候并不能很快就从当前的方法中返回。所以在线程继续执行后，CLR 会留出大约 250 毫秒的时间等待线程被劫持。过了这个时间之后，CLR 再次挂起线程，并检测其指令指针。如果线程到达了一个安全点，那么垃圾收集就可以开始运行。如果线程仍然没有到达一个安全点，那么 CLR 将检测看是否该方法内部又调用了其他的方法。如果确实调用了其他的方法，那么 CLR 将再一次改变线程的堆栈，以使线程在最近执行的方法中返回时就能被劫持。然后，线程被允许继续执行，CLR 会再等一段时间，之后再次尝试劫持线程。

当所有的线程都到达了一个安全点、或者被劫持后，垃圾收集才可以开始运行。当垃圾收集完成后，所有的线程将被允许继续执行，应用程序也将恢复运行。被劫持的线程也会返回到原来调用它们的方法中。

注意 上面提到的算法中还有一个额外的细节需要大家注意。如果 CLR 挂起了一个线程，并检测到该线程正在执行非托管代码，那么该线程的返回地址也会被劫持，并被允许继续执行。但是，在这种情况下，即使线程仍在执行，垃圾收集器也可以启动。这样做不会出现问题是因为非托管代码不会访问未被固定(pinned)的托管对象。而一个固定对象的内存不允许被垃圾收集器任意搬移。如果一个正在执行非托管代码的线程返回到了托管代码中，那么该线程将被劫持，然后挂起，直到垃圾收集完成。

除了上面提到的一些机制(代龄、安全点、劫持)外，垃圾收集器还提供了几种额外的机制来提高对象分配和垃圾收集的性能。

19.9.1 省却同步控制的多线程分配

在一个运行着工作站版本执行引擎(MSCorWks.dll)或者服务器版本执行引擎(MSCorSvr.dll)的多处理器系统上，托管堆中的第 0 代对象会被划分到多个内存区域中，每个线程会有一个独立的区域。这使得多个线程可以同时分配内存，从而避免了托管堆的独占访问。

19.9.2 可扩展并行收集

在一个运行着服务器版本执行引擎(MSCorSvr.dll)的多处理器系统上，托管堆会被划分成几个区间，每个 CPU 会有一个独立的区间。当垃圾收集初始启动时，垃圾收集器将在每个 CPU 上有一个线程，各个线程只负责自己区间中对象的收集工作。并行垃圾收集对于服务器应用程序有着很出色的表现，因为各个工作线程的行为将趋向一致。工作站版本的执行引擎(MSCorWks.dll)不支持此项特性。

19.9.3 并发收集

在一个运行着工作站版本执行引擎(MSCorWks.dll)的多处理器系统上，垃圾收集器有一个额外的背景线程可以在应用程序运行时并发地执行垃圾收集。当一个线程因为分配新对象而使第0代对象的内存空间超过预设的阙值时，垃圾收集器会首先挂起所有线程，然后判定需要收集哪些代的对象。如果垃圾收集器需要收集第0代或者第1代的对象，那么它的处理方式和往常一样。但是，如果垃圾收集器需要收集的是第2代对象，它会放弃本应做的收集工作而继续分配新对象(换一个角度来看，也可以说第0代对象的阙值容量增加了)，应用程序的线程则会被允许继续运行。

当应用程序线程运行时，垃圾收集器有一个处于正常优先级的背景线程会去构造不可达对象图。该线程将和应用程序线程竞争CPU的时间，这会导致应用程序的执行变慢。但是，由于并发垃圾收集只运行在多处理器系统上，所以我们不会看到太大的性能损失。一旦不可达对象图构造完毕，垃圾收集器将挂起所有线程，并判断是否压缩内存。如果垃圾收集器判断需要压缩内存，那么托管堆内存将被压缩，根引用也将被修正，然后应用程序线程被允许继续运行——这样的垃圾收集花费的时间要比通常的少，因为不可达对象图已经构造好了。但是，垃圾收集器也可能决定不压缩内存。实际上，垃圾收集器更喜欢后一种方式。如果我们的系统有很多空闲内存，那么垃圾收集器将不会压缩托管堆，这会在某种程度上提高应用程序的性能，虽然这会导致应用程序进程工作集的增长。当使用并发垃圾收集时，大家一般会发现应用程序消耗的内存要比使用非并发垃圾收集时的多。

并发垃圾收集会为用户带来更好的交互体验，对于交互性的CUI和GUI应用程序来说是最好的选择。但是，对于某些应用程序，并发垃圾收集可能会损伤应用程序的性能，导致使用过多的内存。在测试应用程序时，我们应该使用并发垃圾收集和非并发垃圾收集两种方案分别进行测试，然后根据测试结果选择较好的方案。

我们可以通过创建一个包含gcConcurrent元素的配置文件(本书第2章和第3章曾对配置文件有过描述)来告诉CLR使用并发垃圾收集。下面是一个配置文件的示例：

```
<configuration>
   <runtime>
      <gcConcurrent enabled="false"/>
   </runtime>
</configuration>
```

我们也可以使用 Microsoft .NET Framework Configuration 管理工具来创建一个包含 gcConcurrent 元素的应用程序配置文件。首先打开【控制面板】，选择【管理工具】，然后调用 Microsoft .NET Framework Configuration 工具。在左边树状面板中的【应用程序】节点上，添加一个应用程序或者选择一个现存的应用程序。然后在应用程序上右击并选择【属性】，将出现如图 19.15 所示的对话框。

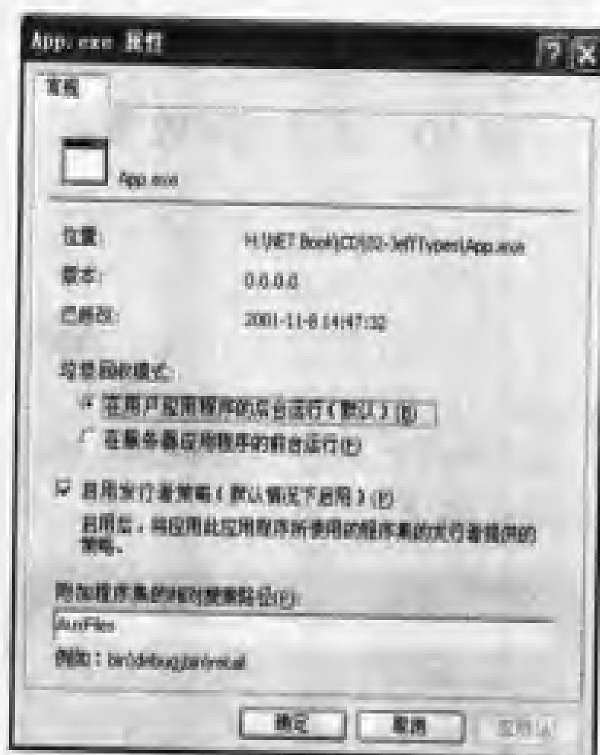


图 19.15 使用 Microsoft .NET Framework Configuration 管理工具为应用程序配置并发垃圾收集

点击【垃圾回收模式】下的单选按钮即可将 gcConcurrent 元素的 enabled 属性设为 true 或 false。

19.9.4 大尺寸对象

垃圾收集器还有一种提升性能的机制，即针对大尺寸对象的特殊收集策略。任何占用内存等于或超过 85 000 字节的对象都被认为是大尺寸对象(large object)。大尺寸对象从一个特殊的大尺寸对象托管堆中分配。该托管堆中对象的终止化和内存释放行为与前面描述的小对象相同。但是，大尺寸对象不会被压缩，因为在托管堆中搬移 85 000 字节以上的内存块会浪费 CPU 比较多的时间。

另外，大尺寸对象总被认为是第 2 代对象，因此我们只应该为那些需要保存很长时间的资源创建大尺寸对象。分配生存期比较短的大尺寸对象将导致垃圾收集器频繁地收集第 2 代对象，这无疑会损伤应用程序的性能。

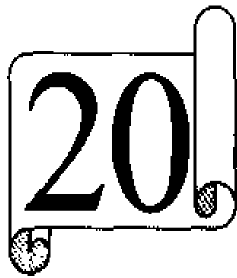
所有这些机制对于我们的应用程序代码都是透明的。对于应用程序开发人员来说，系统看起来就好像只有一个托管堆。这些机制存在的惟一目的就是提高应用程序的性能。

19.10 监视垃圾收集

当我们安装.NET框架时，它会安装一组性能计数器为CLR的一些操作提供许多实时的统计数据。这些统计数据可以通过和Windows一起发布的PerfMon.exe工具或者System Monitor ActiveX控件看到。访问System Monitor控件最简单的方式就是运行PerfMon.exe，然后选择+工具栏按钮，我们将会看到如图19.16所示的【添加计数器】对话框。

图 19.16 PerfMon.exe 中显示的.NET CLR 内存计数器

要监视CLR的垃圾收集器，首先选择.NET CLR Memory性能对象，然后再从实例列表框中选择一个应用程序。最后，选择我们感兴趣的计数器集合，并单击【添加】按钮。若想了解某一特定计数器的含义，大家可以选择该计数器，并单击【说明】按钮。关闭该对话框后，【系统监视器】会图示化我们所选择的实时统计数据。



CLR 寄宿、应用程序域、反射

本章探讨 CLR 寄宿(CLR hosting)、应用程序域(AppDomain)和反射(Reflection)共三个技术主题，这些技术为.NET 框架提供了不可估量的应用价值。微软公司许多现存的应用程序和服务器产品都有在将来寄宿 CLR 的打算。我们将会看到我们今天在学习.NET 框架上的投资会随着时间的推移而获得更多的回报。应用程序域是 CLR 提供的一种旨在减少内存使用、提高系统性能的新型机制。而反射使得我们可以很容易地使用自己或者第三方开发的类型来增强应用程序的功能，从而帮助我们设计出可动态扩展的应用程序。

20.1 元数据：.NET 框架的基石

到目前为止，我们应该已经很清楚元数据在.NET 框架开发平台中占据的重要地位了。元数据描述了一个类型的字段和方法。元数据使得一种语言开发的类型可以被另一种完全不同的语言所使用。垃圾收集器也使用元数据来确定哪些对象是可达的——元数据指出了一个对象中引用了哪些对象。一些开发工具，如微软的 Visual Studio 编辑器，也通过利用元数据来提供智能感知(IntelliSense)特性以及其他一些和帮助有关的功能。另外，元数据还用来帮助执行对象序列化和反序列化操作，从而允许将它们存储在磁盘中或者在网络上传送。实际上，正是由于利用了元数据提供的方便的对象序列化和反序列化功能，使用.NET 框架构建 XML Web 服务才变得如同儿戏一般简单。

综观全书，大家会发现我们大量使用了.NET框架SDK中附带的ILDasm.exe这个工具。该工具可以分析一个托管模块或者程序集中的元数据，然后将其中的信息以可读的格式显示出来。查看元数据的行为就是一个反射的过程，换句话说ILDasm.exe所做的事情就是对一个模块或者程序集中的元数据执行反射，然后将结果显示给用户。

反射对于开发人员来说是一个非常强大的工具。反射允许我们构建可动态扩展的应用程序。例如，任何开发人员都可以开发一个类型，并将其打包到一个程序集中。如果该类型遵循某些规则，那么Visual Studio .NET将能够把该类型(组件)集成到Windows窗体设计器或Web窗体设计器中。Visual Studio .NET可以将该类型添加到【工具箱】窗口中，当我们将类型的一个实例拖到一个窗体上时，该类型的属性将会显示在【属性】窗口中。这种丰富的集成能力和易用性是早期的Win32和COM技术所无法比拟的。本章将向大家演示怎样使用反射来实现这种集成。

另外，一个方法可以使用反射从其他代码中获取信息来改变自己的行为。我们在本书第13章中曾经看到过这样的例子。如果一个枚举类型上应用了一个System.FlagAttribute实例，那么在该枚举实例上调用ToString方法将产生一个表示位标记的值(而非一个数值类型的值)。实际上，定制特性本身就是应用反射的成果。

一个方法还可以使用反射来根据调用者的信息改变自己的行为。例如，我们有可能实现一个方法，当它被同一个程序集中的代码调用时执行一种操作，而当它被别的程序集中的代码调用时则执行另外一种操作。当然，可以选择的策略还有很多。

在深入探讨反射之前，我们首先需要熟悉CLR寄宿和应用程序域这两个机制。本章首先介绍这两个主题，然后再深入反射，并解释怎样将这些机制结合在一起使用。

20.2 CLR寄宿

我们知道目前的.NET框架是运行在微软的Windows平台之上的。这意味着.NET框架必须使用Windows可以理解的技术来构造。这首先就体现在所有的托管模块和程序集文件都必须使用Windows PE文件格式，其或者为一个Windows EXE文件，或者为一个动态连接库(DLL)文件。

在开发CLR时，微软实际上是将其作为一个COM服务器实现在了一个DLL内。也就是说，微软为CLR定义了一个标准的COM接口，并且为该接口和COM服务器分配了GUID。

当我们安装.NET 框架时，表示 CLR 的 COM 服务器就象其他的 COM 服务器一样被注册到了 Windows 的注册表里。如果大家希望了解有关这方面的更多信息，可以参考和.NET 框架 SDK 一起发布的 C++ 头文件 MSCorEE.h。该头文件中包含了一些 GUID 和非托管的 ICorRuntimeHost 接口定义。

任何 Windows 应用程序都可以寄宿 CLR。但是，我们不应该通过调用 CoCreateInstance 来创建 CLR COM 服务器实例。相反，我们的非托管宿主应该调用 CorBindToRuntimeEx 函数(其原型定义在 MSCorEE.h 中)。CorBindToRuntimeEx 函数实现在 MSCorEE.dll(通常位于 C:\Windows\System32 目录下)中。该 DLL 被称作“垫片”(shim)，它的职责是判断创建何种版本的 CLR，其本身并不包括 CLR COM 服务器。

我们知道 1.0 版本的.NET 框架包含有两个版本的 CLR COM 服务器。MSCorWks.dll 文件包含的是工作站版本的 CLR COM 服务器，该版本在单处理器、工作站环境下有较好的性能。MSCorSvr.dll 文件包含的是服务器版本的 CLR COM 服务器，它在多处理器、服务器环境下有较好的性能。微软正在开发一些新版的 CLR，它们在将来也会被安装到用户的硬盘上。

当调用 CorBindToRuntimeEx 函数时，非托管宿主可以传入参数来指定创建哪个版本的 CLR。其中的版本信息指出了是工作站版本、还是服务器版本，以及 CLR 的版本号。CorBindToRuntimeEx 函数在判断到底加载哪个版本的 CLR 时，它除了使用宿主指定的版本信息外，还会搜集另外一些自己需要的信息(例如机器中安装了多少个 CPU，以及安装了哪些版本的 CLR)——也就是说，MSCorEE.dll 并不一定会加载宿主所指定的那个版本。

默认情况下，MSCorEE.dll 会查看托管 EXE 文件，并提取当初生成和测试应用程序时使用的 CLR 的版本信息。但是，我们也可以通过在应用程序的 XML 配置文件(本书第 2 章和第 3 章对此有描述)中插入一些条目来覆盖这一默认设置。下面就是一个 XML 配置文件的例子，它为我们演示了怎样告诉 MSCorEE.dll 去加载某个特定版本的 CLR。

```
<configuration>
   <startup>
      <requiredRuntime version="v1.0.0.0" safemode="true"/>
   </startup>
</configuration>
```

在查看完上面的信息之后，MSCorEE.dll 会加载相应版本的 CLR。如果 safemode 被设置为 false(默认设置)，那么 MSCorEE.dll 将加载最近安装的、与指定的版本相兼容的那个版本的 CLR。

微软通过创建一些注册表设置来指定哪些版本的 CLR 和另外一些版本相互兼容。我们可以通过查看下面注册表子键下的值来得到微软设定的 CLR 版本策略：

```
HKEY_LOCAL_MACHINE\Software\Microsoft\.NETFramework\Policy
```

我们永远都不能去修改该注册表子键下的任何值。当我们安装新版的.NET 框架时，安装程序会根据微软的兼容性测试来修改这些设置。

CorBindToRuntimeEx 函数返回的是一个指向非托管 ICorRuntimeHost 接口的指针。宿主应用程序可以通过调用该接口定义的方法来初始化 CLR。宿主还可以调用其中的方法告诉 CLR 加载哪个程序集，以及从哪个方法开始执行。本书第 1 章 1.3“加载通用语言运行时”一节对 CLR 在托管控制台应用程序和 Windows 窗体应用程序下的工作机制有详细的解释。

20.3 应用程序域

当 CLR COM 服务器被加载到一个 Windows 进程中时，它便开始执行初始化。初始化的部分工作就是创建一个托管堆，该托管堆将用于分配所有的引用对象、以及对它们执行垃圾收集。另外，CLR 还会创建一个可被加载到当前进程中所有托管类型使用的线程池。在进行这些初始化工作的同时，CLR 还会创建一个**应用程序域**(AppDomain)。一个应用程序域是一组程序集的一个逻辑容器。CLR 初始化时创建的第一个应用程序域称作**默认应用程序域**(default AppDomain)，该应用程序域只有在 Windows 进程中断时才会被销毁。

注意 在第 1 版的.NET 框架中，一个 Windows 进程中最多只能存在一个 CLR COM 服务器对象，并且只有在宿主进程中断时，该 CLR COM 服务器对象才会被销毁。也就是说，假如 CorBindToRuntimeEx 函数创建了一个 CLR COM 服务器，并返回一个指向 ICorRuntimeHost 接口的指针，那么在该接口上调用 AddRef 和 Release 方法将没有任何效果。另外，由于一个宿主进程只能创建一个 CLR COM 服务器的实例，所以如果一个宿主进程多次调用 CorBindToRuntimeEx 函数，那么每次返回的都将是同一个 ICorRuntimeHost 指针。

除了默认的应用程序域外，一个宿主还可以指示 CLR 创建额外的应用程序域。另外，托管程序集中的代码也可以告诉 CLR 创建额外的应用程序域。应用程序域有三个非常有用的特点：

- **应用程序域之间是相互隔离的** 一个应用程序域看不到另一个应用程序域中的对象。这种清晰的隔离是强制性的，因为一个应用程序域中的代码不能直接引用另一个应用程序域中创建的对象。这种隔离策略使得应用程序域很容易从一个进程中被卸载下来。

- **应用程序域可以被卸载** CLR 不支持单个程序集的卸载。但是，我们可以告诉 CLR 卸载一个应用程序域及其内包含的所有程序集。

- **应用程序域可以单独实施安全策略和配置策略** 当一个应用程序域被创建时，它将会与特定的证据(evidence)联系在一起。证据是一种与安全相关的特性(feature)，它决定了运行在应用程序域中的程序集所享有的最大权力。通常我们可以使用 AppDomain 的 SetAppDomainPolicy 方法来在应用程序域上应用某种安全策略。另外，System.AppDomainSetup 类也允许我们设置或者查询一个应用程序域的配置策略。这些配置规定了 CLR 怎样定位和加载程序集，它们包括以下内容：

  - **ApplicationName** 一个用户友好的字符串名称，该名称用来标识一个应用程序域。

  - **ApplicationBase** 一个目录，CLR 使用该目录来定位程序集。

  - **PrivateBinPath** 一个目录集合，CLR 使用该目录集合来定位弱命名程序集。

  - **ConfigurationFile** 一个配置文件的路径名，该配置文件中包括了 CLR 用来定位程序集的规则以及远程(remoting)设置、Web 应用程序设置等。

  - **LoaderOptimization** 一个标记，该标记用来告诉 CLR 将加载的程序集是以中立域(domain-neutral)、还是以独立域(single-domain)的方式来对待。

重要 在一个进程中运行多个非托管应用程序是很危险的，因为不同的应用程序将能访问彼此的数据和代码，从而使得一个应用程序可以很容易破坏另外一个应用程序。但托管代码却不存在这样的问题，因为托管 IL 代码是类型安全的，并且也是经过验证的，这使得一个应用程序域中的代码不可能破坏另一个应用程序域中的代码。当然，管理员可以关闭验证过程，从而允许托管代码调用非托管函数。如果管理员这么做，那么以上所有的担保将全部失效，应用程序域的崩溃也是完全有可能的。

图 20.1 演示了一个单独的 Windows 进程，其中运行有一个 CLR COM 服务器。该 CLR 管理着两个应用程序域。每个应用程序域都有自己的加载器堆(loader heap)，其中维护着自应用程序域创建以来被访问过的类型记录。加载器堆中的每个类型都有一个方法表，对于方法表中的每个条目，如果其中的方法至少执行过一次，那么它将指向被 JIT 编译后得到的 x86 代码。

另外，每个应用程序域还有一些已经加载入其中的程序集。AppDomain #1(默认的应用程序域)有三个程序集：MyApp.exe、TypeLib.dll 和 System.dll。TypeLib.dll 程序集中包括三个模块：TypeLib.dll(其中包括着程序集清单)、FUT.netmodule 和 RUT.netmodule。AppDomain #2 中加载有三个单模块的程序集：Wintellect.dll、System.dll 与 Microsoft.dll。

默认情况下，一个程序集会被加载到每个应用程序域中。例如，AppDomain #1 和 AppDomain #2 中都加载了 System.dll 程序集。这意味着 System.dll 程序集的相关信息会在每一个应用程序域的加载器堆中构建一次。甚至 System.dll 中类型定义的一些方法被 JIT 编译后的代码也会在该 Windows 进程的地址空间中存在两份。这样做的优点是一个应用程序域可以完全被从进程中卸载下来，而不影响其他的应用程序域。

但有时候我们也有一些程序集是期望被几个应用程序域所共用的。最典型的例子就是 MSCorLib.dll，这是微软创建的一个单模块程序集。该程序集中包括了 System.Object、System.Int32、以及所有其他对.NET 框架来说所必须的类型。CLR 初始化的时候，该程序集会被自动加载，所有的应用程序域会共享该程序集中的类型。为了减少资源使用，MSCorLib.dll 程序集会以一种中立域的方式被加载。对于以中立域方式加载的程序集来说，CLR 会为它们维护一个特殊的加载器堆。以中立域方式加载的程序集只有在进程中断时才会被卸载。

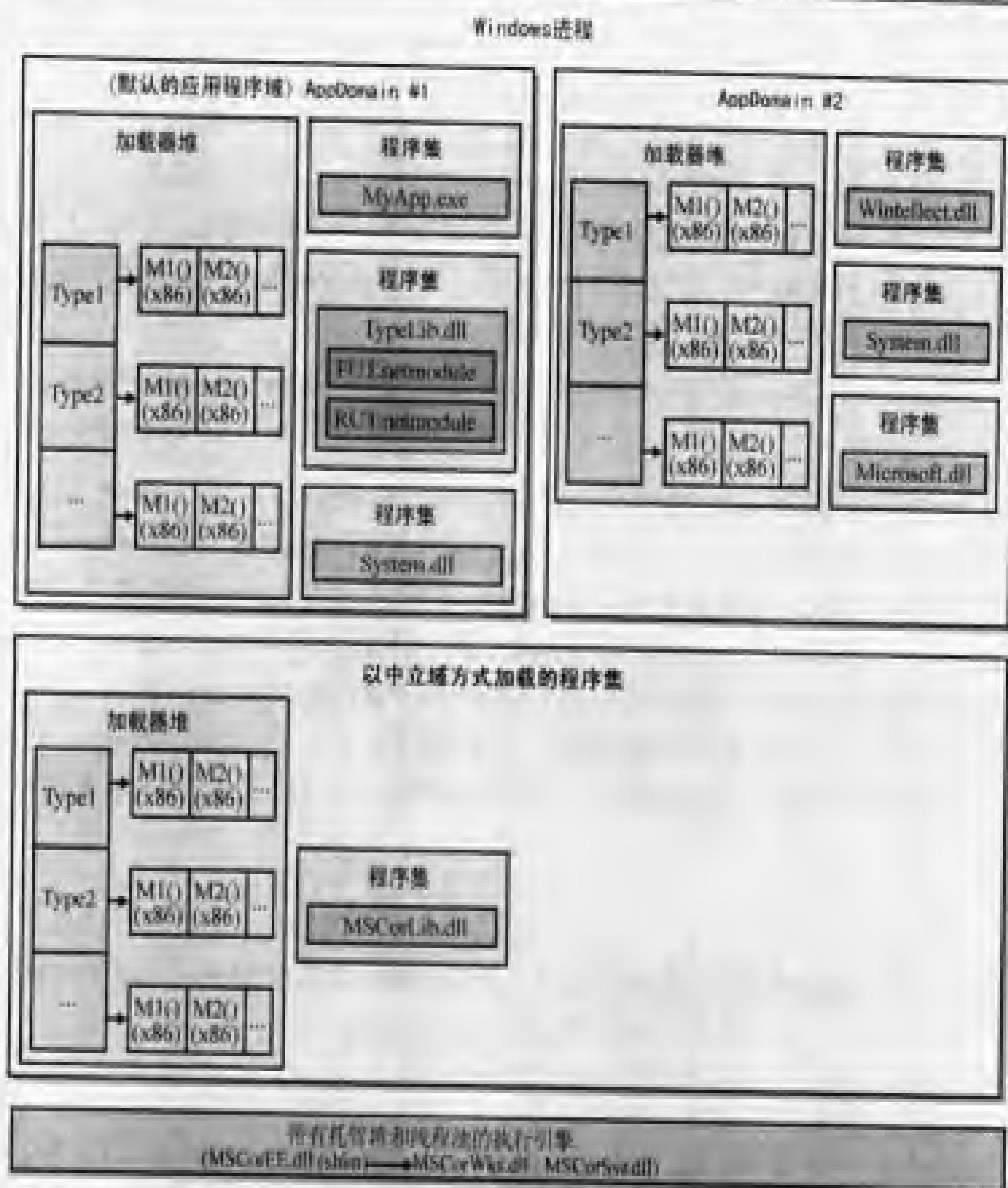


图 20.1　一个寄宿了 CLR 和两个应用程序域的 Windows 进程

20.3.1　跨越应用程序域边界访问对象

一个应用程序域中的代码可以和另一个应用程序域中的类型和对象相互通信。但是，这样的通信必须通过一种预先定义的机制进行。大多数类型在跨越应用程序域边界时是通过传值的方式(by value)来进行封送处理(marshal)的。换句话说，如果我们在一个应用程序域中构造了一个对象，然后又将该对象的引用传递给了另一个应用程序域，那么 CLR 必须首先将该对象的字段序列化到一个内存块中，然后再将该内存块传递给另一个应用程序域，最后再执行反序列化得到新的对象。

目的应用程序域将使用这个新创建的对象引用，它不会访问原来应用程序域中的对象。对于以传值方式进行远程传送(remote)的对象来说，对象的类型必须应用有 System.Serializable 定制特性。

> **注意** 如果需要的话，反序列化对象会导致 CLR 加载相关类型所在的程序集。如果 CLR 不能使用目的应用程序域的策略(例如该应用程序域可能有一个不同的 AppBase 目录)来定位这些程序集，对象将不能被反序列化，并且将会有异常抛出。

除了应用 System.Serializable 定制特性的类型外，继承自 System.MarshalByRefObject 的类型也可以为对象提供跨越应用程序域边界的访问能力。但是，这样的访问是通过传引用(by reference)而非传值(by value)来进行的。假设我们在一个应用程序域中创建了一个对象(其类型继承自 System.MarshalByRefObject)。当该对象的引用被传递给一个目的应用程序域时，CLR 实际上会在目的应用程序域中创建一个代理类型的实例，目的应用程序域中的代码将使用这个代理对象引用。原来的对象及其字段仍然驻留在原来的应用程序域中。代理对象实际上是一个封装器(wrapper)，它知道怎样调用原来应用程序域中的对象上的实例方法。同样，目的应用程序域不会直接访问原来应用程序域中的对象。

很明显，跨越应用程序域边界访问对象会有一些性能损耗。我们应该尽可能地避免这样的操作。

线程和应用程序域之间不存在一对一的关系。当一个应用程序域中的线程调用另一个应用程序域中的方法时，线程会在两个应用程序域间跳转。这意味着跨越应用程序域边界的方法调用会被同步地执行。但是，在任何一个给定的时刻，一个线程都被认为只存在于一个应用程序域中。我们可以调用 System.Threading.Thread 的静态方法 GetDomain 来获得当前执行线程所在的应用程序域(由一个 System.AppDomain 对象所标识)。

当一个应用程序域被卸载时，CLR 会知道哪些线程位于该应用程序域中，并在这些线程中强制产生一个 ThreadAbortException 异常，使它们退出该应用程序域。一旦这些线程离开该应用程序域，CLR 会使所有指向要卸载的应用程序域中的对象的那些代理对象变得无效。在这之后，所有对无效代理对象的方法的调用都将导致一个 System.AppDomainUnloadedException 异常被抛出，因为原来的对象已经不存在了。

本章 20.9"显式卸载程序集：卸载应用程序域"一节中讨论的 AppDomainRunner 示例应用程序将向大家演示怎样以传引用的方式进行跨越应用程序域边界的对象封送处理。

20.3.2　应用程序域事件

表 20.1 简要描述了一个应用程序域提供的几个非常有用的事件：

表 20.1　AppDomain 事件

事件名称	描　述
AssemblyLoad	该事件在每次 CLR 将一个程序集加载到应用程序域中时被触发。事件处理器接受一个标识被加载程序集的 System.Reflection.Assembly 对象
DomainUnload	该事件在应用程序域卸载之前被触发。如果包含应用程序域的进程发生中断，该事件将不会被触发
ProcessExit	该事件在进程中断之前被触发。该事件只会为默认的应用程序域触发，任何其他登记该事件的应用程序域将不会接到事件通知
UnhandledException	该事件在一个应用程序域中出现未处理异常时被触发。该事件在本书第 18 章中有过讨论
AssemblyResolve	该事件在 CLR 不能定位应用程序域所需要的程序集时被触发。事件处理器接受一个标识缺失程序集名称的字符串
ResourceResolve	该事件在 CLR 不能定位应用程序域所需要的资源时被触发。事件处理器接受一个标识缺失资源名称的字符串
TypeResolve	该事件在 CLR 不能定位应用程序域中某个程序集所需要的类型时被触发。事件处理器接受一个标识缺失类型名称的字符串，并且可以通过返回一个 Type 对象引用来告诉 CLR 要使用的类型。通常，处理器会根据客户端的位置或者操作系统来确定返回什么样的 Type 对象

20.3.3 应用程序及其如何寄宿 CLR 和管理应用程序域

到目前为止，我们已经谈论了 CLR 宿主以及它们加载 CLR 和应用程序域的方式，另外我们还探讨了宿主是如何告诉 CLR 来创建和卸载应用程序域的。为了更具体地说明问题，下面为大家描述一些常见的 CLR 寄宿和应用程序域的应用场景。大家将会了解到不同类型的应用程序是如何寄宿 CLR 以及管理应用程序域的。

控制台应用程序和 Windows 窗体应用程序

当我们调用一个托管控制台应用程序、或者 Windows 窗体应用程序时，被称作“垫片”(shim)的 MSCorEE.dll 会检查应用程序的程序集中包含的 CLR 表头信息。该表头信息指出了生成与测试应用程序时所使用的 CLR 的版本。MSCorEE.dll 使用该信息来判断创建哪个 CLR COM 服务器。在 CLR 加载并初始化之后，它会再次检查程序集的 CLR 表头来判断应用程序的入口点方法(Main)。接着，CLR 调用该方法，应用程序便开始运行。

随着代码的运行，它会访问其他类型。当引用到包含在其他程序集中的类型时，CLR 会定位必要的程序集、并将其加载到同一个应用程序域中。任何另外间接被引用的程序集也会被加载到同一个应用程序域中。当应用程序的 Main 方法返回时，默认的应用程序域将被卸载，Windows 进程也随之中断。

> **注意** 顺便提一句，如果我们希望关闭 Windows 进程(包括其中所有的应用程序域)，我们可以调用 System.Environment 的静态方法 Exit。Exit 是中断一个进程的最佳方式，因为它首先会调用托管堆中所有对象的 Finalize 方法，然后还会释放所有由 CLR 维护的非托管 COM 对象。最后，Exit 将调用 Win32 函数 ExitProcess。

除了默认的应用程序域外，控制台应用程序或 Windows 窗体应用程序还可以告诉 CLR 在进程的地址空间上创建额外的应用程序域。但是，这两类应用程序很少使用或者需要多个应用程序域。

ASP.NET Web 窗体和 XML Web 服务应用程序

ASP.NET 是一个 ISAPI DLL(实现于 ASPNet_ISAPI.dll 之中)。当客户端请求一个由 ASP.NET ISAPI DLL 处理的 URL 时，ASP.NET 会创建一个工作者进程(worker process)(ASPNet_wp.exe)。工作者进程是一个寄宿有 CLR COM 服务器的 Windows 进程。

当客户端向一个 Web 应用程序发出一个请求时，ASP.NET 会判断该请求是否为第一次。如果是，ASP.NET 会告诉 CLR 为该 Web 应用程序创建一个新的应用程序域(每个 Web 应用程序可以由它的虚拟根目录来判定)。ASP.NET 然后告诉 CLR 将必要的程序集(其中包含着组成该 Web 应用程序的类型)加载到新建的应用程序域中，并创建一个 Web 应用程序类型的实例，调用其中的方法响应客户端请求。随着 Web 应用程序的运行，如果代码中引用到了更多的类型，CLR 会继续将必要的程序集加载到当前 Web 应用程序的应用程序域中。顺便提一句，强命名程序集(例如 System.Web.dll)是以中立域的方式来加载的，因为这样做可以节省操作系统的资源。

当更多的客户端向一个已经运行的 Web 应用程序发出请求时，ASP.NET 不会再去创建新的应用程序域，相反，它会使用现有的应用程序域，创建一个新的 Web 应用程序类型的实例，并调用其中的方法。这些方法应该已经被 JIT 编译成了本地代码，所以后续客户端请求的处理性能将会有所提高。

如果客户端请求的是另外一个不同的 Web 应用程序，ASP.NET 将告诉 CLR 创建一个新的应用程序域。这个新创建的应用程序域典型地和其他的应用程序域处于同一个工作者进程中。这意味着许多 Web 应用程序将运行在同一个 Windows 进程中，这样做将有助于提高整个系统的效率。当然，不同的 Web 应用程序所需的程序集会被加载到自己的应用程序域中。

Microsoft Internet Explorer

当我们安装.NET 框架时，它会安装一个 MIME 筛选器(MSCorIE.dll)挂接在 5.01 版本及其以上的 IE 浏览器内。该 MIME 筛选器会处理标识有 MIME 类型为“application/octet-stream”或“application/x-msdownload”的下载内容。当该 MIME 筛选器检测到有一个托管程序集被下载时，它会调用 CorBindToRuntimeEx 函数来创建一个 CLR COM 服务器。这使得 IE 浏览器的进程也成为一个 CLR 宿主。

该 MIME 筛选器由 CLR 控制，它可以确保来自同一个 Web 站点的所有程序集都被加载到属于自己的应用程序域中。这使得计算机管理员可以以不同的方式来对待来自不同站点的程序集(例如信任来自 A 站点的程序集，而不信任来自 B 站点的程序集)。这还使得 IE 浏览器可以在用户浏览不同的 Web 应用程序时卸载前一个 Web 应用程序所使用的程序集。

20.3.4 Yukon

SQL Server 的下一个版本(代号 Yukon)仍是一个非托管应用程序，因为它的大多数代码仍然采用非托管 C++开发。但是，Yukon 会在初始化的时候创建一个 CLR COM 服务器。这使得我们可以在 Yukon 中用托管语言(C#、Visual BasicSmalltalk 等)来编写存储过程。

由于这些存储过程运行在应用了特定证据(evidence)的应用程序域中，因此便可以保证它们不会恶意损坏数据库服务器。当执行这些用托管语言编写的存储过程时，Yukon 将告诉 CLR 加载必要的程序集并调用其中的方法，根据相关的证据，这些方法将在一定的安全限制下运行。

这种功能相当厉害！这意味着我们将可以用自己喜欢的语言来编写存储过程，我们将可以在存储过程代码中使用强类型数据对象，而且编写的存储过程代码将以 JIT 编译的方式执行(而非目前的解释执行)。最后，我们还可以在编写存储过程的时候求助于.NET 框架类库(FCL)或者其他程序集中定义的任何类型。显然这将极大地减轻我们编写存储过程的工作，而且最后得到的应用程序性能也会更高。到了这一步，一个开发人员还有什么可求呢？

随着时间的推移，象字处理、电子表格这样的软件产品也将会允许我们选择自己喜欢的语言来编写宏(macro)。这些宏也将可以访问所有 CLR 支持的程序集和类型。它们同样将以 JIT 编译的方式执行，效率自然也会更高。更重要的是，这些宏将运行在一个安全的应用程序域中，因此也可以避免一些恶意的攻击。

20.4 反射概要

我们知道，元数据本质上就是一大堆的表。当我们生成一个程序集或者模块时，编译器会创建一个类型定义表、一个字段定义表、一个方法定义表、等等。FCL 中的 System.Reflection 命名空间包含的一些类型允许我们编写代码来反射(或者说分析)这些元数据表。实际上，System.Reflection 命名空间中的类型为包含在一个程序集或者模块中的元数据提供了一个良好的对象模型。

利用该对象模型中的类型，我们可以很容易地枚举出一个类型定义元数据表中包括的所有类型。对于每一个类型，我们又可以获取它的基类型、它实现的接口、以及与其相关联的一些标记。另外，利用 System.Reflection 命名空间中的一些类型，我们还可以通过分析相关的元数据表来查询一个类型的字段、方法、属性、以及事件。我们还可以查找应用在任何元数据实体上的任何定制特性(有关定制特性的内容可参见本书第 16 章)。利用所有这些信息，我们甚至能够创建出一个类似于 ILDasm.exe 的工具来。

重要　System.Reflection 命名空间中的类型允许我们查询定义元数据表，但是却没有提供任何类型来供我们查询引用元数据表，也没有提供任何反射类型来供我们读取一个方法的 IL 代码。ILDasm.exe 工具是通过直接分析文件的字节来获得这些信息的。幸运的是，托管模块和程序集的文件格式是公开的。实际上，该文件格式已经被递交给 ECMA 技术委员会正在进行标准化工作。所以如果我们编写的应用程序希望查询引用元数据表、或者获取一个托管方法的 IL 指令字节，我们必须自己编写代码来分析文件的格式，因为反射没有为我们提供这种能力。

另外，也有一些定义元数据表信息目前还不能通过反射来获得。例如，我们目前还无法确定可选参数的默认值(Visual Basic 提供的一个特性，C#没有提供)。微软将在.NET 框架的后续版本中修复这些反射类型中的漏洞。

最后，我们应该清楚某些反射类型以及它们提供的一些成员是为那些面向 CLR 的编译器开发人员而专门设计的。应用程序开发人员一般不会使用这些类型和成员．.NET 框架文档没有明确指出哪些类型和成员是专门用于编译器开发人员而非应用程序开发人员的，但是如果我们知道并非所有的反射类型和成员都适用于每个人，那么读起文档来就会减少一些困惑。

事实上，需要使用反射类型的应用程序是比较少的。反射类型典型地应用于一些类库，这些类库通常需要理解一个类型的定义，以便提供某些丰富的功能。例如，FCL 的序列化机制就是使用反射来确定一个类型中定义了哪些字段。序列化格式器然后会获取这些字段的值，并将它们写入一个字节流以供在网络上发送。类似地，Visual Studio 设计器在设计时也使用反射来确定当控件被放在它们的 Web 窗体或 Windows 窗体上时，哪些属性应该被显示给开发人员。

当应用程序需要在运行时加载某个特定的类型或程序集时，反射也显得十分有用。例如，一个应用程序可能会要求用户提供一个程序集和类型的名称，然后就可以显式地加载程序集、构造类型实例、并调用类型上的方法。这种用法在概念上类似于调用 Win32 的 LoadLibrary 和 GetProcAddress 函数。先绑定类型、然后调用方法的方式通常被称作**晚绑定**(late binding)。而**早绑定**(early binding)指一个应用程序使用的类型和方法在编译时就已确定。

本章剩余部分将包含许多示例应用程序，其中都使用了反射机制。每个应用程序都演示了一种使用反射的不同方式，并就如何高效地使用反射这样的话题进行了探讨。

20.5 反射一个程序集中的类型

反射经常被用于判断一个程序集中定义了哪些类型。下面的 Reflector 示例代码(可以从 http://www.Wintellect.com/下载)演示了一种做法：

```
using System;
using System.Reflection;

class App {
    static void Main() {
        Assembly assem = Assembly.GetExecutingAssembly();
        Reflector.ReflectOnAssembly(assem);
    }
}

public class Reflector {
    public static void ReflectOnAssembly(Assembly assem) {
        WriteLine(0, "Assembly: {0}", assem);

        // 查找模块
        foreach (Module m in assem.GetModules()) {
            WriteLine(1, "Module: {0}", m);

            // 查找类型
            foreach (Type t in m.GetTypes()) {
                WriteLine(2, "Type: {0}", t);

                // 查找成员
                foreach (MemberInfo mi in t.GetMembers())
                    WriteLine(3, "{0}: {1}", mi.MemberType, mi);
            }
        }
    }

    private static void WriteLine(Int32 indent, String format,
        params Object[] args) {
```

```
        Console.WriteLine(new String(' ', 3 * indent) + format, args);
    }
}

class SomeType {
    public class InnerType {}
    public Int32 SomeField = 0;
    private static String goo = null;

    private void SomeMethod() { }
    private TimeSpan SomeProperty {
        get { return new TimeSpan(); }
        set { }
    }

    public static event System.Threading.ThreadStart SomeEvent;

    private void NoCompilerWarnings() {
        // 这里的代码仅仅是为了消除编译器的警告信息
        SomeEvent.ToString();
        goo.ToString();
    }
}
```

在上面的 Main 方法中，我们首先调用了 System.Reflection.Assembly 的静态方法 GetExecutingAssembly。该方法判定正在调用的方法所处的程序集，并返回一个 Assembly 对象引用。然后该引用被传递给 Reflector 类的静态方法 ReflectOnAssembly。ReflectOnAssembly 方法首先显示程序集的全名，然后调用 GetModules 方法，返回一个 System.Reflection.Module 类的数组。数组中的每个元素表示程序集中包含的一个模块。

随后我们用一个循环来遍历程序集中的每个模块。在每次循环中，首先显示模块的名称，然后调用 GetTypes。GetTypes 返回一个 System.Type 数组，数组中的每个元素表示模块中定义的一个类型。然后我们再用一个循环来遍历每个类型，在每次循环中显示类型的名称，然后调用 GetMembers。GetMembers 方法又返回一个 System.Reflection.MemberInfo 数组，数组中的每个元素表示一个类型的成员(构造器、方法、字段、属性、事件、或者嵌套类型)。

SomeType 是我们创建的一个演示类型，该类型定义了各种不同的成员，在反射元数据时它们将按照我们期望的形式被显示出来。编译并运行 Reflector 应用程序，我们将会得到以下输出：

```
Assembly: Reflector, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
  Module: Reflector.exe
    Type: App
      Method: Int32 GetHashCode()
      Method: Boolean Equals(System.Object)
      Method: System.String ToString()
      Method: System.Type GetType()
      Constructor: Void .ctor()
    Type: Reflector
      Method: Int32 GetHashCode()
      Method: Boolean Equals(System.Object)
      Method: System.String ToString()
      Method: Void ReflectOnAssembly(System.Reflection.Assembly)
      Method: System.Type GetType()
      Constructor: Void .ctor()
    Type: SomeType
      Field: Int32 SomeField
      Method: Int32 GetHashCode()
      Method: Boolean Equals(System.Object)
      Method: System.String ToString()
      Method: Void add_SomeEvent(System.Threading.ThreadStart)
      Method: Void remove_SomeEvent(System.Threading.ThreadStart)
      Method: System.Type GetType()
      Constructor: Void .ctor()
      Event: System.Threading.ThreadStart SomeEvent
      NestedType: SomeType+InnerType
    Type: SomeType+InnerType
      Method: Int32 GetHashCode()
      Method: Boolean Equals(System.Object)
      Method: System.String ToString()
      Method: System.Type GetType()
      Constructor: Void .ctor()
```

从上面的输出可以看到该程序集的名称为 Reflector，版本为 0.0.0.0，并且没有任何特定的语言文化相关联。另外，该程序集也没有任何的公有密钥标记，这使得它成为一个弱命名程序集(而非一个强命名程序集)。最后，该程序集只包含一个模块：Reflector.exe。模块中总共定义了四种类型：App、Reflector、SomeType 和 SomeType+InnerType(嵌套类型)。对于每一种类型，我们可以看到其内定义的成员。由于 SomeType 定义了各种不同的成员，所以它所显示的结果对我们来说是最有意义的。

大家可能已经注意到了在 Reflector 应用程序中只有公有成员才被显示了出来。我们很快将会在本章后面看到如何利用绑定标记(binding flag)来获得一个类型内定义的所有公有成员和非公有成员。

20.6 反射一个应用程序域中的程序集

Reflector是介绍反射的一个很好的例子，它向我们演示了怎样反射一个程序集中的元数据。然而，有时候我们也可能希望能够反射一个应用程序域中包含的所有程序集。为了演示怎样反射一个应用程序域中所有的程序集，我们来对Reflector例子中的Main方法做一个小小的改动。下面是改动后的Main方法(其他的仍保持不变)：

```
static void Main() {
    foreach (Assembly assem in
        AppDomain.CurrentDomain.GetAssemblies()) {

            Reflector.ReflectOnAssembly(assem);
    }
}
```

在新版的Main中，我们调用了System.AppDomain的静态属性CurrentDomain。该属性判定正在调用的代码所处的应用程序域，并返回一个AppDomain对象引用。然后，我们在该AppDomain对象上调用GetAssemblies方法，该方法返回一个System.Reflection.Assembly数组，数组元素表示GetAssemblies方法被调用之时已经被加载到应用程序域中的程序集。

随后，我们用一个循环来遍历程序集数组，并为每一个元素调用Reflector的静态方法ReflectOnAssembly。新版的Reflector应用程序将显示应用程序域中所有的程序集，每个程序集中包含的所有模块，每个模块中定义的所有类型，以及每个类型中定义的所有成员。

编译并运行新版的Reflector应用程序，我们将会在输出中看到有两个程序集：MSCorLib.dll和Refector.exe。由于MSCorlib.dll定义的类型超过1400个，所以新版的应用程序产生的输出非常之长，这里就不再将它们一一列出了。

20.7 反射一个类型的成员：绑定

在Reflector应用程序中，我们曾经通过调用Type对象的GetMembers方法来获得一个类型的成员。Type实际上提供有两个版本的GetMembers方法。第一个版本不接受任何参数，只返回类型中定义的公有静态成员和公有实例成员。第二个版本接受一个参数：一个System.Reflection.BindingFlags枚举实例。表20.2展示了BindingFlags枚举类型定义的一部分有关的符号。

表 20.2 BindingFlags 枚举类型定义的搜索符号

符 号	值	描 述
Default	0x00	一个没有指定任何标记的占位符。当我们不希望指定表中任何其他的标记时，才使用这个标记
IgnoreCase	0x01	搜索的时候忽略大小写
DeclaredOnly	0x02	只搜索类型中声明定义的成员。(忽略继承成员)
Instance	0x04	搜索实例成员
Static	0x08	搜索静态成员
Public	0x10	搜索公有成员
NonPublic	0x20	搜索非公有成员
FlattenHierarchy	0x40	搜索基类中定义的静态成员

本章前面曾经指出过 Reflector 应用程序的输出不包括类型的私有成员。现在，我们则可以通过传递期望的 BindingFlags 来告诉 GetMembers 方法应该返回何种成员。

为了反射一个类型中的非公有成员，我们要对原来的 Reflector 做一些改动。在改动后的版本中，我们希望 GetMembers 方法能够返回类型中声明的所有公有的和非公有的、静态的和实例的成员。任何从基类中继承的成员将不会被显示。下面是改动后的调用 GetMembers 方法的代码。(其余部分仍保持不变。)

```
BindingFlags bf = BindingFlags.DeclaredOnly |
   BindingFlags.NonPublic | BindingFlags.Public |
   BindingFlags.Instance | BindingFlags.Static;

foreach (MemberInfo mi in t.GetMembers(bf))
   WriteLine(3, "{0}: {1}", mi.MemberType, mi);
```

在上面的代码中，bf 被初始化为一个位标记的集合，它表示我们期望遍历一个类型中所有的公有的和非公有的、静态的和实例的成员。其中 DeclaredOnly 标记表示我们只希望遍历那些类型中定义的成员——而不包括任何从基类型中继承的成员。在 bf 初始化之后，它被传递给 GetMembers 方法，GetMembers 方法将因此而知道我们对处理哪些成员感兴趣。

编译并运行新版的 Reflector 代码，我们将会看到以下输出：

```
Assembly: Reflector, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
  Module: Reflector.exe
    Type: App
      Method: Void Main()
      Constructor: Void .ctor()
    Type: Reflector
      Method: Void ReflectOnAssembly(System.Reflection.Assembly)
      Method: Void WriteLine(Int32, System.String, System.Object[])
      Constructor: Void .ctor()
    Type: SomeType
      Field: Int32 SomeField
      Field: System.Threading.ThreadStart SomeEventEvent
      Field: System.String goo
      Method: Void add_SomeEvent(System.Threading.ThreadStart)
      Method: Void remove_SomeEvent(System.Threading.ThreadStart)
      Method: Void SomeMethod()
      Method: System.TimeSpan get_SomeProperty()
      Method: Void set_SomeProperty(System.TimeSpan)
      Constructor: Void .ctor()
      Property: System.TimeSpan SomeProperty
      Event: System.Threading.ThreadStart SomeEvent
      NestedType: SomeType+InnerType
    Type: SomeType+InnerType
      Constructor: Void .ctor()
```

20.8 显式加载程序集

到目前为止，我们已经知道如何反射被加载到一个应用程序域中的程序集了。除了这些有用的知识外，我们还必须记住CLR判定加载一个程序集的时间：在一个方法第一次被调用的时候，CLR会检查该方法的IL指令看其中引用了哪些类型，然后CLR会加载所有这些被引用类型所处的程序集。如果需要的程序集已经存在于应用程序域中，CLR不会重复加载该程序集。

但是，假设我们现在希望编写一个应用程序来统计所有实现了某个接口的类型的数量。要做到这一点，我们必须实现一个方法，并在其中引用每一个我们希望加载的程序集内至少一个类型。然后，我们调用该方法以使CLR加载所有这些程序集。一旦这些程序集被加载，我们就可以使用前面讨论过的反射方法。

然而，这不是编写此类应用程序理想的方式。相反，我们希望有一种方式能够将程序集显式加载到一个应用程序域中，然后再反射其中的类型。这种技巧类似于使用Win32中的LoadLibrary函数。

System.Reflection.Assembly 类型提供了三个静态方法允许我们显式加载一个程序集：Load、LoadFrom 和 LoadWithPartialName。(每个方法都有几个重载版本。)在这三个方法中，强烈推荐大家尽可能地使用 Load 方法，而避免使用 LoadFrom 和 LoadWithPartialName。

Load 方法通过接受一个程序集标识来加载程序集。如果是一个强命名程序集，那么程序集标识将包括程序集的名称、版本、语言文化、以及公有密钥标记。如果是一个弱命名程序集，那么程序集标识将只包括程序集的名称(不带文件的扩展名)。Load 使用的算法和 CLR 隐式加载程序集的算法相同。如果指定的是一个强命名程序集，那么 Load 将导致 CLR 在程序集上应用一定的策略，并按照"全局程序集缓存(GAC)"、"应用程序基目录"、"私有路径目录"的顺序查找程序集。如果指定的是一个弱命名程序集，那么 CLR 将不会在程序集上应用任何策略，也不会到 GAC 中查找程序集。不管指定的是强命名程序集还是弱命名程序集，如果 CLR 没能找到指定的程序集文件，Load 方法都将抛出一个 System.IO.FileNotFoundException 异常。

Assembly 的静态方法 LoadFrom 的工作原理有所不同。当调用 LoadFrom 时，我们要为其传递程序集文件完整的路径名称(包括文件扩展名)，而 CLR 将加载与指定的程序集文件完全匹配的程序集。我们为 LoadFrom 传递的字符串不能包括任何的强命名信息(包括版本、语言文化以及公有密钥标记)。CLR 不会在我们指定的文件上应用任何策略，它也不会去任何其他地方搜索程序集文件。如果指定的路径上不存在程序集文件，那么 LoadFrom 方法将抛出一个 System.IO.FileNotFoundException 异常。

显式加载程序集的最后一个方法是 LoadWithPartialName。我们应该避免使用该方法，因为应用程序并不能确切地知道要加载的是哪个版本的程序集。该方法仅用来帮助某些客户使用.NET 框架 beta 版本中提供的一些行为。由于存在一些不确定性，这些行为会在.NET 框架以后的版本中被去掉。

对于关心 LoadWithPartialName 方法的读者，下面是它的工作原理。在调用该方法时，我们要为其传递一个程序集标识，其中包含程序集的名称(不带文件扩展名)。程序集的版本、语言文化以及公有密钥标记则是可选项。当 LoadWithPartialName 执行时，CLR 首先检查应用程序的 XML 配置文件来搜索 qualifyAssembly 元素。如果该元素存在，它会告诉 CLR 怎样将一个部分的程序集标识映射为一个完全限定的程序集标识——这时CLR 将使用通常的规则来查找程序集。如果 qualifyAssembly 元素不存在，CLR 将使用指定的名称在应用程序的 AppBase 目录和私有路径目录中搜索程序集。如果找到了匹配的程序集，CLR 将加载它。

如果找不到匹配的程序集，CLR 将使用指定的程序集标识部分到 GAC 中去查找。如果程序集标识中没有包含语言文化或者公有密钥标记信息，LoadWithPartialName 方法的行为将处于未定义状态，因此也就不能保证 CLR 加载的总是某个特定的程序集。如果只是没有指定版本，LoadWithPartialName 方法将加载 GAC 中版本号最高的那个程序集。

重要　一些开发人员可能会注意到 System.AppDomain 也提供有一个 Load 方法。和 Assembly 的静态方法 Load 不同，AppDomain 的 Load 方法是一个实例方法，它允许我们将一个程序集加载到一个特定的应用程序域中。该方法主要用来供非托管代码调用，宿主应用程序可以使用它来将一个程序集加载到一个特定的应用程序域中。托管代码开发人员一般情况下不应该调用该方法，下面解释其原因。

当 AppDomain 的 Load 方法被调用时，它会接受一个标识程序集的字符串参数。Load 方法随后会应用一定的策略在一些通常的位置上——用户磁盘、以及 codebase 引用标识的位置——搜索程序集。我们知道每个应用程序域都有相关的设置来告诉 CLR 怎样搜索程序集。要将程序集加载到指定的应用程序域内，CLR 将使用与指定的应用程序域相关联的设置，而不是与当前调用方法所处的应用程序域相关联的设置。

但是，AppDomain 的 Load 方法会返回一个程序集引用。由于 System.Assembly 并非继承自 System.MarshalByRefObject，所以返回的程序集对象必须以传值的方式被执行封送(marshal)处理，并传回当前调用方法所处的应用程序域中。而这时 CLR 将会使用与当前调用方法所处的应用程序域相关联的设置来搜索和加载该程序集。如果使用当前应用程序域的策略和搜索位置不能找到该程序集，那么将有一个 FileNotFoundException 异常抛出。显然，这种行为不是我们所期望的，所以我们应该避免使用 AppDomain 的 Load 方法。

20.8.1　将程序集象“数据文件”一样加载

设想一个用户在他的硬盘上安装了某个实用程序。该实用程序包含两个程序集：SomeTool.exe(主应用程序)和 Component.dll(包含 SomeTool.exe 使用的一些组件)。这两个程序集的安装情况如下：

```
C:\SomeTool\SomeTool.exe
C:\SomeTool\Component.dll
```

让我们再假设用户还有一个程序集希望使用 SomeTool.exe 来处理。该程序集文件的名称碰巧也为 Component.dll，并且其存放的位置如下：

```
C:\Component.dll
```

现在假设用户执行了如下的命令：

```
C:\>C:\SomeTool\SomeTool.exe C:\Component.dll
```

SomeTool.exe 的 Main 方法开始运行并调用 Assembly.LoadFrom，传递给其的参数就是 SomeTool.exe 要处理的程序集。于是 CLR 加载 C:\Component.dll。现在再假设 Main 调用的其他方法引用到了定义在 C:\SomeTool\Component.dll 程序集中的对象。大家想想看会发生什么情况？CLR 会直接使用先前已经被加载到应用程序域中名为 Component.dll 的程序集吗？或者 CLR 会将 C:\Component.dll 程序集文件象一个"数据文件"一样来对待，而不会将其加载到应用程序域中吗？

事实上，CLR 不会将 C:\Component.dll 程序集文件和 C:\SomeTool\Component.dll 程序集文件视作同一个程序集，因为它们可以定义完全不同的类型和方法。CLR 只会做正确的事情：它会将 C:\SomeTool\Component.dll 加载到应用程序域中，SomeTool.exe 中的代码也会正常地访问其中定义的类型和方法。

下面是 Assembly.LoadFrom 方法的工作原理。当该方法被调用时，CLR 会打开指定的文件，并从文件的元数据中提取程序集的版本、语言文化和公有密钥标记。然后，所有这些信息将被作为参数传递给 Load 方法。Load 方法接着应用所有这些策略信息来搜索程序集。如果找到了匹配的程序集，CLR 会将 LoadFrom 方法指定的程序集文件的完整路径名称和 Load 方法找到的程序集文件的完整路径名称做比较。如果两个路径名称相同，那么该程序集将被认为是应用程序的一部分。如果两个路径名称不同，或者 Load 方法不能找到匹配的文件，那么该程序集将被认为是一个"数据文件"，而不被认为是应用程序的一部分。

当 CLR 需要查找通过 LoadFrom 加载的程序集所依赖的其他程序集时，CLR 会使用通常所用的探测规则。如果它不能在这些探测规则所指定的位置中找到所依赖的程序集，那么它将会在引用程序集所在的目录(以及任何名称和依赖程序集相匹配的子目录下)中继续查找。

前面曾经说过我们应该尽量使用 Load，而避免使用 LoadFrom。其中一个原因就是 LoadFrom 方法的效率要比 Load 方法低许多，因为 LoadFrom 方法内部会调用到 Load 方法，一样会去应用某些策略，并在几个磁盘位置上搜索程序集文件。另一个原因是 LoadFrom 会将程序集看作为一种“数据文件”来加载，如果我们的应用程序域从不同的路径上加载了两个相同的程序集文件，那么将有许多内存被浪费，系统运行时的性能也会降低。而调用 Load 可以确保得到最好的性能，并且相同的程序集不会被多次加载到同一个应用程序域中。

在我们打算调用 LoadFrom 方法的任何时候，我们都应该做一番仔细的考虑，尽力寻找一种使用 Load 的替代方案。当然，也有一些非用 LoadFrom 方法不可的情况(比如前面的 SomeTool.exe 例子)，这时候就要倍加小心。

20.8.2　建立一个异常类型的层次结构

ExceptionTree 示例应用程序(下面列出了它的源代码)向我们展示了所有直接或间接继承自 System.Exception 的类。我们希望这些异常类型能够形成一个树形结构。要实现这一点，我们必须显式加载所有我们希望查看的程序集。这就是应用程序中 LoadAssemblies 方法所做的工作。在所有的程序集被加载之后，我们将得到一个由应用程序域中所有的程序集组成的数组。然后，对于数组中的每一个程序集，我们又可以得到由其中的所有类型组成的数组。

对于每一个类型，应用程序将通过查询 Type 的 BaseType 属性来检查它的基类型。如果返回的 Type 为 System.Exception，那么该类型就是一个异常类型。如果返回的类型为 System.Object，那么该类型就不是一个异常类型。如果返回的类型不属于以上两种类型，那么应用程序将继续递规地检查它的基类型，直至找到 Exception 或者 Object 为止。

下面是 ExceptionTree 应用程序的源代码。程序的输出显示在本书第 18 章(408 页)，这里不再列出。(译注：第 18 章列出的只是程序输出的一小部分。)

```
using System;
using System.Text;
using System.Reflection;
using System.Collections;

class App {
   static void Main() {
      // 显式加载希望反射的程序集
      LoadAssemblies();

      // 初始化计数器和异常类型列表
      Int32 totalTypes = 0, totalExceptionTypes = 0;
      ArrayList exceptionTree = new ArrayList();
```

```
// 遍历加载到应用程序域中所有的程序集
foreach (Assembly a in AppDomain.CurrentDomain.GetAssemblies()) {

    // 遍历定义在程序集中所有的类型
    foreach (Type t in a.GetTypes()) {

        totalTypes++;

        // 如果不是一个公有类，那么忽略该类型
        if (!t.IsClass || !t.IsPublic) continue;

        // 构造一个字符串表示类型的继承层次结构
        StringBuilder typeHierarchy =
            new StringBuilder(t.FullName, 5000);

        // 假设类型不是一个继承自 Exception 的类型
        Boolean derivedFromException = false;

        // 检查类型的基类型是否为 System.Exception
        Type baseType = t.BaseType;
        while ((baseType != null) && !derivedFromException) {
            // 将基类型追加在字符串的末尾
            typeHierarchy.Append("-" + baseType);

            derivedFromException =
                (baseType == typeof(System.Exception));
            baseType = baseType.BaseType;
        }
        // 不再有基类型并且也非继承自 Exception，那么查看下一个类型
        if (!derivedFromException) continue;

        // 找到一个继承自 Exception 的类型
        totalExceptionTypes++;

        // 对于这个继承自 Exception 的类型，
        // 反转层次结构中类型的顺序
        String[] h = typeHierarchy.ToString().Split('-');
        Array.Reverse(h);

        // 构造一个新的字符串，其层次结构为
        // 从 Exception -> 继承自 Exception 的类型。
        // 将该字符串添加到异常类型列表中
        exceptionTree.Add(String.Join("-", h, 1, h.Length - 1));
    }
}
```

```
        // 根据层次结构，对异常类型进行排序
        exceptionTree.Sort();

        // 显示异常树
        foreach (String s in exceptionTree) {
            // 将异常类型的基类型分开
            String[] x = s.Split('-');

            // 按照基类型的数量进行缩进，
            // 然后显示继承关系最近的类型
            Console.WriteLine(
                new String(' ', 3 * x.Length) + x[x.Length - 1]);
        }
        // 显示所考查类型的最终状态
        Console.WriteLine("\n---> Of {0} types, {1} are " +
            "derived from System.Exception.",
            totalTypes, totalExceptionTypes);
        Console.ReadLine();
    }

    static void LoadAssemblies() {
        String[] assemblies = {
            "System,                              PublicKeyToken={0}",
            "System.Data,                         PublicKeyToken={0}",
            "System.Design,                       PublicKeyToken={1}",
            "System.DirectoryServices,            PublicKeyToken={1}",
            "System.Drawing,                      PublicKeyToken={1}",
            "System.Drawing.Design,               PublicKeyToken={1}",
            "System.EnterpriseServices,           PublicKeyToken={1}",
            "System.Management,                   PublicKeyToken={1}",
            "System.Messaging,                    PublicKeyToken={1}",
            "System.Runtime.Remoting,             PublicKeyToken={0}",
            "System.Security,                     PublicKeyToken={1}",
            "System.ServiceProcess,               PublicKeyToken={1}",
            "System.Web,                          PublicKeyToken={1}",
            "System.Web.RegularExpressions,       PublicKeyToken={1}",
            "System.Web.Services,                 PublicKeyToken={1}",
            "System.Windows.Forms,                PublicKeyToken={0}",
            "System.Xml,                          PublicKeyToken={0}",
        };
        String EcmaPublicKeyToken = "b77a5c561934e089";
        String MSPublicKeyToken = "b03f5f7f11d50a3a";

        // 获取包含 System.Object 的程序集的版本
        // 假设所有其他程序集的版本与此版本相同
        Version version =
            typeof(System.Object).Assembly.GetName().Version;
```

```
            // 显式加载希望进行反射的程序集
            foreach (String a in assemblies) {
                String AssemblyIdentity =
                    String.Format(a, EcmaPublicKeyToken, MSPublicKeyToken) +
                     ", Culture=neutral, Version=" + version;

                Assembly.Load(AssemblyIdentity);
            }
        }
    }
```

20.9 显式卸载程序集：卸载应用程序域

CLR 不支持单独卸载程序集的能力，它只支持卸载应用程序域。卸载应用程序域会导致包含在其中的所有程序集都被卸载。卸载一个应用程序域非常容易，我们只需要调用 AppDomain 的静态方法 Unload，并为其传递希望卸载的 AppDomain 对象引用即可。

> **注意** 如前所述，以中立域形式加载的程序集不可能从一个应用程序域中被卸载掉。要“卸载”这些程序集，必须中断当前进程。

在下面的 AppDomainRunner 示例应用程序中，我们首先创建了一个新的应用程序域，然后使用了其中的类型，最后又将该应用程序域和其内所有的程序集一起卸载掉。示例代码还向大家展示了怎样定义一个可以跨越应用程序域边界、并通过引用的方式来进行封送处理的类型。最后，示例代码还向我们展示了如果试图访问一个通过引用封送处理(marshal-by-reference)的对象，而该对象所处的应用程序域已经被卸载的情况下会发生什么事情。

```
using System;
using System.Reflection;
using System.Threading;
class App {
    static void Main() {
        // 创建一个新的应用程序域
        AppDomain ad =AppDomain.CreateDomain("MyNewAppDomain", null, null);

        // 在新创建的应用程序域中创建一个新的 MarshalByRef 对象
        MarshalByRefType mbrt = (MarshalByRefType)
            ad.CreateInstanceAndUnwrap(
                Assembly.GetCallingAssembly().FullName,
                "MarshalByRefType");
```

```
        // 调用该对象上的一个方法。代理对象会将方
        // 法调用远程传送到新创建的应用程序域中
        mbrt.SomeMethod(Thread.GetDomain().FriendlyName);

        // 新创建的应用程序域已经使用完毕，卸载
        // 该应用程序域及其内所有的程序集
        AppDomain.Unload(ad);

        // 试图在已经卸载了的应用程序域中的对象上调
        // 用方法。由于该对象在应用程序域卸载时已经
        // 被销毁，所以将有异常抛出
        try {
            mbrt.SomeMethod(Thread.GetDomain().FriendlyName);

            // 下面代码的输出不会被显示
            Console.WriteLine(
                "Called SomeMethod on object in other AppDomain.\n" +
                "This shouldn't happen.");
        }
        catch (AppDomainUnloadedException) {
            // 异常在这里被捕获，所以下面代码的输出将被显示
            Console.WriteLine(
                "Fail to call SomeMethod on object in other AppDomain.\n" +
                "This should happen.");
        }

        Console.ReadLine();
    }
}

// 下面的类型继承自 MarshalByRefObject
class MarshalByRefType : MarshalByRefObject {

    // 下面的实例方法可以通过一个代理对象被调用
    public void SomeMethod(String sourceAppDomain) {

        // 显示调用方法所处的应用程序域和当前对象所处的应用程序域的名称。
        // 注意：应用程序的线程在两个应用程序域之间进行了跳转
        Console.WriteLine(
            "Code from the '{0}' AppDomain\n" +
            "called into the '{1}' AppDomain.",
            sourceAppDomain, Thread.GetDomain().FriendlyName);
    }
}
```

编译并运行上面的代码，我们将会看到以下输出：

```
Code from the 'AppDomainRunner.exe' AppDomain
called into the 'MyNewAppDomain' AppDomain.
Fail to call SomeMethod on object in other AppDomain.
This should happen.
```

20.10 获取一个 System.Type 对象的引用

反射通常用于根据运行时(而非编译时)得到的信息来获知类型和操作对象。显然，这种对类型和对象的动态搜索和操作伴随有一定的性能损失，所以我们应该尽可能地避免使用这种技术。另外，编译器也不能帮助我们发现和修正使用反射时可能带来的一些和类型安全相关的编程错误。

System.Type 是我们进行类型和对象操作的起点。System.Type 是一个继承自 System.Reflection.MemberInfo(因为一个类型可以是另一个类型的成员)的抽象类型。FCL 中提供了几个继承自 System.Type 的类型：System.RuntimeType 和 System.Reflection.TypeDelegator，以及定义在 System.Reflection.Emit 命名空间中的 EnumBuilder 和 TypeBuilder。除去 FCL 中的几个类外，微软不希望定义任何其他继承自 Type 的类型。

> **注意** TypeDelegator 类允许代码通过封装一个 Type 来将一个 Type 子类化，从而在让原来的 Type 处理大多数工作的同时允许我们重写某些功能。一般情况下，TypeDelegator 没有太大的用处。实际上，微软也意识不到会有什么人去使用 TypeDelegator 做何种事情。

在所有这些类型中，System.RuntimeType 是目前最有意义的一个类型。RuntimeType 是一个 FCL 内部的类型，这意味着我们在.NET 框架文档中看不到它。当应用程序域中的类型第一次被访问时，CLR 会构造一个 RuntimeType 的实例，并初始化该实例的字段来“reflect” (既是“反射”，也是“反映”，这里具有双关语义)类型的信息。

我们知道 System.Object 定义有一个名为 GetType 的方法。当我们调用该方法时，CLR 会判断指定对象的类型，并返回一个指向它的 RuntimeType 对象的引用。

由于应用程序域中的每个类型都只有一个对应的 RuntimeType 对象，所以我们可以使用判等和判异两个操作符来比较两个对象的类型是否相同：

```
Boolean AreObjectsTheSameType(Object o1, Object o2) {
   return o1.GetType() == o2.GetType();
}
```

除了 Object 的 GetType 方法，FCL 还提供了另外几种方式来供我们获取一个 Type 对象：

- System.Type 提供有几个静态 GetType 方法的重载版本。该方法的所有版本都有一个 String 类型的参数。这个参数必须是类型的完整名称(包括它的命名空间)，编译器认为的基元类型名称(例如 C#中的 int、string、bool、等等)是不被接受的。如果该参数只是一个类型名称，GetType 方法会检查调用程序集来看其中是否定义有指定名称的类型。如果定义有，那么方法将返回一个 RuntimeType 对象引用。

  如果调用程序集中没有定义指定的类型，那么 GetType 方法将检查定义在 MSCorlib.dll 中的类型。如果仍没有找到和指定名称相匹配的类型，GetType 方法将根据我们调用的对象和为其传递的参数返回一个 null、或者抛出一个 System.TypeLoadException 异常。.NET 框架文档对此有详细的描述。

  我们也可以为 GetType 方法传递一个限定程序集的类型字符串，例如"System.Int32, mscorlib, Version=1.0.3300.0, Culture=neutral, PublicKey-Token=b77a5c561934e089"。在这种情况下，GetType 方法将会到指定的程序集中查找类型(如果必要的话会加载程序集)。

- System.Type 还提供有以下两个实例方法：GetNestedType 和 GetNestedTypes。

- System.Reflection.Assembly 类型提供有以下三个实例方法：GetType、GetTypes 和 GetExportedTypes。

- System.Reflection.Module 类型提供有以下三个实例方法：GetType、GetTypes 和 FindTypes。

许多编程语言还提供有操作符来允许我们根据一个类型名来获取一个 Type 对象。我们应该尽可能地使用这样的操作符来代替前面列出的各种方法，因为操作符一般会产生更快的代码。在 C#中，这样的操作符就是 typeof。下面的代码演示了它的用法：

```
static void SomeMethod() {
   Type t = typeof(MyType);
   Console.WriteLine(t.ToString());        // 显示"MyType"
}
```

大家可以编译上面的代码，并用 ILDasm.exe 查看生成的 IL 指令。下面是经过注释后的 IL 指令：

```
.method private hidebysig static void SomeMethod() cil managed
{
   // Code size 23 (0x17)
   .maxstack 1
   .locals ([0] class [mscorlib]System.Type t)

   // 查询 MyType 的元数据标记，并将一个指
   // 向内部数据结构的“句柄”放在堆栈上
   IL_0000: ldtoken MyType

   // 查询 RuntimeTypeHandle，并将一个对应
   // 的 RuntimeType 对象引用放在堆栈上
   IL_0005: call class [mscorlib]System.Type
       [mscorlib]System.Type::GetTypeFromHandle(
          valuetype [mscorlib]System.RuntimeTypeHandle)

   // 将 RuntimeType 引用保存在本地变量 t 中
   IL_000a: stloc.0

   // 将 t 中的引用加载到堆栈上
   IL_000b: ldloc.0

   // 调用 RuntimeType 对象的 ToString 方法
   IL_000c: callvirt instance string [mscorlib]System.Type::ToString()

   // 将前面得到字符串传递给 Console.WriteLine.
   IL_0011: call void [mscorlib]System.Console::WriteLine(string)

   // 从方法中返回
   IL_0016: ret
} // end of method App::SomeMethod
```

下面对上述指令做一解释。IL 指令 ldtoken 首先接受一个元数据标记作为操作数。该指令告诉 CLR 通过查询一个内部的数据结构来表示指定的元数据标记。如果不存在表示该元数据标记的数据结构，CLR 会立即创建一个。然后一个指向该数据结构的“句柄”(用一个 System.RuntimeTypeHandle 值类型表示)会被放到虚拟堆栈上。该“句柄”实际上就是前面那个内部数据结构的内存地址，但是我们应该避免直接去访问这些内部的数据结构。

接着，System.Type的静态方法GetTypeFromHandle被调用。该方法接受前面的“句柄”作为参数，并返回一个表示类型的RuntimeType对象引用。剩下的IL指令将RuntimeType引用保存在变量t中，然后在t上调用ToString方法。最后，得到的字符串被传递给Console.WriteLine方法，然后返回。

> **注意** IL指令ldtoken允许我们指定一个元数据标记来表示下面6种元数据表中的一个条目：类型定义/引用表、方法定义/引用表、字段定义/引用表。但是C#中的typeof操作符只接受模块中定义或者引用的类型名作为操作数，它不接受字段或者方法名称。
>
> 我们很少会需要获取一个字段或者方法的“句柄”，这也是为什么大多数编译器没有为接受字段或者方法元数据标记作为参数的ldtoken指令提供操作符的原因。字段和方法“句柄”对于编译器开发人员比较有用，应用程序开发人员一般用不到它们。如果大家对字段句柄比较感兴趣，可以参见System.RuntimeFieldHandle类型和System.Reflection.FieldInfo类型的静态方法GetFieldFromHandle和实例属性Handle。对于方法句柄来说，可以参见System.RuntimeMethodHandle类型和System.Reflection.MethodBase类型的静态方法GetMethodFromHandle和实例属性MethodHandle。

一旦我们有了一个Type对象引用，我们就可以通过查询它的一些属性来了解有关类型的许多内容。大多数的属性，如IsPublic、IsSealed、IsAbstract、IsClass、IsValueType等，都表示的是和类型相关的一些标记。其他一些属性，如Assembly、AssemblyQualifiedName、FullName、Module等，则返回的是类型所定义的程序集、模块、以及类型的完整名称。我们还可以通过查询BaseType属性来获取类型的基类型。另外，一些方法还会给予我们有关类型的更多信息。

.NET框架文档描述了Type类型提供的所有方法和属性，数量非常之多。例如，Type类型提供的公有实例属性就有大约45个。这还不包括Type中定义的方法和字段。接下来的一节将会讨论到其中的一些方法。

注意 顺便提一句，如果我们需要获取一个 Type 对象来标识一个指向类型的引用，我们可以调用 Type 类型的任何一个 GetType 方法，并在类型名称前加上一个&符号作为参数传递给它，看下面的示例代码:

```
using System;
using System.Reflection;
class App {
    static void Main() {
        // 获取 SomeMethod 方法的参数
        ParameterInfo[] p =
            typeof(App).GetMethod("SomeMethod").GetParameters();

        // 获取一个标识 String 引用的 Type 对象
        Type stringRefType = Type.GetType("System.String&");

        // 判断 SomeMethod 的第一个参数是否为一个 String 引用
        Console.WriteLine(p[0].ParameterType == stringRefType);
        // "True"
    }

    // 如果将 ref 改为 out，得到的结果相同
    public void SomeMethod(ref String s) {
        s = null;
    }
}
```

20.11 反射一个类型的成员

字段、构造器、方法、属性、事件以及嵌套类型都可以被定义为一个类型的成员。FCL 中包含有一个名为 System.Reflection.MemberInfo 的类型。本章前面讨论过的各个版本的 Reflector 示例应用程序中就使用了这个类型来查找一个类型中定义了哪些成员。表 20.3 列出了 MemberInfo 类型提供的几个属性和方法。这些属性和方法适用于所有的类型成员。

表 20.3 中列出的大多数属性都很容易理解。但是，开发人员通常会被 DeclaringType 和 ReflectedType 所迷惑。为了理解这两个属性，我们定义如下的类型:

```
class MyType {
   public override String ToString() { return "Hi"; }
}
```

那么执行下面的代码会得到什么结果呢？

```
MemberInfo[] members = typeof(MyType).GetMembers();
```

表 20.3　所有继承自 MemberInfo 的类型所共有的属性和方法

成员名称	成员类型	描　述
Name	String 属性	返回一个表示该成员的 String
MemberType	MemberTypes (枚举类型) 属性	返回该成员的种类——字段、构造器、方法、属性、事件、类型(非嵌套类型)、或者嵌套类型
DeclaringType	Type 属性	返回定义该成员的 Type
ReflectedType	Type 属性	返回用来获取该成员的 Type
GetCustomAttributes	方法，返回一个 Object 数组	返回一个数组，数组元素表示应用到该成员上的定制特性实例。定制特性可以应用于任何成员
IsDefined	方法，返回一个 Boolean	如果成员上应用有至少一个指定的定制特性，方法将返回 true

变量 members 是一个数组引用，数组中的每个元素都标识着一个由 MyType 以及它的所有基类(例如 System.Object)定义的公有成员。如果我们查询表示 ToString 方法的 MemberInfo 元素的 DeclaringType 属性，我们将得到一个 MyType，因为 MyType 中声明了(或者说定义了)一个 ToString 方法。另一方面，如果我们查询表示 Equals 方法的 MemberInfo 元素的 DeclaringType 属性，我们将得到一个 System.Object，因为 Equals 声明在 System.Object 中，而非 MyType 中。

但 ReflectedType 属性将总是返回 MyType，因为它是调用 GetMembers 方法执行反射时指定的类型。如果我们查询.NET 框架文档中的 MemberInfo 类型，我们将会看到它是一个直接继承自 System.Object 的类。图 20.2 为我们展示了上述反射类型的层次结构。

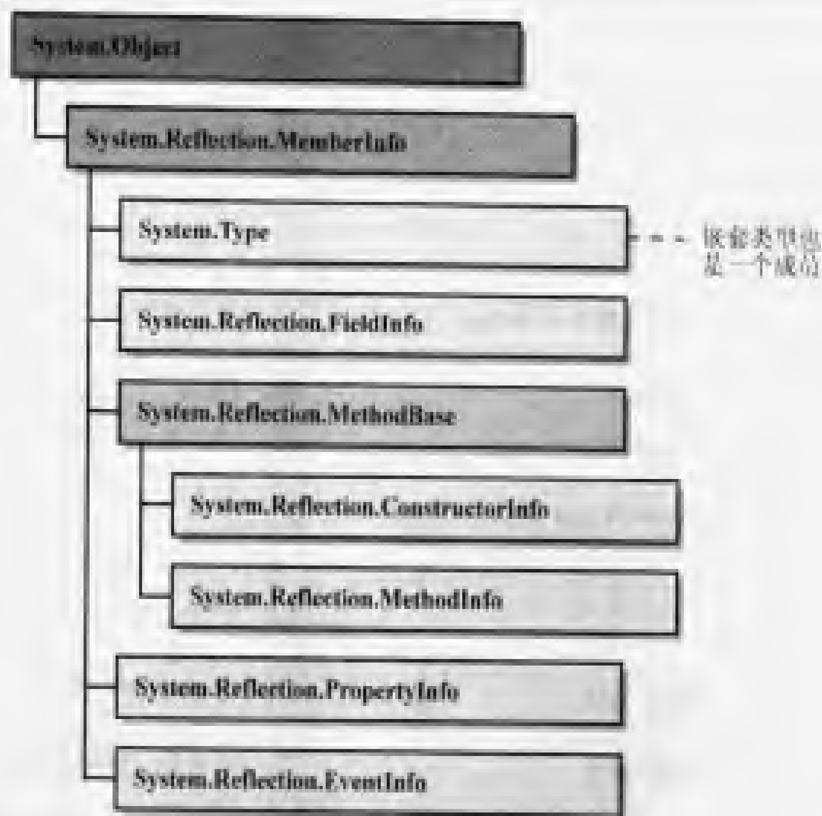


图 20.2 反射类型的层次结构

> **注意** 不要忘记了 System.Type 也是一个继承自 MemberInfo 的类型，因此 Type 也提供有表 20-3 所示的所有属性。

GetMembers 方法返回的数组中的每一个元素都是上面反射类型层次结构中的一个具体类型。除了 Type 的 GetMembers 方法能够返回类型中所有的成员外，Type 还提供有一些方法可以返回特定的成员类型，例如，GetNestedTypes、GetFields、GetConstructors、GetMethods、GetProperties 以及 GetEvents 方法。这些方法返回的都是一个数组，其元素类型分别为 Type、FieldInfo、ConstructorInfo、MethodInfo、PropertyInfo 以及 EventInfo。

图 20.3 为我们总结了可以用来遍历反射对象模型的各种类型。首先，从一个应用程序域出发，我们可以得到其内加载的所有程序集。根据程序集，我们又可以得到组成它的所有模块。由程序集或者模块，我们又可以得到它里面定义的所有类型。最后根据类型，我们又可以得到它的嵌套类型、字段、构造器、方法、属性以及事件。

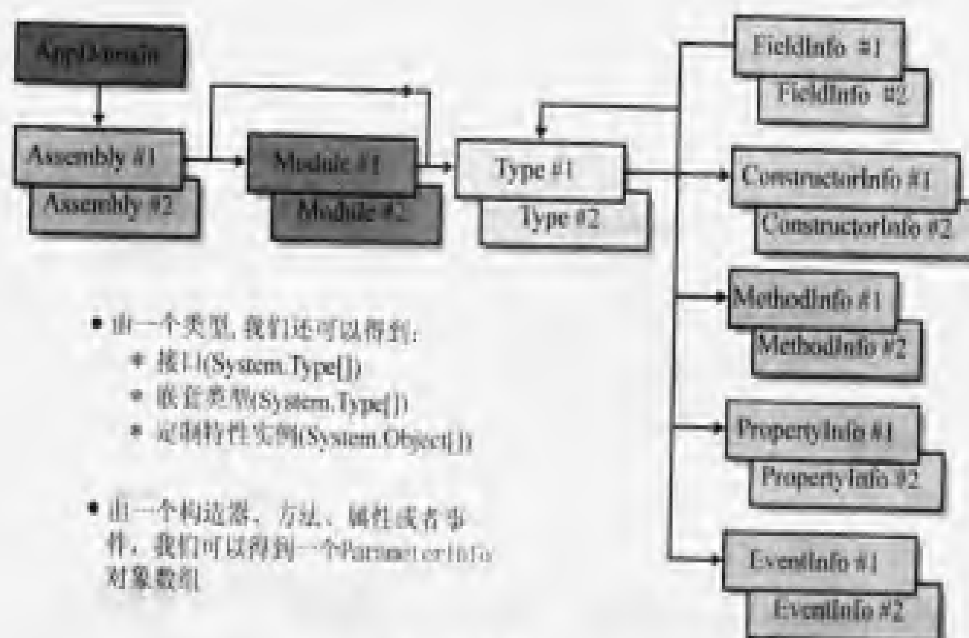


图 20.3　可以用来遍历反射对象模型的各种类型

20.11.1　创建一个类型的实例

一旦我们有了一个继承自 Type 的对象引用，我们可能希望创建一个该类型的实例。FCL 提供了以下几种机制供我们实现这一目的。

- **System.Activator 的 CreateInstance 方法**　Activator 类提供有几个重载版本的静态方法 CreateInstance。当我们调用该方法时，我们可以为其传递一个 Type 对象引用或者一个 String，其中 String 标识了我们希望创建对象的类型。接受 Type 对象作为参数的那些版本比较简单。在这些版本中，除了 Type 对象外，我们还要为类型的实例构造器指定一些参数，这时方法返回的是一个指向新创建对象的引用。

  而接受 String 作为参数的那些版本有些复杂。首先，我们必须指定一个字符串来表示类型所处的程序集。其次，如果我们能够正确配置远程选项，这些方法还允许我们构造远程对象。最后，这些版本返回的不是一个新创建的对象引用。相反，它们返回的是一个 System.Runtime.Remoting.ObjectHandle(继承自 System.MarshalByRefObject)。

  ObjectHandle 允许我们在一个应用程序域中创建一个对象，然后将其传递到其他的应用程序域中，但是却不用立即强制将定义该对象类型的程序集加载到它们当中。当我们要访问对象时，我们首先要调用 ObjectHandle 的 Unwrap 方法。

只有当 Unwrap 方法被调用时，包含类型元数据的程序集才会被加载。如果该程序集不能被加载，Unwrap 方法将抛出一个 System.Runtime.Remoting.RemotingException 异常。注意 Unwrap 方法必须在对象的生存期结束之前被调用；默认时间为 5 分钟。

- **System.Activator 的 CreateInstanceFrom 方法** Activator 类还提供有一组重载版本的静态方法 CreateInstanceFrom。这些方法的行为和 CreateInstance 方法的行为类似，只是我们必须通过字符串参数指定类型和它所在的程序集。程序集将使用 Assembly 的 LoadFrom(而非 Load)方法被加载到当前调用方法所处的应用程序域中。由于所有的 CreateInstanceFrom 方法都不接受 Type 参数，所以它们返回的都是一个 ObjectHandle 引用，我们在使用时必须首先调用 Unwrap 方法。

- **System.AppDomain 的一些方法** AppDomain 类型提供了四种构造类型实例的方法：CreateInstance、CreateInstanceAndUnwrap、CreateIntanceFrom、CreateInstanceFromAndUnwrap。这些方法的行为和 Activator 类提供的方法的行为非常类似，不同之处在于它们都是实例方法，所以它们允许我们指定在哪个应用程序域中构造对象。另外，那些带有 Unwrap 后缀的方法还为我们提供了一些方便，因为我们不必再进行额外的方法调用就可以直接访问新创建的对象。

- **System.Type 的 InvokeMember 实例方法** 通过一个 Type 对象引用，我们可以调用它的 InvokeMember 方法。该方法会根据我们所传的参数查找相匹配的构造器，然后构造类型实例。类型实例总是被创建在当前方法调用所处的应用程序域中。InvokeMember 方法最后会返回新创建的对象引用。本章稍后会对该方法做更详细的讨论。

- **System.Reflection.ConstructorInfo 的 Invoke 实例方法** 利用一个 Type 对象引用，我们可以绑定到一个特殊的构造器上，并获得一个该构造器的 ConstructorInfo 对象引用。然后我们可以利用这个 ConstructorInfo 对象引用来调用它的 Invoke 方法。同样，Invoke 方法总是将对象创建在调用方法当前所处的应用程序域中，方法返回的也是新创建的对象引用。本章稍后也会对该方法做更详细的讨论。

> **注意**　CLR 对值类型是否定义构造器并没有做强制的要求。这就出现了一个问题，因为前面列表中所有的机制都要求通过调用构造器来构造一个类型的实例。为了解决这个问题，微软"加强"了 Activator 的 CreateInstance 方法中某些重载版本的功能，允许它们不用调用构造器就可以创建一个值类型的实例。如果希望不用调用构造器就可以创建一个值类型的实例，我们只能从以下两个版本的方法中选择其一：即参数为 Type 的那个版本的 CreateInstance 方法、或者参数为 Type 和 Boolean 的那个版本的 CreateInstance 方法。

前面列出的几种机制允许我们创建除数组(继承自 System.Array 的类型)和委托(继承自 System.MulticastDelegate 的类型)之外所有类型的实例。

要创建一个数组实例，我们应该调用 Array 的静态方法 CreateInstance(存在几个重载的版本)。所有版本的 CreateInstance 方法接受的第一个参数都是希望创建的数组实例中元素的类型(一个 Type 引用)。而其他参数则允许我们指定数组的维数和上下限的各种组合。

要创建一个委托实例，我们应该调用 Delegate 的静态方法 CreateDelegate(同样存在几个重载的版本)。所有版本的 CreateDelegate 方法接受的第一个参数都是希望创建的委托实例的类型(一个 Type 引用)。而其他参数则允许我们指定要创建的委托实例包装的是一个对象的哪个实例方法、或者一个类型的哪个静态方法。

20.11.2　调用一个类型的方法

调用一个方法最方便的做法便是调用 Type 类型的 InvokeMember 方法。该方法非常强大，利用它我们可以进行许多操作。InvokeMember 方法也存在几个重载的版本。下面讨论参数最多的那个版本，其他的重载版本都是在该版本的基础上选择一些默认参数进行调用的。

```
class Type {
   public Object InvokeMember(
      String name,                 // 成员名称
      BindingFlags invokeAttr,     // 怎样搜索成员
      Binder binder,               // 怎样匹配成员及其参数
      Object target,               // 成员调用的对象
      Object[] args,               // 传递给方法的参数
      CultureInfo culture);        // 某些绑定器(binder)使用的语言文化
   ...
}
```

当我们调用 InvokeMember 方法时，它会查看类型的成员以寻找相应的匹配。如果没有找到这样的匹配，InvokeMember 方法将抛出一个 System.MissingMethodException 异常。如果找到一个这样的匹配，InvokeMember 将调用匹配的方法。InvokeMember 方法的返回值就是调用匹配方法的返回值。但是，如果匹配方法的返回值为 void，InvokeMember 将返回 null。如果我们调用的匹配方法抛出了一个异常，InvokeMember 将捕获该异常，并抛出一个新的 System.Reflection.TargetInvocationException 异常。TargetInvocationException 对象的 InnerException 属性将包含所调用的匹配方法实际抛出的异常。从个人的角度来看，我不喜欢这种处理方式。我更希望 InvokeMember 方法不对异常做任何包装，而直接让它们传递出去。

InvokeMember 方法在内部会执行两项操作。第一，它必须选择要调用的成员——这个过程称作**绑定**(binding)。第二，它必须调用该成员——这个过程称作**调用**(invoking)。

当我们调用 InvokeMember 方法时，我们为其传递的 name 参数指出了我们希望调用的成员的名称。但是，一个特定的名称在一个类型中可能会对应着好几个成员。毕竟，同一个名称的成员可以有好几个重载版本，甚至一个方法和一个字段也可以有相同的名称。(译注：实际上，除了相同种类的成员重载以外，C#不允许一个类中不同种类的成员具有相同的名称，也就是说在一个类中定义同名的方法和字段将会出现编译时错误。)当然，InvokeMember 在调用之前必须先绑定到一个成员上。所有除了 target 外的参数都是用来帮助 InvokeMember 方法确定到底绑定哪个成员的。下面我们来更近一步查看这些参数。

参数 binder 标识了一个类型继承自 System.Reflection.Binder(一个抽象类型)的对象。继承自 Binder 的类型封装了 InvokeMember 方法选择绑定一个成员的规则。Binder 基类型中定义了一些抽象的虚方法，如 BindToField、BindToMethod、ChangeType、ReorderArgumentArray、SelectMethod 以及 SelectProperty。InvokeMember 方法会在内部使用 binder 参数来调用这些方法。

微软定义了一个名为 DefaultBinder 的具体类型，它是一个内部类型(没有记入文档)，其继承自 Binder 基类型。DefaultBinder 是和 FCL 一起发布的一个类型，微软预期几乎每个人都会使用到该类型。当我们将 null 值传递给 InvokeMember 的 binder 参数时，它不会使用 null，相反它会转而使用 DefaultBinder。Type 类型有一个名为 DefaultBinder 的公有只读属性，如果需要，我们可以通过查询该属性来获得一个 DefaultBinder 对象引用。

如果 DefaultBinder 定义的规则不能适用于我们的应用程序，我们就必须定义自己的派生自 Binder 的类型，并将其实例作为 binder 参数传递给 InvokeMember 方法。大家可以到 *http://www.Wintellect.com/* 上下载本书相关的源代码，其中有一个简单的应用 Binder 派生类型的示例。

当绑定器(binder)对象上的方法被调用时，它们需要接受一些参数来进行某些判断。当然，正在搜索的成员的名称是需要传递给绑定器对象的。

另外，指定的 BindingFlags 位标记、以及需要传递给被调用成员的所有参数的类型也要传递给绑定器对象上的方法。

本章前面表 20.2 曾经描述了以下几个 BindingFlags 位标记：Default、IgnoreCase、DeclaredOnly、Instance、Static、Public、NonPublic 以及 FlattenHierarchy。这些位标记会告诉绑定器对象哪些成员应该被包含在搜索中。

除了这些位标记外，绑定器对象还可以通过查看 InvokeMember 的 args 参数来获得所传递参数的数量，利用参数的数量，绑定器对象可以缩小可能的匹配集合的范围。通过检查各个参数的类型，绑定器对象可以进一步缩小该集合的范围。对于参数的类型，绑定器对象可能会应用某些自动类型转换以期获得更多的灵活性。例如，某个类型可能定义了一个参数为 Int64 的方法。如果我们调用 InvokeMember、并为其 args 参数传递了一个 Int32 数组(其中包含着一个 Int32 值)，DefaultBinder 将认为这是一个匹配。当方法被调用时，该 Int32 值将被转换为一个 Int64 值。DefaultBinder 支持表 20.4 中列出的几种转换。

表 20.4 DefaultBinder 支持的几种转换

源类型	目标类型
任何类型	该类型的基类型
任何类型	该类型所实现的接口
Char	UInt16、UInt32、Int32、UInt64、Int64、Single、Double
Byte	Char、UInt16、Int16、UInt32、Int32、UInt64、Int64、Single、Double
SByte	Int16、Int32、Int64、Single、Double
UInt16	UInt32、Int32、UInt64、Int64、Single、Double
Int16	Int32、Int64、Single、Double
UInt32	UInt64、Int64、Single、Double
Int32	Int64、Single、Double
UInt64	Single、Double
Int64	Single、Double
Single	Double
非引用	传引用

还有两个 BindingFlags 位标记允许我们调整 DefaultBinder 的行为。表 20.5 列出了它们。

表 20.5 DefaultBinder 使用的 BindingFlags

符 号	值	描 述
ExactBinding	0x010000	绑定器将查找那些参数类型和传递进来的参数类型相匹配的成员。该位标记只能用于 DefaultBinder 类型，因为定制绑定器的实现会去选择合适的成员。注意绑定器可以选择忽略该标记。实际上，当 DefaultBinder 没能找到一个匹配时，它并不会就此停止。如果传递的参数在不失精度的前提下可以强制转换为一个兼容的类型，那么 DefaultBinder 也将认为这是一个合适的匹配
OptionalParamBinding	0x040000	对于任何参数个数和传递进来的参数个数相匹配的成员，绑定器都将其考虑作为一个匹配。该位标记用于参数中有默认值的成员、或者接受可变数目参数的方法。该位标记只能与 Type 类型的 InvokeMember 方法一起使用

InvokeMember 的最后一个参数 cultrue 也是做绑定之用。但是，DefaultBinder 会完全忽略该参数。如果我们定义了自己的绑定器，我们可能会使用 cultrue 参数来帮助进行参数类型转换。例如，调用者可能会传递一个值为“1,23”的字符串参数。而绑定器可能会检查该字符串，并使用指定的 cultrue 进行解析，将参数的类型转换为一个 Single(如果指定的语言文化为“de-DE”)，或者继续将参数看作为一个 String(如果指定的语言文化为“en-US”)。

到目前为止，我们已经讨论过 InvokeMember 中所有和绑定相关的参数了。还有一个我们没有讨论过的参数是 target。该参数是一个我们期望在其上进行成员调用的对象引用。如果我们调用的是一个类型的静态方法，那么我们应该将 null 传递给该参数。

InvokeMember 是一个非常强大的方法。它允许我们调用一个方法(前面已经讨论过了)，构造一个类型的实例(一般是通过调用一个构造器方法)，以及获取或者设置一个字段的值。我们可以通过指定一个表 20.6 中列出的 BindingFlags 值来告诉 InvokeMember 执行哪种操作。

表 20.6 列出的大部分位标记都是互斥的——我们在调用 InvokeMember 时必须选择其中一个、且只能选择一个。但是，我们可以同时指定 GetField 和 GetProperty，在这种情况下，InvokeMember 会首先搜索匹配的字段，如果找不到匹配的字段，再去搜索匹配的属性。

表20.6 InvokeMember使用的BindingFlags

符 号	值	描 述
InvokeMethod	0x0100	告诉InvokeMember调用一个方法
CreateInstance	0x0200	告诉InvokeMember创建一个新对象并调用其构造器
GetField	0x0400	告诉InvokeMember获取一个字段的值
SetField	0x0800	告诉InvokeMember设置一个字段的值
GetProperty	0x1000	告诉InvokeMember调用一个属性的get访问器方法
SetProperty	0x2000	告诉InvokeMember调用一个属性的set访问器方法

类似地，我们也可以同时指定SetField和SetProperty，它们匹配的顺序和前面的相同。绑定器使用这些位标记来缩小可能的匹配集合。如果我们指定的是BindingFlags.CreateInstance位标记，绑定器对象将知道它只可以选择一个构造器方法。

重要 到目前为止，大家可能会认为使用反射可以很容易地绑定并调用一个非公有成员，从而使得应用程序代码能够访问到一些私有成员，而这是编译器通常所不允许的。但实际上，反射会使用代码访问安全(code access security，简称CAS)机制来确保这种能力不会被滥用。

当我们调用一个方法进行成员绑定时，该方法会首先检查我们试图绑定的成员在编译时对我们是否可见。如果答案是肯定的，那么绑定将会成功。如果成员在编译时对我们不可见，那么方法将会查询System.Security.Permissions.ReflectionPermission许可，也就是查看其Flags属性是否设置了System.Security.Permissions.ReflectionPermissionFlags的TypeInformation位标记。如果设置了该位标记，成员将被成功地绑定。否则，系统将抛出一个System.Security.SecurityException异常。

当我们调用一个方法进行成员调用时，该方法将执行和绑定成员类似的安全检查。唯一不同的是这一次它将检查ReflectionPermission的Flags属性是否设置了ReflectionPermissionFlags的MemberAccess位标记，如果设置了该位标记，成员将被调用。否则，系统也将抛出一个System.Security.SecurityException异常。

20.11.3 一次绑定、多次调用

利用 Type 的 InvokeMember 方法，我们可以访问一个类型的所有成员。但是，我们应该清楚每次调用 InvokeMember 时，它都必须先要被绑定到一个特定的成员上，然后才能进行调用。每次进行成员调用都要执行这样一次绑定是非常耗时的，如果我们频繁地进行这样的操作，应用程序的性能将会受到一定的损伤。因此，如果我们打算频繁地访问一个成员，我们最好一次性地绑定到期望的成员上，之后便可以按照任意的频度来访问它们。

绑定一个成员(并不进行调用)可以通过调用以下几个 Type 类型的方法来完成：GetFields、GetConstructors、GetMethods、GetProperties、GetEvents 以及其他类似的方法。所有这些方法返回的对象类型都为我们提供了直接访问特定成员的方法。表 20.7 总结了这些类型、以及我们可以在其上调用的方法。

表 20.7 用来绑定成员的类型

类 型	成员描述
FieldInfo	调用 GetValue 可以获取字段的值
	调用 SetValue 可以设置字段的值
ConstructorInfo	调用 Invoke 可以构造类型的一个实例
MethodInfo	调用 Invoke 可以调用类型的一个方法
PropertyInfo	调用 GetValue 可以调用一个属性的 get 访问器方法
	调用 SetValue 可以调用一个属性的 set 访问器方法
EventInfo	调用 AddEventHandler 可以调用一个事件的 add 访问器方法
	调用 RemoveEventHandler 可以调用一个事件的 remove 访问器方法

其中，PropertyInfo 类型仅表示着一个属性的元数据信息(第 10 章曾经讨论过)。这些元数据信息可以通过 PropertyInfo 的 CanRead、CanWrite 以及 PropertyType 只读属性来获取。它们指出了一个属性是否可读、可写以及它的数据类型是什么。PropertyInfo 有一个 GetAccessors 方法，其返回值为一个 MethodInfo 数组：其中一个是 get 访问器方法(如果存在的话)，另一个是 set 访问器方法(如果存在的话)。PropertyInfo 提供的 GetGetMethod 和 GetSetMethod 方法可能更具价值，它们将只返回一个 MethodInfo 对象。PropertyInfo 提供的 GetValue 和 SetValue 两个方法仅仅是为了帮助我们更方便地操作 PropertyInfo 对象，它们在内部会获取适当的 MethodInfo，并进行相应的调用。

和 PropertyInfo 类型相似，EventInfo 类型仅表示着一个事件的元数据信息(第 11 章曾经讨论过)。EventInfo 类型提供有一个 EventHandlerType 只读属性，该属性返回一个 Type，表示事件对应的委托类型。EventInfo 还提供有 GetAddMethod 和 GetRemoveMethod 两个方法，它们都返回的是一个适当的 MethodInfo 对象。类似地，EventInfo 提供的 AddEventHandler 和 RemoveEventHandler 两个方法也是为了我们能够更方便地操作 EventInfo 对象，它们也会在内部获取适当的 MethodInfo，并进行相应的调用。

当表 20.7 右边列出的任何一个方法被调用时，它们无需再执行成员的绑定工作，它们唯一的操作就是调用成员。我们可以多次调用这些方法，因为不再需要绑定，它们的性能自然也会比较好。

大家可能注意到了 ConstructorInfo 的 Invoke、MethodInfo 的 Invoke、PropertyInfo 的 GetValue 和 SetValue 方法都提供有重载版本，它们接受的参数包括类型为继承自 Binder 的对象和一些 BindingFlags。这可能会使大家认为这些方法会去执行成员的绑定工作，但是事实并不是这样。

当这些方法被调用时，类型为继承自 Binder 的对象将被用来执行一些类型转换(例如将一个 Int32 参数转换为一个 Int64)以便方法调用能够成功。至于 BindingFlags 参数，这里只能传递 BindingFlags.SuppressChangeType 位标记。类似于 ExactBinding 位标记，绑定器对象可以选择忽略这个位标记。但是，DefaultBinder 不会忽略它。当 DefaultBinder 看到这个位标记时，它不会转换任何参数。如果我们使用了这个位标记，而传进来的参数又和方法期望的参数不匹配，那么将有一个 ArgumentException 异常被抛出。

通常，当我们使用 BindingFlags.ExactBinding 位标记绑定一个成员时，我们也要指定 BindingFlags.SuppressChangeType 位标记来调用该成员。如果没有同时使用这两个位标记，成员调用很有可能会失败，除非我们传递的参数和方法期望的参数能够精确地匹配。顺便提一句，当我们通过 MemberInfo 类型的 InvokeMethod 方法(译注：这里应该为 Type 类型的 InvokeMember 方法)来绑定和调用一个成员时，对于上面两个绑定位标记，我们可能希望同时指定、或者同时都不指定。

下面的示例应用程序向大家演示了使用反射来访问类型成员的各种方式。代码中既展示了怎样使用 Type 类型的 InvokeMember 方法来同时绑定和调用一个成员，也展示了怎样先进行成员的绑定，然后再执行调用。

```
// 将下面一行语句注掉可以将
// 绑定和调用操作分开进行
#define BindAndInvokeTogether

using System;
using System.Reflection;
using System.Threading;
```

```
// 下面的类用来演示反射机制。其中定义
// 有字段、构造器、方法、属性、和事件
class SomeType {
    Int32 someField;
    public SomeType(ref Int32 x) { x *= 2; }
    public override String ToString() { return someField.ToString(); }
    public Int32 SomeProp {
        get { return someField; }
        set {
            if (value < 1)
                throw new ArgumentOutOfRangeException(
                    "value", value, "value must be > 0");
            someField = value;
        }
    }
    public event ThreadStart SomeEvent;
    private void NoCompilerWarnings() {
        SomeEvent.ToString();
    }
}

class App {
    static void Main() {
        Type t = typeof(SomeType);
        BindingFlags bf = BindingFlags.DeclaredOnly |
            BindingFlags.Public | BindingFlags.NonPublic |
            BindingFlags.Instance;

        #if BindAndInvokeTogether

        //构造一个类型的实例
        Object[] args = new Object[] { 12 };        // 构造器参数
        Console.WriteLine("x before constructor called: " + args[0]);
        Object obj = t.InvokeMember(null,
            bf | BindingFlags.CreateInstance, null, null, args);
        Console.WriteLine("Type: " + obj.GetType().ToString());
        Console.WriteLine("x after constructor returns: " + args[0]);
        //读取或者设置一个字段
        t.InvokeMember("someField",
            bf | BindingFlags.SetField, null, obj, new Object[] { 5 });
        Int32 v = (Int32) t.InvokeMember("someField",
            bf | BindingFlags.GetField, null, obj, null);
        Console.WriteLine("someField: " + v);
        // 调用一个方法
        String s = (String) t.InvokeMember("ToString",
            bf | BindingFlags.InvokeMethod, null, obj, null);
        Console.WriteLine("ToString: " + s);
```

```
//读取或者设置一个属性
try {
   t.InvokeMember("SomeProp",
      bf | BindingFlags.SetProperty, null, obj, new Object[] { 0 });
}
catch (TargetInvocationException e) {
   if (e.InnerException.GetType() !=
      typeof(ArgumentOutOfRangeException))
      throw;
   Console.WriteLine("Property set catch.");
}

t.InvokeMember("SomeProp",
   bf | BindingFlags.SetProperty, null, obj, new Object[] { 2 });
v = (Int32) t.InvokeMember("SomeProp",
   bf | BindingFlags.GetProperty, null, obj, null);
Console.WriteLine("SomeProp: " + v);

//注意：InvokeMember 不支持事件

#else

// 构造一个实例
ConstructorInfo ctor = t.GetConstructor(
   new Type[] { Type.GetType("System.Int32&") });
Object[] args = new Object[] { 12 };          // 构造器参数
Console.WriteLine("x before constructor called: " + args[0]);
Object obj = ctor.Invoke(args);
Console.WriteLine("Type: " + obj.GetType().ToString());
Console.WriteLine("x after constructor returns: " + args[0]);

// 读取或者设置一个字段
FieldInfo fi = obj.GetType().GetField("someField", bf);
fi.SetValue(obj, 33);
Console.WriteLine("someField: " + fi.GetValue(obj));

// 调用一个方法
MethodInfo mi = obj.GetType().GetMethod("ToString", bf);
String s = (String) mi.Invoke(obj, null);
Console.WriteLine("ToString: " + s);

//读取或者设置一个属性
PropertyInfo pi =
   obj.GetType().GetProperty("SomeProp", typeof(Int32));
foreach (MethodInfo m in pi.GetAccessors())
   Console.WriteLine(m);
try {
   pi.SetValue(obj, 0, null);
}
```

```
        catch (TargetInvocationException e) {
            if (e.InnerException.GetType() !=
                typeof(ArgumentOutOfRangeException))
                throw;
            Console.WriteLine("Property set catch.");
        }
        pi.SetValue(obj, 2, null);
        Console.WriteLine("SomeProp: " + pi.GetValue(obj, null));

        // 在事件上添加或者移除一个委托
        EventInfo ei = obj.GetType().GetEvent("SomeEvent", bf);
        Console.WriteLine("AddMethod: " + ei.GetAddMethod());
        Console.WriteLine("RemoveMethod: " + ei.GetRemoveMethod());
        Console.WriteLine("EventHandlerType: " + ei.EventHandlerType);

        ThreadStart ts = new ThreadStart(Main);
        ei.AddEventHandler(obj, ts);
        ei.RemoveEventHandler(obj, ts);

        #endif
    }
}
```

在定义 BindAndInvokeTogether 符号的情况下，编译并运行上面的代码，我们将会得到以下输出：

```
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
someField: 5
ToString: 5
Property set catch.
SomeProp: 2
```

注意 SomeType 构造器接受的是一个 Int32 类型的传引用参数。上面的代码演示了怎样调用这样的构造器，以及怎样在构造器返回之后查看改变后的 Int32 值。

删除 BindAndInvokeTogether 符号定义，然后编译并运行上面的代码，我们将会得到以下输出：

```
x before constructor called: 12
Type: SomeType
x after constructor returns: 24
someField: 33
ToString: 33
Void set_SomeProp(Int32)
Int32 get_SomeProp()
Property set catch.
```

```
SomeProp: 2
AddMethod: Void add_SomeEvent(System.Threading.ThreadStart)
RemoveMethod: Void remove_SomeEvent(System.Threading.ThreadStart)
EventHandlerType: System.Threading.ThreadStart
```

20.12　反射一个类型的接口

要获得一个类型所实现的接口集合，我们可以调用 Type 类型的 FindInterface、GetInterface、或 GetInterfaces 方法。所有这些方法返回的都是表示接口的 Type 对象。

确定一个类型的哪些成员实现了某个特定的接口有点复杂，因为多个接口可以定义同样的方法。例如，IBookRetailer 接口和 IMusicRetailer 接口有可能同时定义一个名为 Purchase 的方法。为了得到针对某个特定接口的 MethodInfo 对象，我们可以调用 Type 类的实例方法 GetInterfaceMap。该方法返回一个 System.Reflection.InterfaceMapping(一个值类型)实例。表 20.8 列出了 InterfaceMapping 类型定义的四个公有字段。

表 20.8　InterfaceMapping 类型定义的公有字段

字段名称	数据类型	描　述
TargetType	Type	调用 GetInterfaceMapping 方法的类型
InterfaceType	Type	传递给 GetInterfaceMapping 方法的接口类型
InterfaceMethods	MethodInfo[]	数组中的元素提供了接口方法的一些信息
TargetMethods	MethodInfo[]	数组中的元素提供了实现对应接口方法的类型方法的一些信息

InterfaceMethods 数组和 TargetMethods 数组之间是相互对应的。也就是说 InterfaceMethods[0]标识的 MethodInfo 对象反映了接口中定义的成员信息，TargetMethods[0]标识的 MethodInfo 对象则反映了 TargetType 中定义的对应接口成员的信息。来看下面一个示例应用程序。

```
using System;
using System.Reflection;

// 定义两个接口以备测试
public interface IBookRetailer : IDisposable {
   void Purchase();
   void ApplyDiscount();
}
public interface IMusicRetailer {
   void Purchase();
}

// 下面的类实现了本程序集中定义的两个接
// 口以及另外一个程序集中定义的一个接口
class MyRetailer : IBookRetailer, IMusicRetailer {
   public void Purchase() { }
   public void Dispose() { }
   void IBookRetailer.Purchase() { }
   public void ApplyDiscount() {}
   void IMusicRetailer.Purchase() { }
}

class App {
   static void Main() {
      // 查找本程序集中定义的、MyRetailer
      // 实现的接口，首先创建一个指向筛
      // 选器方法的委托实例，然后将其传
      // 递给 FindInterfaces 方法
      Type t = typeof(MyRetailer);
      Type[] interfaces = t.FindInterfaces(
         new TypeFilter(App.TypeFilter),
         Assembly.GetCallingAssembly().GetName());
      Console.WriteLine("MyRetailer implements the following " +
         "interfaces (defined in this assembly):");

      // 显示每个接口的信息
      foreach (Type i in interfaces) {
         Console.WriteLine("\nInterface: " + i);

         // 获取那些映射了接口方法的类型方法
         InterfaceMapping map = t.GetInterfaceMap(i);

         for (Int32 m = 0; m < map.InterfaceMethods.Length; m++) {
            // 显示接口方法的名称以及
            // 实现接口方法的类型方法
```

```
                Console.WriteLine(" {0} is implemented by {1}",
                    map.InterfaceMethods[m], map.TargetMethods[m]);
            }
        }
    }
    // 下面的筛选器委托方法接受一个类型和一个对象，然后执行
    // 一些检查，如果类型应该包括在 Type.FindInterfaces
    // 返回的 Type 数组中，那么方法将返回 true
    static Boolean TypeFilter(Type t, Object filterCriteria) {
        // 如果接口定义在由 filterCriteria 标识
        // 的程序集中，那么方法将返回 true
        return t.Assembly.GetName().ToString() ==
            filterCriteria.ToString();
    }
}
```

20.13　反射的性能

一般情况下，使用反射来调用方法、或者访问字段和属性是比较慢的，这有以下几个原因：

- 在查找期望的成员时，绑定操作会导致许多次的字符串比较。
- 在传递参数时，我们要首先构造一个数组并初始化其中的元素。然后在内部方法调用时，反射机制又要从数组中提取参数，并将它们放到堆栈上。
- CLR 必须检查被传入一个方法的参数个数和对应的类型都正确无误。
- CLR 必须确保调用者有适当的安全许可来访问成员。

基于以上原因，我们最好避免使用反射来访问类型成员。另外，如果我们正在编写的应用程序需要动态地定位和构造类型，我们应该遵循以下几点原则：

- 让类型继承自一个编译时已经确定的基类型。然后在运行时构造一个该类型的实例，并将其引用存放在一个声明类型为其基类型的变量中(如果大家选择的语言要求的话，可以做适当的转型)，最后调用基类型中定义的虚方法。
- 让类型实现一个编译时已经确定的接口类型。然后在运行时构造一个该类型的实例，并将其引用存放在一个声明类型为其接口类型的变量中(如果大家选择的语言要求的话，可以做适当的转型)，最后调用接口类型中定义的方法。相对于上面使用基类型的做法，我个人更喜欢使用接口的方式，因为我们选择的基类型并不是在所有的情况下都能工作的很好。

- 让类型实现的方法的名称和原型匹配一个编译时已经确定的委托类型。(译注：这里的说法是不正确的，只需要方法的原型和委托类型相匹配就可以了，无需匹配方法名称。)然后在运行时构造一个该类型的实例，并用该实例对象和方法的名称构造一个委托类型的实例，最后通过委托对象来调用期望的方法。这种技巧在三种技巧中需要的工作量最多，尤其在我们需要对一个类型的多个方法进行调用时，工作量会急剧上升。另外注意，通过委托来调用一个方法比直接调用一个类型或接口的方法要慢一些。

在使用上面三种技巧的时候，强烈建议大家将基类型、接口类型、或委托类型定义在自己的程序集中，以减少可能出现的版本问题。要了解这种做法的更多信息，可参见本书第15章第2节“设计支持插件组件的应用程序”。

索 引

有关本索引的反馈，请发电子邮件至：*mspindex@microsoft.com*

Symbols and Numbers

-= operator
  overloaded for instances of delegate types, 383
  unregistering delegates with an event, 236
& (ampersand), suffixing the name of a type in a GetType method, 538
!= operator
  comparing two delegate objects, 377
  overload provided for, 258
  strongly typed operator overloads for, 159
!= symbol for inequality in C#, 191
Of Exceps Thrown/Sec counter, 428
Of Exceps Thrown counter, 428
Of Filters/Sec counter, 428–29
Of Finallys/Sec counter, 429
@ symbol, identifying a verbatim string, 255
@ sign, prepending response files in the CSC.exe command line, 88
[] operator, overloading in C#, 220, 224
[ ] (square brackets), placing custom attributes within in C#, 346–47
^ symbol in C# and in Visual Basic .NET, 191
{ } (braces), specifying format information within, 281
+ symbol
  applying to primitive numbers in C#, 190–91
  applying to strings in C#, 191
+= operator
  appending objects to a linked list of delegates, 369
  in C+, 235
  with the checked statement in C#, 133
  overloaded for instances of delegate types, 383
+ operator
  concatenating strings, 254–55
  using on nonliteral strings, 255
<, > (angle brackets), applying custom attributes in Microsoft Visual Basic, 347
<> symbol for inequality in Visual Basic .NET, 191
== operator
  overload provided for, 258
  in ReferenceEquals, 161
  strongly typed operator overloads for, 159
  using instead of calling Object's ReferenceEquals method, 161
2-D graphics, 23
32-bit number, advantages over, offered by the exception handling mechanism, 395
32-bit status values, not returned in the .NET Framework, 396
64-bit hash of a public key, 76
64-bit PE file format, 11

A

Abort method, 405
abstract C# attribute
  for methods, 176
  for types, 174
Abstract CLR attribute
  for methods, 176
  for types, 174
abstract types, inheritance as, 326
abstract Unicode characters, 267, 268–69
abstracted view of an implied assumption, 423
accessibility modifiers, 173–74
accessor methods, 216
  explicitly defining for events, 242
  name used in C# for, 222
Activator class, 542
add accessor method, 235, 236–38
Add An Application To Configure link, 68
add and remove accessor methods, 242
add and remove methods for events, 233
Add Counters dialog box, 506
add instruction
  in CLR, 131–32
  in IL, 19
Add Reference dialog box in Visual Studio .NET, 52–53
AddEventHandler method, 549
AddHandler method, 242
/addmodule switch with the C# compiler, 49, 50
add.ovf instruction in CLR, 132
address spaces, 20
AdjustToUniversal bit symbol, 288
administrative control, 64–69, 101–8
/Adv switch with ILDasm, 41
advanced menu items on the View menu, 41

AL.exe, 53–55
/algid switch, 77
/company switch, 59
/copyright switch, 59
/culture switch, 62
/delay switch, 91
/description switch, 59
/fileversion switch, 58
/keyfile switch, 79, 91, 107
/keyname switch, 92
/link switch, 55
/linkresource switches, 107
/main switch, 55
/out switch, 59, 107
/product switch, 59
/productversion switch, 58, 59
/target switch, 58
/target:exe switch, 55
/target:winexe switch, 55
/trademark switch, 59
/version switch, 58, 59, 107
/win32icon switch, 56
/win32res switch, 56
correcting a bug in a version number, 61
creating a publisher policy assembly file, 106–7
algorithms
for garbage collection, 453, 455–59
hash, 76–77, 163
intellectual property protection for, 12
aliases, 124
AllowCurrencySymbol bit symbol, 287
AllowDecimalPoint bit symbol, 286
AllowExponent bit symbol, 286
AllowHexSpecifier bit symbol, 287
AllowInnerWhite bit symbol, 288
AllowLeadingSign bit symbol, 286
AllowLeadingWhite bit symbol, 286, 288
AllowMultiple property, 351–52
AllowParentheses bit symbol, 286
AllowThousands bit symbol, 286
AllowTrailingSign bit symbol, 286
AllowTrailingWhite bit symbol, 286, 288
AllowWhiteSpace bit symbol, 289
AmbiguousMatchException exception, 356
ampersand (&), suffixing the name of a type in a GetType method, 538
angle brackets (<, >), applying custom attributes in Microsoft Visual Basic, 347
ANSI C++, 459. *See also* C++
Any symbol for a NumberStyle bit combination, 287
APL, 13
App object, 370
App type, 44
AppDomain boundaries, 513–15
AppDomain type, 542
AppDomainRunner sample application, 515, 532–34
AppDomains, 21, 510–13
accessing from multiple, 264
characteristics of, 511
creating additional, 511
events exposed by, 515
explicitly loading assemblies into, 525–32
identifying for a thread's current execution, 514
individually securing and configuring, 511
isolation of, 511
pulling several versions of assemblies into, 333
reflecting over all assemblies in, 523
threads and, 514
unloading, 467, 511, 514, 532
AppDomainSetup class, 511
AppDomainUnloaded Exception exception, 514
AppendFormat method, 274, 281, 282–84
AppendInsert method, 273
application developers, 416, 424, 432
Application SafeMode entry, 111
application-specific namespaces, 24
ApplicationBase configuration for an AppDomain, 511
ApplicationException type, 410, 411
ApplicationHistory subdirectory, 109, 111
ApplicationName configuration setting for an AppDomain, 511
applications
administrative control over, 64–69
architecting to use events, 228
binding to assemblies, 90
building, 37–38, 508
debugging in Visual Studio .NET, 448–50
deployment of, 63–64
design of, 412
designing to support plug-in components, 331–33
hosting the CLR and managing AppDomains, 516–17
installation complexities, 36, 37
performance of managed, 18
recording assemblies loaded for, 109
repairing faulty, 109–12
running multiple, in a single process, 21, 512
running under debuggers, 456
types of, 38
walking reflection's OM, 540–41
Applications node of the Microsoft .NET Framework Configuration tool, 68
apply attribute of the publisherPolicy element, 103

appobj object, 370
appobj.FeedbackToFile, 370
arbitrary arrays, redimensioning, 323–24
arbitrary bounds, 318
architecture of the .NET Framework, 3
args parameter of InvokeMember, 545
ArgumentException exception, 413, 549
ArgumentException type, 410
ArgumentOutOfRangeException exception, 273, 304, 422
ArithmeticException type, 410
array of bytes, 292
array of characters, 292
array of polygons, 311
Array.Copy method, 315
ArrayList object, 141
arrays, 309–12
  accessing with nonverifiable (unsafe) code, 319–21, 323
  accessing with type-safe code, 321–23
  casting, 315–16
  containing enumerated type symbolic names, 302–3
  copying elements and sections from one to another, 314
  creating, 309, 318
  creating instances of, 314, 543
  fast access to, 319–23
  implicitly derived from System.Array, 312
  overhead information associated with, 311
  passing and returning, 316–17
  redimensioning, 323–24
  required to be zero-based, 310
as operator, 120
ASCII encoding, 290
ASCIIEncoding class, 292
_asm keyword, 32
ASP.NET
  receiving notification of unhandled exceptions, 440
  tracing options offered by, 441
  Web applications, 517
ASP.NET ISAPI DLL, 516
ASP.NET Web Forms
  configuration files for, 67
  garbage collector with, 498
  unhandled exceptions and, 440–41
ASP.NET XML Web services, 441
ASPNet_wp.exe, 516
assemblies, 7, 38, 45, 47
  adding modules to, 49
  adding using the Visual Studio.NET IDE, 52–53
  backward compatibility testing and, 108
  binding applications to, 90
  breaking up the deployment of files, 8
  building multifiles, 49–52
  building with only a public key, 91
  building with reference to other strongly named assemblies, 87–89
  characteristics of, 45
  combining managed modules into, 7–9
  combining modules to form, 45–46
  combining types from different programming languages, 47
  consisting of multiple files, 46
  containing application types, 332
  coordinating the side-by-side execution of, 98
  creating with AL.exe, 53
  cultures of, 61–62
  deploying, 63–64, 71, 73, 95–96
  determining types defined by, 520–22
  distinguishing by company, 74
  dragging and dropping into the Explorer's window, 84
  embedding Win32 resources into, 56
  executing code for, 11–21
  historical record of, 110
  identifying, 48, 73, 74
  included in the FCL, 21
  including resource files in, 55–56
  kinds of, 9, 72
  loading as data files, 527–29
  loading explicitly, 525–32
  loading from different Web sites, 517
  loading into AppDomains, 512
  loading multiple versions into the GAC, 86
  locating, 511
  packaging, 63
  privately deployed, 63–64
  programmatically distinguishing between, 125
  purpose of, 46
  rebasing, 93
  referencing MSCorLib.dll, 101
  reflecting over all in an AppDomain, 523
  registering in the GAC, 83
  relationship to namespaces, 126
  self-describing, 8–9
  signing, 76, 77, 92
  strong name for, 73–79
  types of, 9
  unloading explicitly, 532–34
  using types across, 332
  version numbers associated with, 60–61
  version resource information in, 56–61
assembly, marking methods or fields as, 25

Assembly accessibility in CLR, 173
assembly culture tags, 61
assembly files
installed into the CLR directory and into the GAC, 87
probing for, 66
assembly identity, 526
assembly information, displaying for types, 126
assembly language, writing IL in, 12
Assembly Linker utility. *See* AL.exe
assembly load information, 109
Assembly type, 526
AssemblyAlgIDAttribute attribute, 77
[assembly:CLSCompliant(true)] attribute, 28
AssemblyCultureAttribute custom attribute, 62
AssemblyDef table, 48
full public key always stored, 79
information for a weakly named assembly, 79
version number stored in, 60–61
AssemblyFileVersion version number, 60
assemblyIdentity element, 102, 103
AssemblyInfo.cs file
automatically created in Visual Studio .NET, 58
correcting an error in, 61
created by Visual Studio .NET, 95
AssemblyInformationalVersionAttribute version number, 60
AssemblyKeyFileAttribute attribute, 76, 91
AssemblyKeyNameAttribute attribute, 92
AssemblyLoad event, 515
AssemblyName class, 75
AssemblyRef table, 40, 49, 78
referencing assemblies without an extension, 99
version number embedded in, 61
AssemblyResolve event, 515
AssemblyVersion, 58, 60–61
AssemInfo.cs file, 93–95
assumptions
for calling any method, 404
of a generational garbage collector, 493–94, 495
implied, 402–5
Attach to Process dialog box, 449
attribute instances, 359–62
attribute objects, 355
attribute types, 357
attributes. *See also* custom attributes
applying, 347
applying more than once, 351
applying multiple to a single target, 349
checking fields, 359
defining custom, 345–49, 349–52
inheriting, 352
of strongly named assemblies, 74
AttributeTargets enumerated type, 351
AttributeUsageAttribute type, 350–51
Authenticode technology, 90
AuxFiles subdirectory, 65

B

background thread, 504
backing fields, 218
backslash characters, treatment of, 255
backward compatibility, not having to maintain, 97
backward compatibility testing, assemblies and, 108
base-64 string, 298
base class's constructor, 182
base interface, 553
base types
casting to, 117, 118
inheritance from, 326
vs interfaces, 330
selecting for exception types, 412
versioning issues when adding or modifying members of, 210–11
base.Equals, 157
BaseType property, 537
before-field-init semantics, 188
beforefieldinit metadata flag, 188
behavior of types, 26
BinaryReader type, 290
BinarySearch method, 314, 425
BinaryWriter type, 290, 484–85
BindAndInvokeTogether sample code, 549–53
Binder base type, 544
Binder-derived type, 544
binding, performed by InvokeMember, 544
binding policy, 101–8
Binding Policy tab, 105
BindingFlags enumerated type
with DefaultBinder, 546
with InvokeMember, 547
passing the binder's methods to, 545
search symbols defined by, 523–24
bindingRedirect element, 102, 103
bit flag enumerated types, 306
bit flags, 305–7
bit symbols
defined by DateTimeStyles, 288
defined by the NumberStyles type, 285–87
BitArray type, 220–21
bits, emitting pseudo-custom attributes in metadata as, 362
BodyName property of Encoding-derived classes, 293
BOMs. *See* byte order marks
bool primitive type, 128
Boolean Dispose method, 475, 476
_box operator in C++ with Managed Extensions, 144

boxed form for value types and reference types, 138
boxed value types
  changing fields in, 333–36
  lifetime compared to an unboxed, 142
boxing, 141–42, 341
  effect of unnecessary, on performance and memory usage, 148
  examples of, 145–52
  getting rid of unwanted, 341
  reducing for value types, 341
  value types instances, 138
braces { }, specifying format information within, 281
Break Into The Debugger options, 447
bug fixes, deploying, 108
bugs
  detecting during development and testing, 394
  related to exception handling in the FCL, 424–26
build of an assembly, 60
built-in numeric types, 277
byte order marks (BOMs), 292, 296
byte primitive type, 128
bytes, 296–98

C

C , handling of overflows, 131
C#, 13
  applying custom attributes, 346–47
  casting with the is and as operators, 119–21
  catch block for non-CLS-compliant exceptions, 399
  constructing a string object, 253–54
  developing a standardized version of, 14
  handling of overflows, 132
  namespaces in, 125
  operators defined by, 190–91, 191–92
  primitive types in, 177
  requirement for method calls to specify out or ref, 203
  rules for casting primitive types, 130
  special syntax for entering literal strings into source code, 254
  support for conversion operators, 200
  support for delegate chains, 383
  support of jagged arrays, 311
  terms for accessibility modifiers, 173
C# compiler
  automatic referencing of MsCorLib.dll, 38
  command-line switches producing assemblies, 49
  knowledge of primitive types, 129–31
  multiple response files supported by, 88
  source code constructs for an event, 232–35
  treatment of enumerated types, 300
  using directive, 122
C format for currency in numeric types, 277
C-runtime heap
  compared to the managed heap, 453
  memory allocation compared to managed heap allocation, 454
C++. *See also* unmanaged C++
  accessing existing unmanaged code, 7
  destructor, 463
  Exceptions selection, 445
  handling of overflows, 131
  unmanaged modules produced by the compiler, 7
C++ with Managed Extensions
  _box operator and dynamic_cast operator, 144
  throwing non-CLS-compliant exceptions, 399
  unboxing without copying fields, 144
cabinet (.cab) files
  packaging a strongly named assembly in, 82
  packaging and installing assembly files, 63
call IL instruction, 99, 209
callback functions, 365, 366
callback methods
  in the .NET Framework, 365
  calling, 368
  checking for the names of, 373
  invoking, 374
  registering in ASP.NET, 440, 441
  registering with the DomainUnload event, 190
callvirt IL instruction, 209
CAN-DO relationship, 330, 331
"Cannot access a closed file" message, 480
"cannot explicitly call operator or accessor" error, 236
capacity field, 271–72
Capacity property of StringBuilder, 273
carriage returns, hard-coding into strings, 254
case-sensitive comparisons
  with Compare, 259
  of strings, 257
casting, 251, 252
  arrays, 315–16
  to base types, 117, 118
  to derived types, 118
  detected by the C# compiler, 199
  exposing to the developer, 117–18
  programming patterns supported by compilers, 130
"catch all" filter funclet, 430
catch blocks, 398–400
  catching exceptions that shouldn't be caught, 424–25
  choices at the end of, 399–400
  rethrowing StackOverflowException exceptions, 405
  searching from top to bottom, 398
  using too often and improperly, 418–19
"catch filter funclet", 429–30, 431

catch filters, 429–32
complex, 430–31
specifying exception variables in C#, 399
.cctor, 189
chain, linking delegates in, 369
chaining, delegate support for, 377–83, 384–86
Change method, 335
ChangeType method, 130
Char instances, converting to numeric types, 251–52
char primitive type, 128
Char type, 249, 250
character array field, 272
character array in StringBuilder, 271
characters, 249–52
converting to lowercase or uppercase equivalents, 250
encoding/decoding streams of, 296–98
examining in a string, 266–69
returning the numeric equivalent of, 250
Chars as the name of String's indexer, 224
Chars method, 266
Chars property of StringBuilder, 273
/checked- command-line switch with the C# compiler, 133
/checked+ command-line switch with the C# complier, 132
checked operator in C#, 132, 133
checked statement in C#, 132–33
class constructors. *See* type constructors
class libraries, reflection used for, 519
class library developers, 416, 424
classes
adhering to the dispose pattern, 475
declaring reference types as, 137
documenting assumptions in the development of, 403
indicating a reference type with documentation, 135
cleanup, forcing at a deterministic time, 480
cleanup code
ensuring the execution of, 393, 417
keeping in a localized location, 393
cleanup operations, 400
Clear method of System.Array, 314
CLI (Common Language Infrastructure), 14
client code, catching an exception thrown by a server, 433
client-side applications, graceful recovery for, 432
Clone method, 165
for copying strings, 270
implementing, 166
of System.Array, 314
cloning objects, 164–66
Close method
calling explicitly, 482–85
discouraging the use of, 482
not officially part of the dispose pattern, 480
offered by OSHandle, 475
reasons for calling, 482
CloseHandle, calling with Finalize, 460
CLR (common language runtime), 4
accessibility modifiers defined by, 173
accessing all facilities of, 13
allocating an internal data structure, 15–16
arithmetic operations performed by, 131
arrays supported by, 309
assemblies and, 45
assumption of no bugs in, 404
calling type constructors, 188
checking casting operations at runtime, 118–19
checking the registry for DbgJITDebugLaunchSetting value, 437
complex filters supported by, 430, 431
constructing String objects, 254
controls in the layout of a type's fields, 140
default binding policy, overriding, 101–8
detecting internal data structure corruption or bug, 405
detecting the shut down of, 468
deterministic destruction not supported, 463
enumerated types in, 300
events within, 227–28
hijacking threads, 501
historical record of assemblies, 110
hosting, 508–10
IL instructions for calling a method, 209
implemented as a COM server contained inside a DLL, 508
inheritance supported by, 325
interoperability scenarios supported by, 31–32
kinds of properties, 215
language integration permitted by, 27
loading, 9–11
loading the last-known good set of assemblies, 111
locating manifest assemblies, 100
management of exceptions, 432
metadata produced, 171
namespaces and, 122
operators and, 190
overloading methods based on out and ref parameters, 203
passing parameters by reference, 201
presenting via a language perspective, 13
primitive value types, 316
properties in, 217
resetting the starting points for exceptions, 442

resolving referenced types, 99
resources allocated from the managed heap, 453
response to unhandled exceptions, 437–38
searching up the call stack, 398
shutting down, 467
spawning the debugger, 438
specifying how languages expose operator overloads, 191
string interning mechanism, 262–66
String type tightly integrated with, 256
System.Decimal type not treated as a primitive type, 134
telling not to use the concurrent collector, 504
terms for accessibility modifiers, 173
threshold selected for generation 1, 495
threshold size for generation 0, 494
throwing exceptions involving implied assumptions, 404
transitioning unmanaged threads into, 434
treatment of parameterless and parameterful properties, 221–22
unhandled exceptions for different threads, 437, 438
unloading an AppDomain, 467
value types with parameterless constructors allowed by, 186
verifying the validity of indexes into arrays, 312
version policy settings in the registry, 510
walking up the thread's call stack, 394
CLR COM server
no more than one in a Windows process, 510
two versions of, 509
/clr command-line switch, 32
CLR-compliant programming language, 27
CLR/CTS, set of features offered, 27–28
CLR exceptions, displaying by namespace, 446
CLR header
in a managed module, 5
in a PE file, 38–39
CLS (Common Language Specification), 27–31
C# primitive types compliant with, 128
requirement for zero-based arrays, 310
rules, 28–31
CLS-compliant exceptions, 399, 406
CLS-compliant method names, 191–92
CLS-compliant types, 28
code
debugging different kinds of, 449–50
inlining for accessor methods, 219
leveraging existing with IComparable.CompareTo, 340
reasons for failure, 393
safe, 20
code access security, 37, 547
code explosion, 183
code pages, coding 16-bit characters into arbitrary, 291
code values, comparing characters in the string, 257
code verification process, 6
codeBase element, 46
loading strongly named assembly files from, 90
URL identified by, 96
in the XML configuration file, 102–3
Codebases tab, 105
CodePage property of Encoding-derived classes, 293
COFF header, 5
Collect methods of GC, 467, 499
collection
defining a protected instance field to reference, 241
of event/delegate pairs, 239
implementing via a hash table, 241
collection classes, 331
collection-related interfaces, 331
Collections namespace, types in, 22
COM code, interoperating with unmanaged, 407
COM interface, defined for the CLR, 508–9
COM methods, 394
COM objects, 330
COM tab on the Add Reference dialog box, 53
Combine methods, 233, 376, 378, 383
comma-separated multiple attributes, 349
command-line switches. *See also* AL.exe; SN.exe
creating a publisher policy assembly file, 107
setting version resource fields, 58
Comments field, 59
common initialization constructor, calling, 183–84
Common Language Infrastructure (CLI), 14
common language runtime. *See* CLR
Common Language Runtime check box, 449
Common Language Runtime Exceptions node, 448
Common Language Runtime Exceptions selection, 445, 446
Common Language Specification. *See* CLS
common metadata format, 171
Common Object File Format header, 5
Common Type System. *See* CTS
/company switch of AL.exe, 59
CompanyName field, 59
Compare method
comparing strings, 256
determining whether two strings are equal, 258
differences with CompareOrdinal, 258–59
sorting strings, 259
CompareInfo class, 262
CompareInfo object, 259
CompareInfo property of CurrentCulture, 259

CompareOptions enumerated type, 259
CompareOrdinal method, 257, 258–59
CompareTo method. *See also* IComparable.CompareTo method
with Char type, 250
comparing strings, 256
of IComparable, 329
making type-safe, 339
compiler/linker, 10, 11
compiler primitive types, 160. *See also* primitive types
compiler switch for recompiling C++ code, 32
compilers
behind-the-scenes delegate processing, 371
checking assemblies for type definitions, 123
creating assemblies from modules built by different, 53
delegate class defined by, 371
producing managed modules, 7
recognition of common programming patterns, 129–31
support of namespaces, 125
as syntax checkers, 4
targeting CLR, 4
treatment of namespaces, 123
turning managed modules into assemblies, 8
compile-time errors, 339
compile-time type checking, 339
complex catch filters, 430–31
component library applications, 22
components, 35. *See also* types
Concat method, 146
concurrent collections, 504
concurrent garbage collector
configuring an application to use, 505
telling CLR not to use, 504
ConditionalAttribute attribute class, 351
.config extension, 65
configuration files
creating with a gcConcurrent element, 504
for executable applications, 67
opening and parsing information in, 65
placing in application directories, 64–65
configuration settings for an AppDomain, 511
configuration tool, provided in the .NET Framework, 68–69
ConfigurationFile configuration setting for an AppDomain, 511
consistent equality property, 154
consistent implementation, 330
console applications, 22, 516
console user interface. *See* CUI
constants, 169, 177–78, 301
constructor methods, 44
naming, 187
not required for value types, 184
writing to read-only fields, 179
ConstructorInfo object, 542
ConstructorInfo type, 548, 549
constructors, 181
defined by the Exception base type, 413
initializing all value type fields, 186–87
parameters of, 348
provided by the String type, 254
specifying for custom attributes, 353, 354
constructs, generated for events by the C# compiler, 232–33
containers, offered by CSPs, 92
Continue options in the Visual Studio .NET Exception dialog
box, 447–48
control classes, inheritance and, 331
Control type, 238–39
conversion constructors and methods, defining for a type, 197
conversion operator methods, 197–200
conversion operators, 170, 198
conversions
performed by the Copy method, 315–16
supported by DefaultBinder, 545
Convert method of Encoding-derived classes, 296
Convert type, 130, 251, 252
Copy method
converting array elements, 315–16
for copying strings, 270
of System.Array, 314
/copyright switch of AL.exe, 59
CopyTo method, 270, 314
CorBindToRuntimeEx function, 509, 510
_CorDllMain function, 11
core types
converting within, 130
holding an enumerated type's value, 301
in MSCorLib.dll, 38
_CorExeMain function, 10
CorTokenType enumerated type, 52
CPU architecture, independence from, 13
CPU instructions, compiling IL code into native, 16
CPU machine languages, compared to IL, 12
CPUs, optimizing native code for newer, 18
CreateDelegate methods, 387, 543
CreateInstance flag, 547
CreateInstance methods
of Activator, 541–42, 543
of AppDomain, 542
of Array, 314, 318, 323, 543

CreateInstanceAndUnwrap method, 542
CreateInstanceFrom methods
of AppDomain, 542
of System.Activator, 542
CreateInstanceFromAndUnwrap method, 542
Crypto API, 90
cryptographic service providers (CSPs), 92
CSC.exe
/reference command-line switch, 87
automatically incrementing the assembly, 61
combining resources into assemblies, 56
CSC.rsp file, 88
CSPs (cryptographic service providers), 92
.ctor method, 44, 182, 187
Ctrl+M in ILDasm, 41
CTS (Common Type System), 24–26, 27
CUI (console user interface), 38
CUI executable assembly, 49
CUI PE files, 55
culture information associate, 277
culture-neutral assembly, 62
culture-neutral currency, 279
culture parameter of InvokeMember, 546
culture-sensitive information, 277
culture-specific sorting tables, 258
CultureInfo class, 250
CultureInfo object, 256, 257
CultureInfo type, 277, 278
cultures
of assemblies, 61–62
specifying for characters, 250
string sorting and, 259–61
/culture:text switch of AL.exe, 62
currency symbol
international sign for, 279
for a NumberStyle bit combination, 287
in a string, 287
CurrencyDecimalSeparator property, 286
CurrencyGroupSeparator property, 286
CurrencySymbol property, 287
CurrentCell property, 425
CurrentCulture property, 250, 277
CurrentDomain property, 523
custom attribute types, 348
custom attributes, 345, 348. *See also* attributes
applying, 347
defining, 349–52
detecting the case of, 354–59
as instances of types serialized to a byte stream residing in metadata, 354
for setting version resource fields, 57–59
custom formatter, 282–84

D

D format
for decimal in enumerated and numeric types, 277
for long date in the DateTime type, 276–77
d format for short date in the DateTime type, 276–77
data encapsulation, 216
data files
adding to assemblies, 46–47
loading assemblies as, 527–29
data members, 177
data types
allowed for an attribute type's instance constructor, fields, and properties, 353
catch filters as, 430
directly supported by a compiler, 127
writing complex, 484–85
database connections, opening and closing, 472
DataGrid control, 425
DateTime object, 275
DateTime type
formats recognized by, 276–77
overloads of the Parse method, 289
Parse method offered by, 288
DateTimeFormatInfo type, 277–78, 279
DateTimeStyles type, bit symbols defined by, 288
DbgJITDebugLaunchSetting registry value, 437, 438
DbgManagedDebugger registry value, 438
debug builds, turning on the /checked+ switch, 133
/debug command-line switch
in compilers, 443, 444
preventing garbage collection from collecting reference objects, 456
debug dialog box, displaying, 438
debug versions, unhandled exception policy for, 432
DebuggableAttribute custom attribute, 444, 456
debugger
CLR spawning, 438
requesting notification from, 447
debugging
both managed and unmanaged code, 450
exceptions, 445–48
decimal primitive type, 128
decimal-separator character in a string, 286
Decimal type, 134, 200, 280
Decimal values, compiling code using, 134
DeclaredOnly flag, 524, 545
DeclaredType property, 538, 539
Decoder class, 297
Decoder-derived classes, 297
decoding, 289
deep copy, 165, 316–17
default AppDomain, 510

default attributes, 352
default constructor, 181, 314
Default flag, defined by the BindingFlags enumerated type, 524, 545
default global CSC.rsp file, 88–89
default properties, 215, 220. *See also* parameterful properties
Default property of the Encoding class, 292
DefaultBinder internal (undocumented) concrete type, 544, 545
DefaultMemberAttribute attribute, 224
definition language files, 6
definition metadata tables
  applying custom attributes to entries in, 347
  in module metadata blocks, 39–40
  querying, 519
/delay command-line switch of AL.exe, 91
delayed signing, 90–95
DelaySignAttribute attribute, 91
delegate chains, 377–83
  C# support for, 383
  control over invoking, 384–86
  internal representation of, 378, 379
  removing delegates from, 381–83
Delegate class, 375
  Combine and Remove methods of, 376
  merging with MulticastDelegate, 376
  static methods defined to manipulate a linked-list chain of delegate objects, 377–78
delegate fields, making private, 233
delegate keyword, 371
delegate objects
  calling each in a GetInvocationList array, 384–86
  considered to be immutable, 381
  invoking, 379–80
  returning an array of references to, 384
Delegate references, 378
Delegate type, methods defined by, 387
delegate types
  built by the C# compiler, 232
  defining, 231, 242, 372
Delegate.Combine, calling, 383
Delegate.Remove, calling, 383
delegates, 227, 366–68
  calling back instance methods, 370
  calling back static methods, 368–70
  chaining, 377–83
  code for declaring, creating and using, 366–68
  combining into a linked list, 378
  comparing for equality, 377
  creating instances of, 543
  demystifying, 371–75
  exposing a callback function mechanism, 365
  linking to form a chain, 369
  removing from chains, 381–83
  types of, 375
delete operator, not available in CLR, 117
delType parameter in CreateDelegate methods, 387
dependentAssembly element in the XML configuration file, 102, 103
deployment goals for .NET Framework, 36–37
derivation hierarchy, walking up, 355
derived class, overriding the virtual Boolean Dispose method, 476
derived types, 326
  casting to, 118
  giving control over event firing, 231
/description switch of AL.exe, 59
destination AppDomain, 513–14
destructors, 426, 463
deterministic destruction, 463
developers
  application, 416, 424, 432
  class library, 416, 424
  plug-in, 332
development platform, .NET Framework as, 3
Diagnostics namespace, 23
dimensions in an array, returning the number of, 312
directory entry 14, 11
disambiguating types, 125
Dispose method
  calling explicitly, 482–85
  discouraging the use of, 482
  of IDisposable, 235
  implemented as an explicit interface method implementation, 479
  public, parameterless, 475
  reasons for calling, 482
  sealing, 475
Dispose method of FileStream, 479
dispose pattern, 472
  classes adhering to, 475
  defining types implementing, 481
  implementing in the OSHandle class, 472–76
  types implementing, 477–82
disposing parameter, 475–76
DivideByZeroException exception, 422
DLL assembly, 49
DLL file, producing, 54
"DLL hell", 36, 37
DllImportAttribute type, 348
DLLs
  ability to execute side by side, 97
  managed, 11
  from Microsoft or other vendors, 36
  updating, 36

domain-neutral assemblies, 512, 513
DomainUnload event, 190, 515
double primitive type, 128
download cache of a user, 96
Drawing namespace, 23
dynamic-link libraries. *See* DLLs
dynamic memory, 179
dynamic strings, 263
dynamically created strings, 263
dynamic_cast operator, 144
DynamicInvoke method, 245, 387

E

E format in numeric types, 277
early binding, 520
early bound type, 99
East Asian Language files, 261
ECMAScript, 14
ElementIndex property, 269
/embed switch, 55
embedded resources, 48
/embedresource switch, 107
Encoder-derived classes, 297, 298
encoding, 290
Encoding class, 291–92
Encoding-derived classes, 291
  methods of, 296
  properties of, 293–96
EncodingName property, 293
encodings, converting between characters and bytes, 289–98
EndsWith method, 257
EnsureCapacity method, 273
EnterpriseServices namespace, 23
_EntryPoint function, 55
Enum base type, 301–4
enumerated types, 299–304
  applying instances of FlagsAttribute to, 349
  declaring multiple symbols with the same numeric value, 302
  defining at the same level as the requiring class, 304
  defining to identify bit flags, 305
  expressing the set of bit flags that can be combined, 305
  formatting supported by, 277
  reasons for using, 299–300
enumeration, indicating a value type in the documentation, 135
ephemeral garbage collector. *See* generational garbage collector
equality
  comparing delegates for, 377
  implementing for types, 160
  properties of, 154
Equals method
  with Char type, 250
  for comparing strings, 257
  comparing two delegate objects, 377
  of Encoding-derived classes, 296
  implementing, 154–62
  implementing for a value type, 157–60
  implementing for types directly inheriting Object's Equals implementation, 154–56
  implementing for types inheriting outside of Object, 156–60
  implementing incorrectly, 418–19
  overriden by System.ValueType, 139
  static, 155–56
  of StringBuilder, 274
  of System.Object, 116
  of the System.Object type, 153–54
error
  CS0106: The modifier ... is not valid for this item, 340
  CS0111: ... already defines a member..., 223
  CS0117: ... does not contain a definition for ..., 342
  CS0123: The signature of method ... does not match this delegate type, 370
  CS0165: Use of unassigned local variable ..., 202
  CS0171: Field ... must be fully assigned before control leaves the constructor, 187
  CS0515: ... access modifiers are not allowed on static constructors, 188
  CS0568: Structs cannot contain explicit parameterless constructors, 185
  CS0573: ... cannot have instance field initializers in structs, 186
  CS0579: Duplicate ... attribute, 351
  CS1533: Invoke cannot be called directly on a delegate, 374
Error event, 441
error handling, 402. *See also* exception handling
error report, sending to Microsoft over the Internet, 433
errors, exception handling and, 402
escape mechanism for special characters in C#, 254
European Computer Manufacturer's Association. *See* ECMAScript
event definition entry, 233
event/delegate pairs, 239
event handlers, 231
event information, types holding, 230
event members, 228–33
event registration, explicitly controlling, 236–38
EventArgs type, 230, 241–42

EventDef table, 39
EventHandler, prototype of, 231
EventHandlerList type, 246
EventHandlerSet type, 241
  compared to EventHandlerList, 246
  designing, 243–46
  implementation of, 245
EventHandlerType read-only property, 549
EventInfo type, 548, 549
events, 25, 170, 227
  architecting applications to use, 228
  within the CLR, 227–28
  defining, 232–33
  defining members for, 241–43
  defining to translate input into events, 243
  designing to define lots of, 238–43
  designing types that expose, 228–33
  designing types to listen for, 234–36
  explicitly defining, 242
  exposed by AppDomain, 515
  unregistering interest in all, 235
evidence, applied to AppDomains, 511
ExactBinding flag, 546, 549
Exception base type, public constructors defined by, 413
exception classes, 408–11, 411–15
Exception-derived types, 529–32
exception filters, 398, 428–29
exception handling
  benefits of, 393–94
  bugs in the FCL, 424–26
  evolution of, 394–96
  impact on performance, 427
  learning curve, 394
  mechanics of, 396–401
  mechanisms, 395, 396–401
  misconceptions regarding, 402
  overhead added by, 427
  performance impact of, 395
  performance issues related to, 426–29
  performance penalty invoking, 406
  of secured files, 406
exception hierarchies, 410
exception information, sending over the Internet, 437
exception-related counters, 427–29
exception situations, 393–94
exception stack traces, 441–44
Exception Tree sample application, 529–32
Exception type
  displaying all classes derived from, 529–32
  in an exception filter, 398
  never throwing for private methods, 411
exception type hierarchy vs. the namespace hierarchy, 447
exception types
  adding your own, 448
  defined in the MSCorLib.dll assembly, 408–9
  defining, 412, 413–15, 426
  making serializable, 413
  marking as sealed, 412
  naming, 412
exception variables, 399
ExceptionObject property, 437
exceptions, 401–6
  AmbiguousMatchException exception, 356
  AppDomainUnloaded Exception exception, 514
  ArgumentException exception, 413, 549
  ArgumentOutOfRangeException exception, 273, 304, 422
  can't easily be ignored, 395
  catching one and throwing another, 421–24
  debugging, 445–48
  defined, 402
  displaying the number thrown per second, 428
  displaying the total number thrown, 428
  DivideByZeroException exception, 422
  don't have to be detected where they occur, 395
  ExecutionEngineException exception, 404, 405, 411
  FieldAccessException exception, 176
  FileLoadException exception, 90
  FileNotFoundException exception, 46, 65, 66, 90, 526
  FormatException exception, 276
  guidelines for using, 416–24
  IndexOutOfRangeException exception, 312, 319
  indicating the previous, 407
  InvalidCastException exception, 118, 119, 143, 339, 419
  IOException exception, 478
  IsolatedStorageException, 410
  kinds displayed by Visual Studio .NET, 445
  MethodAccessException exception, 176
  MissingMethodException exception, 544
  not used with Win32 and COM APIs, 394
  NullReferenceException exception, 139, 143, 156
  ObjectDisposedException exception, 480–81, 482
  OutOfMemoryException exception, 404–5, 410, 411, 455
  OverflowException exception, 132, 134, 251
  recovering from gracefully, 419–20
  RemotingException exception, 542
  rethrowing the same, 421
  SecurityException exceptions, 406, 547
  StackOverflowException exception, 404, 405, 410, 411

System.Security.VerifierException exception, 20
TargetInvocationException exception, 544
ThreadAbortException exception, 405, 514
throwing in the finally block, 401
thrown by nonprivate methods, 411
TypeInitializationException exception, 188
TypeLoadException exception, 535
unhandled, 432–37
VerifierException exception, 19, 20
Exceptions dialog box, 445
EXE assembly, 9
EXE file, producing, 54
executable applications (EXEs), configuration files for, 67
ExecutionEngineException exception, 404, 405, 411
Exists method, 425
Exit method, 516
ExitProcess function, 516
Expensive objects, managing a pool of, 491–93
explicit casts between primitive types, 130
explicit cleanup, 452–53, 472
explicit conversion operator, 199
explicit interface member implementations, 338–43
explicit interface method implementations, 342, 479
explicit keyword in C#, 199
exponent format, 286
ExportedTypesDef table, 48
extensible applications, 331–33

F

F format
calling the Format method, 306–7
for fixed-point in numeric types, 277
for flags in enumerated types, 277
factorable base class library, 14
failure code, ignoring in Win32, 395
failure of code, reasons for, 393
family, marking methods or fields as, 25, 26
Family accessibility in CLR, 173
Family and Assembly accessibility in CLR, 173
Family or Assembly accessibility in CLR, 173
fatal error CS0009: Metadata file, 52
fatal exceptions, 410
fatal problems, hiding, 419
faulty applications, repairing, 109–12
FCL (.NET Framework Class Library), 21–24, 128
bugs related to exception handling, 424–26
catching one exception and throwing another, 425
collections, 331
encodings supported by, 290–91
EventHandlerList type defined in, 246
exception classes, 408–11
exception types, 422
handling of reflection, 425
interfaces defined in, 326–27
methods, 129, 410
namespaces in, 22–24
obtaining Type objects, 535
predefined attributes shipped with, 346
StackTrace type offered by, 443
type names, 129
types, 128, 193, 276
types produced by, 534
Feedback class, 372, 379
Feedback delegate, 368, 379
feedback parameter of ProcessItems, 368
FeedbackToConsole method, 368–69
FeedbackToFile method, 370
FeedbackToMsgBox method, 369
field copy, following unboxing, 143
field members of types, 215
FieldAccessException exception, 176
FieldDef table, 39
FieldInfo type
read-only properties offered by, 363
using to bind to a member, 548
fields, 25, 169, 178–80
accessibility modifiers for, 173
arranging the layout for a type, 140
changing in a boxed value type, 333–36
encapsulating access to, 216
immutable, 163
initializing for reference type objects, 182–83
making private, 169
marking as internal, 25
options for controlling access to, 25
predefined attributes applicable to, 175
specifying for custom attributes, 353, 354
storing in dynamic memory, 179
volatile, 179
file versioning, 72
FileAttributes type, 305
FileDef table, 48
FileDescription field, 59
FILEFLAGS, 58
FILEFLAGSMASK, 58
FileLoadException exception, 90
FileNotFoundException exception, 46, 65, 66, 90, 526
FILEOS, 58
files
breaking up the deployment of, 8
creating temporary, 478–81
downloading incrementally, 46
explicitly closing, 478–81
identifying in an assembly, 48
rebasing, 93

files, *continued*
renaming, 39
signing, 76
FileStream class, 477–78
FileStream objects, 460
FILESUBTYPE, 58
FILETYPE, 58
FILEVERSION, 58
FileVersion field, 59
/fileversion switch of AL.exe, 58
Final CLR attribute for methods, 176
finalizable objects, 463, 464
finalization, 459–66, 468–71
finalization list, 468–69
interaction with the freachable queue, 470–71
scanned by the garbage collector, 488
Finalize method, 116
avoiding using when designing a type, 463–64
calling, 463
code constructing new objects in, 467
compared to an unmanaged C++ destructor, 463
dedicated thread for calling, 467
dispose pattern and, 472
events causing an object to call, 467
implementing, 460
killing, 467
most common reason to implement, 464
no control over the execution of, 464
order for calling not guaranteed, 464
preventing from being called, 477
speed syntax for defining, 461–62
static not supported by the CLR, 190
thread dedicated to executing, 434
value types and, 139–40
finalized objects, 489
finalizer thread, 434
finally block, 397, 398, 400–401
calling Dispose or Close from, 482–83
displaying the number executed per second, 429
importance of, 416–17
FindInterfaces method, 553
FindTypes method, 535
Fire method, 242, 245
fixed size, indicating for an array, 313
[Flags] attribute, 306, 307
FlagsAttribute attribute, 305–6, 346
FlagsAttribute class, 349
FlagsAttribute constructor, 349
FlagsAttribute type, 349, 355
FlattenHierarchy flag, 524, 545
float primitive type, 128
Float symbol, 287
format information, specifying within braces, 281
Format method
called by the System.Enum.ToString method, 302
of Enum, 354–55
of ICustomFormatter, 283, 284
of String, 280
format parameter of the ToString method, 276
format strings, 277
FormatException exception, 276
formatProvider parameter
of the IFormattable interface's ToString method, 276
of ToString, 277
formatter, providing a custom, 282–84
Framework Configuration administrative tool. *See* .NET Framework Configuration tool
freachable queue, 469, 470–71
Friend accessibility in Visual Basic .NET, 173
friendly method names, 196
friendly operator methods, 196
FromBase64CharArray method, 298
FromBase64String method, 298
fs variable, casting to an IDisposable, 479
fully qualified names, 123
functions, calling, 31, 282–83

G

g format for general date in the DateTime type, 276–77
G format for general in enumerated and numeric types, 277
GAC (global assembly cache), 79, 526
displaying the contents of, with ShFusion.dll, 83
holding multiple versions of assemblies, 86
internal structure of, 85–86
purpose of, 85
GAC directory
example of, 85–86
structure of, 79–80
GACUtil.exe, 80–82
garbage-collected environment, 117
garbage-collected platform, 451–55
garbage collection, 452
algorithm, 453, 455–59
forcing, 499, 500
monitoring, 506
performance kit generated by, 458
performance test for generation 0, 495
preventing from collecting reference objects, 456
garbage collector
causing a beep at each collection, 465
compacting the memory, 488
decisions on compacting memory, 504
dynamically modifying generation thresholds, 498

examining only old objects written to since the last collection, 495
forcing a collection, 499
mechanisms offered by, 503
obtaining roots, 456
performance impact of executing a Finalize method, 463
performance issues, 501–6
programmatic control of, 499–501
running, 488
scanning the finalization list, 488
scanning the long weak reference table, 488
scanning the short weak reference table, 488
thread building the graph of unreachable objects, 504
tracking the lifetime of objects, 6
value type instances not under the control of, 135
walking the roots, 457
walking through reachable objects recursively, 457
garbage objects, contiguous blocks of, 457
GCBeep class, 465
GCBeep objects, 468
GCBeep type, 468
GC.MaxGeneration property, 499
"General format" (G), 277
generation 0, 494, 497
generation 0 is full event, 467
generation 1, 497
objects in, 494
threshold selected for, 495
generation 2, 497
generational garbage collector, 493, 495
generations, 493–98
in a garbage collector, 455
selecting threshold sizes for, 497
supported by the managed heap, 497
get_
prepending by the compiler, 222
prepending to a property name, 219
get accessor method
emitting a method representing the parameterful property, 222
for a property, 218
get and set accessor methods, renaming in C#, 222–23
get method, specifying to define a property, 218
GetAccessors method, 548
GetAddMethod method, 549
GetAssemblies method of AppDomain, 523
GetAttributes method, 305
GetBaseException method, 407
get_Bit method, 223
GetByteCount method, 292, 293, 298
GetBytes method, 292, 298
GetCharCount method
of Decoder, 297
of Encoding, 292, 293
GetChars method, 292, 297
GetConstructors method, 548
GetCustomAttribute method, 355
passing a type to, 357
of System.Attribute, 356
GetCustomAttributes method, 355
defined by the reflection types, 356
passing a type to, 357
of System.Attribute, 356
GetCustomAttributes property, 539
GetDecoder method, 297
GetDomain method, 514
GetEncoder method, 297
GetEncoding method, 292
GetEnumerator method, 267, 313
GetEvents method, 548
GetExecutingAssembly method, 521
GetExportedTypes method, 535
GetField flag, 546, 547
GetFields method, 548
GetFormat method, 278–79
GetGeneration methods of GC, 500
GetGetMethod method, 548
GetHashCode method, 162
for comparing strings, 257
defining, 163
of Encoding-derived classes, 296
overriding along with Equals, 162
overriding by System.ValueType, 139
System.Object's implementation of, 163
System.ValueType's implementation of, 163–64
GetInterface method, 553
GetInterfaceMap instance method, 553
GetInterfaces method, 553
GetInvocationList instance method, 384
GetLastError, calling for Win32 functions, 394
GetLength method, 312
GetLowerBound method, 312, 318
GetMaxByteCount method, 293
GetMaxCharCount method, 293
GetMembers method
returning an array, 521
of the Type object, 523
GetMethods method, 548
GetName method, 303
GetNames method, 303
GetNestedType method, 535
GetNumericValue method, 250–51
GetPreamble method, 296

GetProperties method, 548
GetProperty flag, 546, 547
GetRemoveMethod method, 549
GetSetMethod method, 548
GetString method, 292
GetTextElement method, 269
GetTextElementEnumerator method, 267, 268–69
GetType method, 117
of System.Object, 116, 534
of System.Reflection.Assembly type, 535
of System.Reflection.Module, 535
of System.Type, 535
GetTypeFromHandle method, 537
GetTypes method
returning an array of System.Type elements, 521
of System.Reflection.Assembly type, 535
of System.Reflection.Module, 535
GetUnderlyingType method, 301
GetUnicodeCategory method, 249, 250
GetUpperBound method, 313, 318
GetValue method
of PropertyInfo, 548, 549
of System.Array, 313
GetValues method, 302–3
GetVersionEx method, 348
global assembly cache. *See* GAC
global CSC.rsp file, 88–89
Global.asax file, 441
Globalization namespace, 23
globally deployed assemblies, 71, 73
grouping-separator character in a string, 286
GUI (graphical user interface), 38
applications, 38
executable assembly, 49
PE files, 55
GUIDs, assigned to the COM interface for the CLR, 508

H

handles, obtaining for fields and methods, 537
hash algorithm, overriding the default, 76–77
hash code values, persistence and, 164
hash codes
obtaining for any and all objects, 162
returning for a string, 257
returning for an object's value, 116
selecting an algorithm for calculating, 163
hash table
changing a key object in, 162
implementing a collection, 241
hash value in the AssemblyRef table, 40
Hashtable type, 162, 221
HasShutdownStarted read-only property, 468
HeaderName property of Encoding-derived classes, 293
heap. *See* managed heap
HelpLink property of the System.Exception type, 407
hex digits in a string, 287
hexadecimal numbers, parsing, 287–88
HexNumber symbol for a NumberStyle bit combination, 287
hidden parameter in an indexer's set accessor method, 221
hidden this pointer, received by instance methods, 209
hierarchies, designing for exception types, 412
high-priority run-time thread, dedicated to calling Finalize methods, 470
high surrogates, 267
hijacked threads, 501
historical record of assemblies used by an application, 110
HRESULT, returned by COM methods, 394
HResult property of the System.Exception type, 407

I

I, prefixing the name of an interface type, 327
/I switch with GACUtil.exe, 80, 81, 82
IANA-registered name, 293
IChangeBoxedPoint interface, 335
ICloneable interface
for classes, 165
definition of, 327
implementing, 270
IComparable interface, 329
IComparable.CompareTo method, 340. *See also* CompareTo method
icons, embedding into assemblies, 56
IConvertible interface, 251, 252
IConvertible method, 342–43
ICorRuntimeHost interface, 509, 510
ICustomFormatter interface, 283
.idata section of an assembly file, 10
IDisposable interface, 476, 483
IDL files, 6
IEnumerator, returning for an array, 313
IFormatProvider interface
FCL types implementing, 279
of the IFormattable interface's ToString method, 276
implementing, 278, 283
IFormattable interface, 276
IgnoreCase flag, 524, 545
IL (intermediate language), 11–12
converting to native CPU instructions, 15
instructions, 131–32, 311

instructions for calling a method, 209
intellectual property and, 12
no instructions for manipulating registers, 19
typeless instructions, 19
verification and, 19–21
IL Assembler, 12
IL assembly language, 13
IL code, 5
binding with metadata, 6
compiling into native code with NGen.exe, 19
compiling into native CPU instructions, 16
emitting instead of native X86 instructions, 32
produced by CLR-compliant compilers, 6
for a type constructor method, 189
IL disassembler. *See* ILDasm.exe
ILAsm.exe, 12, 47
ILDasm.exe, 12, 30, 33–34
examining code for the ProcessItems method, 374, 375
examining metadata within a PE file, 41–44
examining metadata's manifest tables, 51–52
examining the metadata produced for the delegate, 371–72
example output from, 98
Locale used instead of culture, 78
obtaining an IL source, 47
output showing metadata, 172
View Statistics menu item, 44–45
IMAGE_COR20_HEADER, 39
immutable assemblies, 332
immutable fields, 163
immutable strings, 249, 255–56
implementation detail, hiding, 421–24
implementation inheritance, 326, 331
implementation token, 52
implicit assumptions of programmatic interfaces, 402–3
implicit casts, performing between primitive types, 130
implicit conversion operator, 199
implicit keyword, 199
implicitly sealed methods, 139
implied assumptions, 404–5
import section of an assembly file. *See* .idata section of an assembly file
Imports statement in Visual Basic .NET, 122
InAttribute type, 348
index checking, performance cost of, 319
index properties, 220. *See also* parameterful properties
IndexerName attribute, applying to a parameterful property, 224
IndexerName custom attribute, 223
indexers, 215, 220. *See also* default properties
in BitArray type, 221
defining multiple with the same signature, 223
exposing in C#, 220
set accessor methods for, 221
syntax for expressing in C#, 222
taking Objects as parameters, 221
indexes into arrays, 312
IndexOf/LastIndexOf method, 267
IndexOf method, 262, 314
IndexOfAny/LastIndexOfAny method, 267
IndexOutOfRangeException exception, 312, 319
inequality (!=), comparing two delegate objects, 377
inherit parameter, not honored by reflector type methods, 357
inheritance, 325
Inherited property, 351, 352
.ini file, 109
initialization constructor, 183–84
initialization of fields for reference type objects, 182–83
Initialize method, 314
InitOnly CLR attribute for fields, 175
inline assembly language, /clr switch and, 32
inline initialization syntax, 180
inlining code, 219
inner exceptions, difficulties in setting, 422
InnerException property, 407, 422
installation complexities of Windows, 36, 37
Instance CLR attribute for methods, 175
instance constructors, 170, 181–87
instance data fields, not using, 241
instance events, 170
instance fields, 170, 178
Instance flag, 524, 545
instance methods, 209, 370
InstanceCallbacks method, 370
instances of a type, 541–43
int primitive type, 128
Int32 array, 208
Int32 hash code, 162
Int32 type
casting to an IConvertible, 343
looking more like a "simple type", 342
Parse method for, 285
Parse method: overloads offered by, 288
Integer symbol for a NumberStyle bit combination, 287
intellectual property protection for algorithms, 12
IntelliSense, parsing metadata, 6
interface inheritance, 326
interface methods, 326
defining parameters and return values for, 332
not declaring as public, 337–38

InterfaceMapping type, 553
InterfaceMethods field, 553
interfaces, 150, 152, 325
vs. base types, 330
changing fields in a boxed value type, 334–35
considered as reference types, 330
contents of, 326
defined in the FCL, 326–27
explicit member implementation, 338–43
flexibility at the cost of compile-time type safety, 338
implemented by the String type, 253
implementing multiple with the same method, 336–38
inheriting the contract of other interfaces, 327
nonstatic members of, 327
reflecting over a type's, 553–56
InterfaceType field, 553
interlanguage operability, 27
intermediate language code. *See* IL code
Intern method, 263–64
internal, marking methods or fields as, 25
internal accessibility in C#, 173
internal data structures in the CLR, 15–16, 405
internal hash table, 262, 263
internal memory, running out of, 405
internal structure of the GAC, 85–86
internal table, created by the JIT compiler, 455–56
internal types, allowed by C#, 125
InternalName field, 59
international sign for currency, 279
Internet, downloading files incrementally from, 46
Internet Assigned Numbers Authority. *See* IANA-registered name
Internet Explorer, 67, 517
interoperability, scenarios supported by the CLR, 31–32
InvalidCastException exception, 118, 119, 143, 339, 419
invariant culture, 258, 279
InvariantCulture property, 258, 279
Invoke method
of ConstructorInfo, 542, 549
of the delegate object, 374
of the Delegate type, 379, 384
of MethodInfo, 425, 549
InvokeMember instance method
actions performed by, 546
binding and invoking a member, 549–53
of Type, 542, 543–48
InvokeMethod flag, 547
IO namespace, types in, 23
IOException exception, 478
IOException type, 410
/IR switch of GACUtil.exe, 80, 82
IS-A relationship, 330, 331
is operator
casting in C#, 119–20
checking the type of obj, 160
IsBrowserDisplay property, 293
IsBrowserSave property, 293
IsDefined method, 355
of Attribute, 356
of Enum, 304
passing a type to, 357
of Type, 355
IsDefined property, 539
IsFinalizingForUnload method, 468
IsFixedSize property, 313
IsInterned method, 263, 264–66
isJITOptimizerDisabled parameter, 444, 456
IsLastIndexOf method, 262
IsMailNewsDisplay property, 293
IsMailNewsSave property, 293
IsolatedStorageException, 410
isolation of AppDomains, 511
IsPrefix method, 262
IsReadOnly property, 313
IsSuffix method, 262
IsSynchronized property, 313
IsTerminating property, 437
Item property with a type, 222

J

jagged arrays, 309, 311
Japanese characters, viewing, 261
Java operator, 193
JIT (just-in-time) compiler, 15
forbidding from inlining methods, 444
inlining methods, 443
internal table created by, 455–56
JITCompiler function, 16
jitDebugging configuration setting, 440
JMP instruction of the stub function, 10
JScript, 14

K

key for each collection entry, 241
key object, changing in a hash table, 162
key/value pair, adding to a Hashtable object, 162
key words. *See* primitive types
/keyfile command-line switch of AL.exe, 79, 91, 107
/keyname switch of AL.exe, 92
killing an application, 439
"known to be safe" policies, 73

L

language compilers. *See* compilers
languages. *See* programming languages
Languages tab of the Regional And Language Options dialog box, 261
large object heap, 505
large objects, allocating, 505
LastIndexOf method, 267, 314
LastIndexOfAny method, 267
late binding, 520
Launch method, 437
LayoutKind.Auto for reference types, 140
LayoutKind.Sequential for value types, 140
ldelem instruction in type-safe IL code, 323
ldind.i4 instruction in unsafe IL code, 323
ldstr (load string) IL instruction, 254
ldtoken IL instruction, 536, 537
leading space, allowing Parse to skip over, 285
LegalCopyright field, 59
LegalTrademarks field, 59
Length method, 266
Length property
  of StringBuilder, 272, 273
  of System.Array, 312
letter casing as a culture-dependent operation, 250
/lib command-line switch of CSC.exe, 87
LIB environment variable, 87
"lightweight" types, 135
/link switch of AL.exe, 55
linked list
  adding and removing delegates from, 236
  allowing delegate objects to be part of a, 377
  of delegates, 232, 369
/linkresource switches
  of AL.exe, 107
  of the C# compiler, 56
listeners, 170, 232
literal strings
  adding to the internal hash table, 262–63
  expressing directly in source code, 253
  syntax for entering into source code in C#, 254
  writing to metadata once, 266
literals, 130–31
Load method, 526, 527
LoadAssemblies method, 529
loader heap in an AppDomain, 512
LoaderOptimization configuration setting for an AppDomain, 511
LoadFrom method, 526
  of Assembly, 542
  changing to Load when possible, 529
LoadLibrary function
  calling to load a managed assembly, 9
  similar technique to, 526
LoadWith PartialName method, 526–27
local CSC.rsp file, 88
Locale, used by ILDasm.exe instead of culture, 78
localized location for cleanup code, 393
lock statement, 417, 475
long primitive type, 128, 129
long weak reference table, 487–88
long weak references, 487, 489
low surrogates, 267
lower bounds for an array, 311, 312
Lucida Sans Unicode font, 269

M

M format for month/day in the DateTime type, 276–77
machine language
  compared to IL, 12
  CPU-independent, 11
machine-wide policies, 67
Machine.config file, 67, 103
macros in future productivity applications, 518
MailManager sample application, 227–38
/main command-line switch of AL.exe, 55
Main method, 44
main thread, 434
managed applications
  performance of, 18
  in a single virtual address space, 21
managed assemblies
  creating for unmanaged COM components, 31
  invoking, 11
managed code. *See also* IL code
  allocating types in, 136
  calling unmanaged functions in DLLs, 31
  compared to unmanaged, 18
  debugging, 449
  using an existing COM component (server), 31
managed compilers, 427
managed DLL, loading and initializing the CLR, 11
managed environment, performance ramifications of, 17–18
managed EXE, loading and initializing the CLR, 10
managed heap, 453, 454
  after garbage collections, 458, 471, 496, 497
  allocating reference types from, 134
  arrays of value and reference types in, 309, 310
  assumption of infinite address space and storage, 454–55
  before a collection, 457
  compacting, 458
  generation 1, 494
  long weak reference table, 487

managed heap, *continued*
multiple threads accessing, 501
newly initialized, 494
partitioning into multiple memory arenas, 503
short weak reference table, 487
showing pointers in its finalization list, 469
showing pointers moved from the finalization list to the freachable queue, 470
splitting into several sections by CPU, 503
managed main thread, 434
managed modules, 4
combining into assemblies, 7–9
compilers producing, 7
compiling source code into, 4–5
examining with ILDasm, 30
parts of, 5
turning into assemblies, 47
managed PE files. *See* PE (portable executable) files
Management namespace, types in, 23
manifest, 7, 8, 45, 47, 76
manifest metadata tables, 39, 47–48
descriptions and types included in, 51
producing a small file containing, 54, 55
ManifestResourceDef table, 48, 55, 56
manual threads, 434
MarshalByRefObject, 514
MarshalAsAttribute type, 348
Match method, 359–62
MaxCapacity property of StringBuilder, 273
maximum capacity field, maintained by StringBuilder, 271
maximum generations, supported by the managed heap, 499
MaxValue field in the Char type, 249
MemberAccess bit, 547
MemberInfo-derived types, 538–39
MemberRef table, 40
members
binding to, 548
contained in types, 24–25
defined by types, 169–72
defining for events, 241–43
within a type, 538–52
of types, 28–29
types used to bind to, 548
MemberType property, 539
MemberwiseClone method, 116
calling, 182
of System.Object, 165–66
memcpy function, 458
memory
assumption that enough is available, 404
buffer implemented by the System.IO.FileStream type, 484
compacting, 488
layout differences between reference and value types, 137
reclaiming from objects requiring finalization, 471
running out of internal, 405
memory stream, serializing and deserializing, 317
Message property of the System.Exception type, 407
metadata, 6, 507–8
act of examining, 508
binding with IL code, 6
common format for all CLR languages, 171
examining within a PE file, 41–44
as the key to the Microsoft .NET Framework development platform, 171
parsing, 6
reflecting over an assembly's, 520–23
translating a type with members into, 171–72
writing literal strings to, 266
metadata block in a PE file, 39–41
metadata tables
in managed modules, 5
reflecting over, 518
metadata tokens, 52
method calls, executing across AppDomain boundaries, 514
method definition, emitted by the C# compiler, 191
method names
for C# operators, 191–92
friendly, 196
qualifying with the name of the interface, 337
method parameters, passing by value, 200–201
method parity, not complete for String and StringBuilder classes, 274–75
Method property, 373
method table pointer, 453
MethodAccessException exception, 176
MethodDef table, 39
MethodDef token, 5, 38
MethodImplAttribute attribute, 233, 242, 444
MethodInfo elements, returning an array of, 548
MethodInfo objects, 553
MethodInfo type, 548, 549
_methodPtr field, 372
methodPtr parameter for a delegate, 373
methods, 25, 170, 209
accessibility modifiers for, 173
assumptions for calling any, 404
calling a type's, 543–47
calling for the first time, 15
calling for the second time, 16, 17
calling within a checked operator or statement, 133

for copying strings, 270
defining multiple, 200
defining protected, 231
defining to notify registered objects of events, 242
defining to translate input into events, 231–32, 243
defining with the same name and same parameters, 209
designing to return arrays with zero elements, 317
for displaying the type of objects, 208
of Encoding-derived classes, 296
executing, 15
forbidding from being inlined, 444
implicitly scaled, 139
implied assumptions from calling, 403
inlining by the JIT compiler, 443
loading types referenced by, 52
for manipulating strings, 270
marking as internal, 25
marking as synchronized, 233
offered by Decoder-derived classes, 297
offered by Encoder derived classes, 298
options for controlling access to, 25
overloaded versions of, 149
overloading based on out and ref parameters, 203
passing a variable number of parameters to, 206–8
passing parameters by reference to, 200–206
predefined attributes applicable to, 175
qualifying with the name of the interface, 340
reflection altering the behavior of, 508
of System.Attribute reflecting over metadata, 356
taking an arbitrary number of parameters of any type, 208
MgdUEPolicy method, 436–37
Microsoft, version-numbering scheme used by, 60
Microsoft Common Object Runtime Library. *See* MSCorLib.dll
Microsoft compilers, delegate types derived from
MulticastDelegate class, 376
Microsoft Component Object Runtime Execution Engine. *See* MSCorEE.dll file
Microsoft Internet Explorer, 67, 517
Microsoft .NET Framework. *See* .NET Framework
Microsoft .NET Framework Configuration tool. *See* .NET Framework Configuration tool
Microsoft Office XP applications, unhandled exceptions in, 432–33
Microsoft PowerPoint, Unhandled Exception dialog box, 433
Microsoft SQL Server, next version of, 517
Microsoft T-SQL check box in the Attach to Process dialog box, 449
Microsoft Visual Basic. *See* Visual Basic .NET
Microsoft Windows. *See* Windows
MIME filter, installed by the .NET Framework, 517
MinValue field in the Char type, 249
MissingInterfaceException type, 412
MissingMethodException exception, 544
mixed-language programming, 13
modifiers for an interface definition, 327
module version ID, 39
ModuleDef table, 39
ModuleRef table, 40
modules
adding to assemblies, 49
assemblies referenced by, 40
building types into, 37–45
combining to form assemblies, 45–46
events defined in, 39
fields defined in, 39
identifying, 39
members referenced by, 40
methods defined in, 39
parameters defined in, 39
properties defined in, 39
types defined in, 39
types referenced by, 40
MSCorEE.dll dynamic-link library, 10, 11
MSCorEE.dll file, 9, 509
MSCorEE.h C++ header file, 509
MSCorIE.dll, 517
MSCorLib.dll assembly, 123, 408–9
MSCorLib.dll file, 38, 512
assemblies referencing, 101
System.Console type code in, 37
using interfaces or types defined in, 332
MSCorSvr.dll file, 509
MSCorWks.dll file, 509
MSI files, packaging assembly files into, 63
MSIExec.exe, 63, 82
multicast delegates, 375
MulticastDelegate class
derived from the Delegate class, 375
GetInvocationList instance method, 384
idea of merging with Delegate, 376
public instance properties defined by, 373
MulticastDelegate type, all delegate types derived from, 372
multidimension arrays, 311
multifile assemblies
building, 49–52
command-line tools required for creating, 49
creating with AL.exe, 54
reasons for using, 46–47
Multilanguage Standard Common Object Runtime Library. *See* MSCorLib.dll assembly
multiple attributes, applying to a single target, 349

multiple inheritance, 325
multiple interfaces, 336–38
multiple response files, 88
multiple unmanaged applications, running in a single process, 512
MustInherit Visual Basic .NET attribute for types, 174
MustOverride VisualBasic .NET attribute for methods, 176
mutable strings, 272
MyRetailer class, 553–55

N

N format for number in numeric types, 277
Name property, common to MemberInfo-derived types, 539
named parameters, 348
namespace hierarchy vs. the exception type hierarchy, 447
namespace information, displaying for types, 126
namespaces, 121
  CLR and, 122
  creating, 125
  in the FCL, 22–24
  relationship to assemblies, 126
  treatment by compilers, 123
  using to locate a particular type, 121
National Language Support (NLS), types for, 23
Native check box in the Attach to Process dialog box, 449
native code offsets, 456
native CPU instructions, compiling IL code into, 16
Native Run-Time Checks selection, 445
NativeWindow's internal method, 440
nested types, accessibility modifiers for, 174
.NET Application Restore dialog box, 110, 111
.NET CLR Memory performance object, 506
.NET Framework
  architecture of, 3
  culture-specific sorting tables included with, 258
  dealing with versioning problems, 72
  deployment goals, 36–37
  determining versions installed, 9
  exception-related counters, 427–29
  incredible value provided by, 507
  installing on customers' machines, 9
  metadata as the cornerstone of, 507–8
  representing characters in, 249
  running on top of Microsoft Windows, 508
  standardizing, 14
.NET Framework Class Library. *See* FCL
.NET Framework Configuration tool, 68–69, 104–5, 505
  creating or modifying an application configuration file, 110–12
  turning off verification, 20
.NET Framework reference documentation, 135, 342
.NET Framework SDK documentation, 126
.NET Framework Wizards utility, 110
Net namespace, types in, 23
.NET tab on the Add Reference dialog box, 53
network communications, types for, 23
new C# attribute for methods, 176
new keyword, 213
New method in Visual Basic .NET source code, 187
new operator
  causing the compiler to emit a newobj instruction, 453
  creating objects, 117
  initializing value type instances, 138
  newobj instruction emitted by, 451
  not used to construct a String object in C#, 253
newline characters, hard-coding into strings, 254
NewLine property of the System.Environment type, 254
newobj instruction, 117, 254, 453
  calling, 451
  steps performed by, 453
Newslot CLR attribute for methods, 176
NextObjPtr, 453
NGen.exe tool, 19
/noconfig command-line switch, 89
NoCurrentDateDefault bit symbol, 288
non-CLS-compliant attributes, 356
non-CLS-compliant exceptions
  catching, 399, 421
  not recommended, 426
  throwing, 399
non-reachable objects, 471
non-zero-based arrays, 310
nonabstract attributes, 349
None bit symbol, 286, 288
nonfatal situations, 410
nongarbage objects, shifting down in memory, 458
noninvariant culture, 258
nonliteral strings, using the + operator on, 255
nonprivate methods, validating, 411
NonPublic flag, 524, 545
nonstatic members of an interface, 327
nonstatic methods, 209
nonvirtual methods, 117
nonzero lower bound, casting arrays with, 318
/nostdlib switch with the C# compiler, 38
NotInheritable Visual Basic .NET attribute for types, 174

NotOverridable VisualBasic .NET attribute for methods, 176
null reference type variable, 139
NullReferenceException exception, 139, 143, 156
Number symbol for a NumberStyle bit combination, 287
number types, parsing strings into, 285
NumberDecimalSeparator property of NumberFormatInfo, 286
NumberFormatInfo type, 277, 279
NumberGroupSeparator property of NumberFormatInfo, 286
numbers, formatting with ToString, 277
NumberStyles parameter, 285
NumberStyles type, bit symbols defined by, 285–87
numeric types
  displaying to the user, 275
  techniques for converting to Char instances, 251–52
numeric values
  converting into a string of flags, 306–7
  determining the legality of, 304
  formatting to represent a culture-neutral currency, 279
  formatting to represent Vietnamese currency, 278

O

obfuscator utilities, 12
Object base type, 22
object corruption
  not possible in the garbage collection environment, 458
  special tools designed to locate, 452
object model types, 518
object-oriented machine language, IL as, 12
object pool, designing with resurrection, 491–93
object primitive type, 128
Object type, 26, 153–54
ObjectDisposedException exception, 480–81, 482
ObjectHandle type, 541
objects
  accessing across AppDomain boundaries, 514
  allocating consecutively in memory in the managed heap, 454
  casting to various types, 117
  cloning, 164–66
  collecting short-lived, 255
  constructing static, read-only to identify events, 241
  converting from one type to another, 197
  creating, 117, 453, 490
  deserializing, 514
  deterministically disposing or closing, 472
  equality of, 153–62
  forcing to clean up, 471–82
  formatting multiple into a single string, 280–81
  on the freachable queue, 470
  implementing operations on, 153
  initializing overhead numbers for, 117
  obtaining a string representation for, 275–84
  obtaining an Int32 hash code for, 162
  parsing strings to obtain, 285–89
  pinned, 503
  re-creating on remote machines, 6
  registering interest in events, 232, 233, 235
  remoting by value, 514
  separating into generations, 455
  tracking the lifetime of, 6
  treating as being of multiple types, 325
Office XP applications, unhandled exceptions in, 432–33
one-dimension array, reversing the order of elements in, 314
OnThreadException method
  of Application, 439
  System.Windows.Forms.Control type's implementation of, 439
op_ methods, calling explicitly, 196
op_Addition method
  in C#, 191
  calling explicitly in Visual Basic.NET, 195
  calling in Visual Basic .NET, 193
  offered by a Visual Basic .NET type, 193, 194
operations, backing out of partially completed, 420–21
operator keyword in C#, 199
operator overload methods, 190–96
  compared to conversion operator methods, 198–99
  not defined by FCC types, 193
operator overloads, 170, 191, 193
operators
  defined by programming languages, 190
  interpretation by compilers, 131
  for obtaining a Type object, 535–36
op_Explicit method, 199, 200
op_Implicit method, 199, 200
op_methods in C#, 191–92
optimized native code, 17
OptionalParamBinding flag, 546
OriginalFilename field, 59
OSHandle class
  new and better version of, 472–76
  overriding, 476–77
OSHandle object, cleaning up, 475
OSHandle type, 460–61, 462, 474–75
out keyword
  with reference types, 203–6
  with value types, 201, 203

/out switch of AL.exe, 59, 107
OutAttribute type, 348
OutOfMemoryException exception, 404–5, 410, 411, 455
OverflowException exception, 132, 134, 251
overflows, resulting from arithmetic operations on primitive types, 131
overhead
added by exception handling, 427
information associated with arrays, 311
overhead fields for objects, 453
overhead members, initializing for objects, 117
overloaded versions of methods, 149
Overridable VisualBasic .NET attribute for methods, 175
override C# attribute for methods, 176
Override CLR attribute for methods, 176
override method, changing from virtual to, 214
Overrides VisualBasic .NET attribute for methods, 176

P

P format for percent in numeric types, 277
P/Invoke, 31
-p switch of SN.exe, 91
parallel collections, 503
ParamArrayAttribute custom attribute, 207, 208
ParamDef table, 39
parameter validation, performing for enumerated types, 304
parameterful properties, 215–19, 220–25
exposing in C#, 220
selecting the primary, 224
parameterless constructors, not allowed for value types in C#, 185–86
parameterless properties, 220
parameterless ToString method, 275, 276
parameters
of a constructor, 348
for indexers, 221
passing a variable number of, to a method, 206–8
passing by reference to a method, 200–206
setting fields or properties, 348
ParamName property of ArgumentException, 413
params keyword, 207, 208
parentheses in a string, 286
Parse method, 285
with Char type, 250
of the DateTime type, 288
of Enum, 303, 307
for hexadecimal numbers, 287–88
skipping over a leading space, 285
ParseCombiningCharacters method
manipulating abstract Unicode characters, 268–69
of StringInfo, 267, 269
ParseExact method, 289
partial signing. *See* delayed signing
partially completed operations, backing out of, 420–21
PE file format, 4
PE file header, 5, 38
PE (portable executable) files, 38
creating, 107
embedding standard Win32 Version resources into, 56
file format for, 11
parts of, 38
producing with AL.exe, 54
producing without manifest metadata tables, 49
PE header
in a managed module, 5
in a PE file, 38
PE modules. *See* modules
PerfMon.exe
examining the impact of exception handling on performance, 427, 428
showing the .NET CLR memory counters, 506
performance counters, installed by the .NET Framework, 506
performance ramifications
of boxing and unboxing/copying operations, 144
of exception handling, 426–29
of garbage collection, 458
of JIT compiling, 16
of a managed environment, 17–18
of a method call, 16
permission attribute classes, 351
PEVerify utility (PEVerify.exe), 20
picture format strings, 277, 289–98
pinned objects, 503
Platform Invoke. *See* P/Invoke
platform sensitivity of the NewLine property, 254
plug-in components, designing applications to support, 331–33
plug-in developers, 332
Point objects, 163
Point structures, 141
pointers
maintained by the managed heap, 453
obtaining to raw value types, 142, 143
policy information, creating for new assemblies, 106
pool threads, 434
portable executable (PE) file, 4
positional parameters, 348

PowerPoint, Unhandled Exception dialog box, 433
precise semantics, 188
predefined attributes, 173
  applicable to fields, 175
  applicable to methods, 175–76
  applicable to types, 174
_prev field, 377
  of MulticastDelegate, 372, 373
  setting to null, 378, 381
  setting to the old head of the chain, 381
primary parameterful property, 224
primitive type names, compared to FCL type names, 129
primitive types, 127–28
  C# compiler's knowledge of, 129–31
  declaring as enumerated types, 301
  defining constants for, 177
  implementations of == and != operators, 160
  mapping directly to FCL types in C#, 128
  overflows resulting from arithmetic operations on, 131
  Strings as, 253
  writing as literals, 130–31
printf function, C code calling, 32–33
private, marking methods or fields as, 25
Private accessibility
  in CLR, 173
  in Visual Basic .NET, 173
private accessibility in C#, 173
private deployment, 71, 95–96
private fields of MulticastDelegate, 372
private key, access to, 90
private parameterless constructor, defining, 174
PrivateBinPath configuration setting for an AppDomain, 511
PrivateBuild field, 59
privately deployed assemblies, 63–64, 73
privatePath attribute, 65
probing element in the XML configuration file, 102
"The process cannot access the file "Temp.dat"..." message, 478
ProcessExit event, 515
ProcessItems method, 368
  calling App's private method, 369
  calls to, 368–69
  intermediate language (IL) for the Set type's, 374, 375
/product switch of AL.exe, 59
ProductName field, 59
PRODUCTVERSION, 58
ProductVersion field, 59
/productversion switch of AL.exe, 58, 59
programmatic control of the garbage collection, 499–501
programmatic interface, 402–3
programming, mixed language, 13
programming errors, forgetting to free memory and using freed memory, 452, 458
programming language constructs
  mapping to CLR fields and methods, 30–31
  translating into fields and methods, 28–29
programming languages
  capabilities of, 3–4
  handling of overflows, 131
  selecting in the .NET Framework, 3–4
  support of operator overloading, 191
  switching easily, 13
programming paradigm of the .NET Framework, 22
programming patterns, compiler recognition of common, 129–31
Projects tab on the Add Reference dialog box, 53
properties, 25, 170, 215
  of Encoding-derived classes, 293–96
  of the MemberInfo-derived types, 538–39
  as smart fields, 218
  specifying for custom attributes, 353, 354
  support in C#, 219
  in the System.Exception type, 406–7
  of Type objects, 537, 538
  using for operations that execute quickly, 219
  without backing fields, 218
properties dialog box for an assembly, 104–5
property definition entries, 219
property page dialog box in Visual Studio .NET, 450
PropertyDef table, 39
PropertyInfo class, 219, 225
PropertyInfo type, 548, 549
protected, virtual method, 242
protected accessibility in C#, 173
Protected accessibility in Visual Basic .NET, 173
Protected Friend accessibility in Visual Basic .NET, 173
protected instance field, 241
protected internal
  accessibility in C#, 173
  marking methods or fields as, 26
protected methods of System.Object, 116
provider, IFormatProvider parameter, 285
pseudo-custom attributes, 362–63
public
  marking methods or fields as, 26
  marking types as, 25
public, parameterless constructor, 181–82
public, parameterless Dispose method, 475

Public accessibility
  in CLR, 173
  in Visual Basic .NET, 173
public accessibility in C#, 173
public delegate, 368
Public flag, 524, 545
public instance methods, 115–16
public key numbers, displaying with SN.exe, 75–76
public key tokens, 40, 76, 78
public keys
  embedding in the manifest, 76
  extracting from a file, 91
  hashing, 78
  size of, 76
public/private key cryptographic technologies, 74
public/private key pair
  creating, 75–76
  identifying an assembly's publisher, 73
  signing the publisher policy assembly with, 107
  uniquely marking assemblies, 74
public types
  allowed by C#, 125
  identifying in assemblies, 48
publicly exported type, 52
publisher policies, version numbers for, 107
publisher policy assemblies, 108
publisher policy assembly file, 106–7
publisher policy control, 106–8
publisherPolicy element, 103, 108
publishers, authenticating, 90

Q

qsort function of C runtime, 365
qualifyAssembly element, 526

R

R format for round-trip in numeric types, 277
-R switch of the SN.exe utility, 92
rank of an array, 311
Rank property of System.Array, 312
Rational parameter, taken by the two op_Explicit methods, 200
-Rc switch of the SN.exe utility, 92
reachable objects
  on the freachable queue, 470
  graphing all, 457
Read method, 401, 402
read-only fields, 25, 179, 180
read-only properties, 218
read-write fields, 179, 180
readonly C# attribute for fields, 175
ReadOnly VisualBasic .NET attribute for fields, 175
Rebase.exe tool, 93
recovery code, placing in a central location, 393–94
redimensioning, arrays, 323–24
redistribution package for installing .NET Framework, 9
ref keyword, 201
  with reference types, 203–6
  with value types, 202, 203
/reference command-line switch
  in the C# compiler, 77
  of CSC.exe, 87
/reference compiler switch with the C# compiler, 52
reference metadata tables, 40, 519
reference type elements, unboxing to value type elements, 316
reference type fields, 155–56
reference type variables, 139
reference types, 134
  compared to value types, 136–40
  converting value types to, 141–42
  creating arrays of, 309, 310
  defining with type constructors, 187
  implementing the Equals method for, 160
  implementing zero or more interfaces, 330
  needs for, 519
  out and ref with, 203–6
  performance considerations when working with, 134
  shallow copying and, 166
  unboxing, 143
referenced assemblies, specifying to the complier, 77
referenced types, resolving, 99
ReferenceEquals method
  of Object, 262, 263
  with System.Object, 161–62
references, obtaining to a System.Type object, 534–38
ReflectedType property, common to
    MemberInfo-derived types, 538, 539
reflection, 354, 507, 508, 518–20
  avoiding for accessing a member, 555
  code access security used by, 547
  as handled by the FCL, 425
  performance issues with, 534, 555
  sample applications using, 520–56
Reflection namespace, 23, 518, 519
reflection-related exception types, 410
reflection types
  current holes in, 519
  hierarchy of, 540
ReflectionPermission permission, 547
ReflectOnAssembly method, 521, 523

Reflector sample code, 520–22, 523
reflexive equality property, 154
RegAsm.exe tool, 32
Regional And Language Options dialog box, 261
"registering" an assembly in the GAC, 83
release versions, unhandled exception policy for, 432
RemotingException exception, 542
remove accessor method, 236–38
Remove method
of the Delegate class, 376
of the Delegate type, 381–83
of StringBuilder, 274
of System.Delegate, 233
RemoveEventHandler method, 549
RemoveHandler method, 242
renaming files, 39
Replace method of StringBuilder, 274
replaceable parameters
formatting, 284
identified by the Format method of String, 280
ReRegisterForFinalize method, 490, 491, 493
resource editor in Visual Studio .NET, 56–57
resource files, 46–47, 55–56
resource leaks
avoiding with finally blocks, 416
not possible in the garbage collection environment, 458
special tools designed to locate, 452
resource management, simplified by garbage collection, 452
resource-only assemblies, 54. *See also* satellite assemblies
/resource switch of the C# compiler, 56
ResourceResolve event, 515
resources
identifying in assemblies, 48
steps required to access, 451–52
Resources namespace, 23
response files, 88
resurrected objects, 471
resurrection, 489–91
avoiding if possible, 490
designing an object pool using, 491–93
not tracked by a short weak reference, 488
tracked by a long weak reference table, 489
return type, methods differing by, 200
return values in a delegate chain, 379, 380
reverse engineering of IL code, 12
Reverse method, 314
roots for applications, 455–58
RSA digital signature, 77
Run method, 439
run-time errors, compile-time errors preferred to, 339
RunClassConstructor method, 189
runtime
casting operations checked at, 118–19
resolving type references, 98–101
RuntimeHelpers type, 189
Runtime.InteropServices namespace, 23
Runtime.Remoting namespace, 23
Runtime.Serialization namespace, 23
RuntimeType type, 534–35

S

S format for sortable date in the DateTime type, 276–77
safe code, 20
safe conversions, 130
safe point, 501
satellite assemblies, 54, 62, 66
sbyte primitive type, 128
scalable parallel collections, 503
SCM (Service Control Manager), 22
Script check box in the Attach to Process dialog box, 449
scripting code, debugging, 448, 449
sealed C# attribute
for methods, 176
for types, 174
Sealed CLR attribute for types, 174
search symbols, 524
secured files, exception handling of, 406
security model in the .NET Framework, 37
Security namespace, 23
security of Windows applications, 36
security policy, applying to an AppDomain, 511
SecurityException exceptions, 406, 547
SEH (structured exception handling) mechanism, 396
self-describing assemblies, 8–9, 48, 49
self-tuning collector, garbage collector as, 498
semantics, 188
Serializable attribute, 362, 413, 514
SerializableAttribute attribute, 346
server
client code catching an exception thrown by, 433
marshalling back the stack trace to the client, 444
server-side application, graceful recovery for, 432
server version of the CLR COM server, 509
Service Control Manager (SCM), 22
service pack versions, 108
set_
prepending by the compiler, 222
prepending to a property name, 219

set accessor method
emitting a method representing the parameterful property, 222
for an indexer, 221
for a property, 218
Set class, defining a public delegate, 368
set method, specifying to define a property, 218
SetAppDomainPolicy method, 511
set_Bit method, 223
Set.Feedback delegate object, 368
SetField flag, 547
setjmp, /clr switch and, 32
SetProperty flag, 547
SetValue method
of PropertyInfo, 548, 549
of System.Array, 313
SHA-1 hash algorithm, 77
Shadows VisualBasic .NET attribute for methods, 176
shallow copy, 165
performed by the Array.Copy method, 316
of the source array, 314
shared assemblies, 512
Shared VisualBasic .NET attribute, 175
shell extension, displaying assemblies installed into the GAC, 83
shim DLL, 509
short-lived objects, 255
short primitive type, 128
short weak reference table, 487–88
short weak references, 487, 488
shortcut links, 63
ShutdownThePool method of Expensive, 493
side-by-side execution, 96–98
sign characters, leading/trailing in a string, 286
signature of a method, 25
silent overflow, 131
single-cast delegates, 375
single-dimension arrays, 311
single-module assembly, 512
smart fields, properties as, 218
snapshots of the set of assemblies loaded by an application, 109
SN.exe, 75
-p switch, 91
R switch, 92
-Rc switch, 92
-tp switch, 75, 76
-Vr switch, 91
-Vu switch, 92
-Vx switch, 92
SOAP fault XML, 441
SoapException object, 441
Sort method, 313, 425
SortedList type, 325
sorting strings, 258, 259–61
source code
compiling, 3–7, 77
compiling files, 4–5
file pathnames and line numbers, 443
Source property of the System.Exception type, 407
special characters in C#, 254
special exceptions, 411
"special" method in C#, 191
special method names for C# operators, 191–92
SpecialBuild field, 59
specialname flag, 193–94, 196
specialname method, 191
specific exceptions, 398
spoofing types, 119
SQL Server, next version of, 517
square brackets [ ], placing custom attributes within in C#, 346–47
stack-based code in IL, 19
stack-based value type fields, 186
stack frames, displaying the number traversed, 429
stack space, assumption that enough is available, 404
stack trace
in an exception, 395
remoting, 444–45
returning the complete, 443
StackOverflowException exception, 404, 405, 410, 411
StackTrace method, 443
StackTrace object, 443
StackTrace property, 407, 441–43, 444, 445
StackTrace type, 443
StartsWith method, 257
state information, 215–16
static C# attribute, 175
Static CLR attribute, 175
static constructors. *See* type constructors
static events, 170
static fields
initializing, 170
syntax for initializing, 188–89
static Finalize methods, not supported by the CLR, 190
Static flag, 524, 545
static methods, 209
calling back with delegates, 368–70
offered by the Char type, 250
of the System.Convert type, 130
static parameterful properties, 222
static read-only fields, 179

static read-only objects, 241
StaticCallbacks method, 368
stored procedures, writing for Yukon, 518
Stream class, IS-A relationship with other classes derived from, 331
stream classes, using inheritance for, 331
stream I/O, 23
streaming data, design of classes related to, 331
StreamReader type, 290
StreamWriter type, 290
string characters, methods for examining, 266–67
String class, 249
  compared to StringBuilder, 274–75
  overloads of the static Format method offered by, 281
  sealed, 256
string comparisons, performance effect of, 262
string description in an exception, 395
string formatting and culture information, 276
string interning
  mechanism in the CLR, 262–66
  on a per-process basis, 264
String objects
  constructing in C#, 253–54
  creating in C++, 146
  swapping references to two, 205–6
string of comma-delimited symbols, converting into a numeric value, 307
string pooling, 266
string primitive type, 128, 129
String references, 256
string representations, 116, 275–84
String type, 253–70
  constructors provided by, 254
  indexer's name changed in, 224
  methods for comparing strings, 256–59
StringBuilder object, 271–72
StringBuilder type, 255, 270–71
  chaining operations together, 274
  internal fields maintained by, 271–72
  members of, 272–74
StringInfo class, 269
StringInfo type, 267
strings
  comparing, 256–62
  concatenating to form a single string in C+, 254–55
  constructing, 253–55
  declaring in C#, 255
  dynamically constructing, 270–75
  examining the characters of, 266–69
  formatting multiple objects into a single, 280–81
  merging multiple occurrences into a single instance, 266
  methods for copying, 270
  methods for manipulating, 270
  from non-Unicode systems, 289
  parsing to obtain objects, 285–89
  reasons for comparing, 258
  returning the number of characters in, 266
  sorting, 258, 259–61
  treating as tokens, 270
Strong Name Utility, 75
strong reference, 485
strongly named assemblies, 72–73, 522
  attributes of, 74
  creating, 75–77
  deploying an arbitrary directory, 95–96
  installing into the GAC, 80
  privately deploying, 95–96
  tamper resistance of, 89–90
strongly typed Equals method, 159, 160
struct, declaring value types as, 137
StructLayoutAttribute attribute, 140
StructLayoutAttribute type, 348
structure, indicating a value type in the documentation, 135
structured exception handling (SEH), 396
stub functions, 10, 11
style NumberStyles parameter, 285
subdirectories, scanning for assembly files, 66
subhierarchies, defining for exception types, 412
SubString method, 270
SuppressChangeType flag, 549
SuppressFinalize method of GC, 477, 479
surrogate character pairs in UTF-8, 290
surrogate characters in Unicode, 267
switch/case statement, working efficiently on strings, 264–66
symbolic names, defining a set with value pairs, 299
symbols
  defined by enumerated types, 301
  looking up the equivalent value of, 303
  for NumberStyle's bit combinations, 287–88
  showing for managed code, 449
symmetric equality property, 154
SyncBlockIndex, 117, 150, 453
synchronization-free allocations, 503
synchronized, marking methods as, 233
SyncRoot property of System.Array, 313
syntax checkers, compilers as, 4
System Monitor ActiveX control, 427
System Monitor control, 506
System namespace
  CLR exceptions defined in, 446
  types contained in, 22
System.AppDomainSetup class, 511

System.Array type, 312–14
System.Attribute type
deriving custom attribute types from, 348
methods defined by, 355, 356
System.AttributeTargets enumerated type, 351
System.Boolean type, 128
System.Byte type, 128
System.Char structure, 249
System.Char type, 128, 249
System.Collections namespace, 22
System.Collections.Hashtable type, 221
System.ComponentModel.EventHandlerList type, 246
System.Console type, 32, 37
System.Console WriteLine method, 32–33
System.Convert type, 251, 252
System.Decimal type, 128, 134, 200
System.Delegate type, 375, 387
System.Diagnostics namespace, 23
System.Diagnostics.DebuggableAttribute custom attribute, 444
System.Diagnostics.Debugger type, 437
System.dll assembly, 512, 513
System.Double type, 128
System.Drawing namespace, 23
System.Drawing Properties dialog box, 104–5
System.EnterpriseServices namespace, 23
System.Enum base type, 301–4
System EventArgs type, 241–42
System.EventHandler delegate type, 242
SystemException base type, 410, 411
System.Exception type, 406–7
in C#, 398
displaying all classes derived from, 529–32
SystemException type
exceptions derived from, 410
never throwing for private methods, 411
System.FlagsAttribute custom attribute, 305–6
System.Globalization namespace, 23
System.Globalization.CultureInfo class, 250
System.Globalization.CultureInfo type
implementing the IFormatProvider interface, 278
returning an instance of, 277
System.Globalization.StringInfo type, 267
System.Globalization.TextElementEnumerator object, 267
System.Globalization.UnicodeCategory enumerated type, 249
System.ICloneable interface, 327
System.IDisposable interface, 474–75
System.IFormatProvider interface, 276
System.IFormattable interface, 276
System.Int16 type, 128
System.Int32 type, 128
System.Int64 type, 128, 129
System.IO namespace, 23
System.IO.BinaryReader type, 290
System.IO.BinaryWriter type, 290, 484–85
System.IO.Stream class, methods provided by, 331
System.IO.StreamReader type, 290
System.IO.StreamWriter type, 290
System.Management namespace, 23
System.MarshalByRefObject, 514
System.MulticastDelegate class, 375
System.MulticastDelegate type, 372
System.Net namespace, 23
System.Object class, 115–16
System.Object type, 128
all types derived from, 115
Equals method offered by, 153–54
inheritance from, 26
System.ParamArrayAttribute custom attribute, 207, 208
System.Reflection namespace, 23, 356, 518, 519
System.Reflection.Assembly type, 526
System.Reflection.AssemblyAlgIDAttribute custom attribute, 77
System.Reflection.AssemblyKeyFileAttribute attribute, 76
System.Reflection.AssemblyKeyNameAttribute attribute, 92
System.Reflection.AssemblyName class, 75
System.Reflection.FieldInfo type, 363
System.Reflection.MemberInfo type, 538
System.Reflection.PropertyInfo class, 219, 225
System.Resources namespace, 23
System.Runtime.CompilerServices.IndexerName custom attribute, 223
System.Runtime.CompilerServices.RuntimeHelpers type, 189
System.Runtime.InteropServices namespace, 23
System.Runtime.Remoting namespace, 23
System.Runtime.Serialization namespace, 23
System.RuntimeType, 534
Systems Properties dialog box for the GAC, 83–84
System.SByte type, 128
System.Security namespace, 23
System.Security.Permissions.ReflectionPermission permission, 547
System.Security.VerifierException exception, 19, 20
System.Serializable custom attribute, 514
System.SerializableAttribute type, 362
System.ServiceProcess namespace, 24
System.Single type, 128
System.String class, 249
System.String type, 128, 129, 224, 253–70
System.Text namespace, 23

System.Text.StringBuilder type, 255, 270–71
System.Threading namespace, 23
System.Threading.Thread's Abort method, 405
System.Type, read-only properties offered by, 363
System.Type object
  checking for the existence of an attribute via, 355
  obtaining a reference to, 534–38
System.Type type, 534
System.UInt16 type, 128
System.UInt32 type, 128
System.UInt64 type, 128
System.ValueType type
  structures derived from, 135
  value types derived from, 157
System.WeakReference type, 487
System.Web.Services namespace, 24
System.Web.UI namespace, 24
System.Web.UI.Control type, 239, 246
System.Web.UI.TemplateControl class, 440
System.Windows.Forms namespace, 24
System.Windows.Forms.Application class, 439
System.Windows.Forms.Control type, 238–39, 246
System.Windows.Forms.NativeWindow type, 439
System.Xml namespace, 23
SZ arrays, 311

T

T format for time date in the DateTime type, 276–77
T-SQL procedures, debugging, 449
tamper resistance of strongly named assemblies, 89–90
_target field
  comparing for two delegates, 377
  of MulticastDelegate, 372
target parameter
  for a delegate, 373
  of InvokeMember, 546
  of WeakReference, 487
Target property
  defined by the MulticastDelegate class, 373
  querying the WeakReference object's, 487
  of WeakReference, 488
/target switch of AL.exe, 58
/target:exe command-line switch
  of AL.exe, 55
  with the C# compiler, 49
TargetInvocationException exception, 544
/target:library switch with the C# compiler, 49
TargetMethods field, 553
targets, inheritable for attributes, 352
TargetSite property of the System.Exception type, 407
TargetType field, 553
/target:winexe command-line switch
  of AL.exe, 55
  with the C# compiler, 49
technical committee (TC39) of ECMA, 14
temporary file, creating, 478–81
/t:exe switch with the C# compiler, 38
Text namespace, 23
.text section of the PE file, 10, 11
TextElementEnumerator object
  acquiring, 267
  returning, 269
this pointer, received by instance methods, 209
thread pool threads, 434
thread-safe manipulation of a StringBuilder object, 272
thread synchronization situations, avoiding for Finalize methods, 470
ThreadAbortException exception, 405, 514
ThreadException event of Application, 440
ThreadExceptionEventHandler delegate, 440
Threading namespace, 23
threads
  AppDomains and, 514
  executing unmanaged code, 503
  falling out the bottom of catch blocks, 400
  falling out the bottom of finally blocks, 401
  hijacking, 501
  kinds of, 434
  multiple accessing the managed heap, 501
  suspending during garbage collection, 501
thread's stack, allocating value types on, 135
threshold size, selecting, 494, 497
throw keyword, rethrowing exceptions, 421
Throw To Catch Depth/Sec counter, 429
time-zone specifiers, 288
/title switch of AL.exe, 59
TlbExp.exe tool, 32
TlbImp.exe tool, 31
/t:library switch with the C# compiler, 50
/t:module switch with the C# compiler, 49
ToBase64CharArray method, 298
ToBase64String method, 298
ToChar method, 251
ToCharArray method, 267
tokens, treating strings as, 270
ToLower method, 250
ToObject methods, 304
top-level namespace name, 125
ToSingle method, 343
ToString method
  with Char type, 250
  for copying strings, 270
  of Decimal, 278–79
  of Enum, 302, 303

ToString method, *continued*
formatting output, 306
of IFormattable, 276, 278
obtaining a string representation for any object, 275
overloads offered by many types, 280
problems with the parameterless, 276
of StringBuilder, 271, 273
of System.Object, 116
ToUInt16 method, 251
ToUpper method, 250
-tp switch of SN.exe, 75, 76
trackResurrection parameter of WeakReference, 487
/trademark switch of AL.exe, 59
Transfer method, 403
transitions, stepping over during debugging, 450
transitive equality property, 154
try block, 397–98, 401
/t:winexe switch with the C# compiler, 38
two-dimension array, 318
type binding, flowchart of, 100
type checking, making possible compile-time, 339
type constructors, 170, 187–90
type conversions. *See* casting
type definition for an event, 228–33
type definition metadata table, 518
type fields, allocating dynamic memory for, 178
type initializers. *See* type constructors
type libraries, 6
Type objects
returning, 116
ways of obtaining, 535
type references, resolving during runtime, 98–101
type safe, delegates as, 369–70
type-safe CompareTo method, 339
type-safe Equals method, 160
type safety
checking in the EventHandlerSet, 245
of CLR, 117
compromising, 206
improving with explicit interface member implementations, 341
type-safety quiz, 121
Type type, 534, 540
typedef, compared to a delegate, 368
TypeDef table, 39, 48
TypeDelegator class, 534
TypeInformation bit, 547
TypeInitializationException exception, 188
typeless instructions in IL, 19
TypeLoadException exception, 535
typeof operator, 353, 535–36, 537
TypeRef table, 40
TypeResolve event, 515
types, 24, 35
accessibility modifiers for, 173
accessing, 99, 100, 514
across assemblies, 332
adding conversion operator methods to, 198
adding data members to, 177–80
all derived from System.Object, 115
avoiding defining with the base type in another assembly, 332–33
avoiding the Finalize method when designing, 463–64
behavior of, 26
building into a module, 37–45
calling methods, 543–47
casting between, 117–21
compatibilities offered by event members, 227
converting, 197
creating instances of, 541–43
creating new instances with identical fields, 116
declaring as value types, 138
declaring in unmanaged C/C++, 136
defined by types, 170
defining, 37
defining conversion constructors and methods for, 197
defining instance constructors, 182
defining to hold additional event information, 230
defining to implement the dispose pattern, 481
defining to use events provided by another type, 234–36
defining to wrap an unmanaged resource, 460
designing to define lots of events, 238–43
designing to expose events, 228–33
designing to listen for events, 234–36
developing for use by other developers, 416
disambiguating, 125
displaying namespace and assembly information for, 126
dynamically locating and constructing, 555–56
ease of deriving from a base type, 330
for events, 231
Finalize method called even if the instance constructor throws an exception, 465–66
getting resurrected out of your control, 490
implementing from different programming languages, 47
implementing interfaces, 328
implementing or inheriting methods, 329
implementing the dispose pattern, 477–82
implicit assumptions in the definition of, 403–6
indexers offered by, 221
inheriting interfaces, 327
item property offered by, 222
kinds of, 127, 134

loading specific at run time, 519–20
members contained in, 24–25
members defined by, 169–72
members of, 28–29
offering explicit cleanup, 472
partitioning among separate files, 46
predefined attributes applicable to, 174
referencing with fully qualified names, 123
reflecting over members of, 523–25, 538–52
scoped by the referencing assembly, 64
setting the initial state of, 187
spoofing, 119
state information defined by, 215
unloading, 190
versioning issues when adding or modifying members of, 210–11
visibility rules, 25
walking reflection's OM, 540–41

U

U format, universal time in the DateTime type, 277
/U switch with GACUtil.exe, 80, 81, 82
uint primitive type, 128
ulong primitive type, 128
unbox and copy operations in C#, 143
unboxed value types, 138, 150
unboxing, 142–44, 145–52
unchecked operator in C#, 132, 133
unchecked statement in C#, 132–33
underlying type, declaring for an enumerated type, 301
Unhandled Exception dialog box in Microsoft PowerPoint, 433
unhandled exceptions, 432–37
ASP.NET Web Forms and, 440–41
ASP.NET XML Web services and, 441
directing the debugger how to respond, 447–48
implementing a policy for, 434–37
notification of, 394
Windows Forms and, 439–40
UnhandledException event, 438, 515
UnhandledExceptionEventArgs object, 437
UnhandledExceptionEventHandler delegate, 436
Unicode
encoding, 290
surrogate characters in, 267
Unicode Transformation Format. *See* UTF
UnicodeCategory enumerated type, 249
UnicodeEncoding class, 292
unmanaged C/C++. *See also* C; C++
allocating types in, 136
callback functions not type-safe, 365
unmanaged C++
compilers, 426–27
garbage collection not available in, 459
unmanaged code
compared to managed, 18
debugging, 449
interoperability with, 31–34
thread executing, 503
using managed type (server), 31–32
unmanaged COM code, interoperating with, 407
unmanaged COM components, creating managed assemblies for, 31
unmanaged functions, calling with managed code, 31
unmanaged modules, 7
unmanaged resources
freeing, 464
types wrapping, 460
unmanaged threads, 434
unnested types, marking with Public or Assembly accessibility, 174
unreachable objects, 456
unrecoverable exceptions, backing out of partially completed operations, 420–21
Unregister method, 236
unregistering
delegates from the linked list, 232, 233
interest in all events, 235
unsafe array manipulation technique, 319–21, 323
"unsafe" conversions, explicit casts required for in C#, 130
unsafe keyword, 20
unsigned integer types, not supported in Visual Basic .NET, 193
Unwrap method, 541–42
/UR switch of GACUtil.exe, 81, 82
URL
with documentation on an exception, 407
identified by a configuration file's codeBase element, 96
Use Parent Setting option in the Visual Studio .NET Exception dialog box, 447–48
user-defined attributes. *See* custom attributes
users
allowing to debug applications, 433
notifying of an unhandled exception in Windows Forms, 439
ushort primitive type, 128
using directive
in the C# complier, 122
creating an alias for a single type or namespace, 124
using statement in C#, 417, 483–84

UTF (Unicode Transformation Format), 290
UTF-7 encoding, 290
UTF-8 encoding, 290, 291
UTF-16 encoding, 290
UTF7Encoding class, 292
UTF8Decoder class, 297
UTF8Encoding class, 292

V

_value modifier in C++ with Managed Extensions, 137
value parameter in C#, 221
value type constructors, 184–87
value type elements, boxing to reference type elements, 316
value type fields
  comparing, 156
  stack-based, 186
value type instances
  allocating, 135
  boxing, 138
value type objects, 138
value type variables, 139
value types, 134, 135
  boxing, 330
  boxing an instance of, 141–42
  boxing and unboxing, 141–52
  changing fields in boxed, 333–36
  cloning, 166
  compared to reference types, 136–40
  converting into heap-managed objects, 141–42
  converting to reference types, 141–42
  creating an instance of without calling a constructor, 543
  creating arrays of, 309, 310
  defining type-safe version of Equals, 160
  defining with type constructors, 187
  derived from System.ValueType, 139
  enumerated types and, 300
  Finalize method and, 139–40
  implementing one or more interfaces, 135
  implementing zero or more interfaces, 330
  manually boxing, 149–50
  nuances of, 336
  out and ref with, 201–2, 203
  parameterless constructors not allowed in C#, 185–86
  reducing boxing operations for, 341
  referencing instances of, 141
  testing for equality, 159–60
  unboxed, 150
ValueType type
  implementing the Equals method, 157–58
  structures derived from, 135
variables, life extended by the JIT compiler, 456
vectors, 311
verbatim strings, 255
verification
  during IL compilation, 19
  turning back on for assemblies, 92
  turning off, 20
VerifierException exception, 19, 20
version numbers
  associated with assemblies, 60–61
  format of, 59–61
  for publisher policies, 107
version resource information in assemblies, 56–61
/version switch of AL.exe, 58, 59, 107
Version tab
  of a Properties dialog box, 56
  of the System Properties dialog box, 83, 84
versioning issues, 476
  avoiding for assemblies, 332
  base types compared to interfaces, 330
  causing source compatibility problems, 210–14
  for constants, 178
  dealing with problems in the .NET Framework, 72
  fixing with a static read-only field, 179
Vietnamese currency, 278
View.MetaInfo.Show! menu item in ILDasm, 41
View.Statistics menu item in ILDasm, 44–45
violations, 395. *See also* exceptions
virtual address spaces, 20
virtual C# attribute for methods, 175
Virtual CLR attribute for methods, 175
virtual keyword, omitting from an interface method in C#, 327
virtual methods
  calling, 209
  defaulting to public and sealed in C#, 475
  defining by interfaces, 330
  versioning, 210–14
visibility rules for types, 25
Visual Basic .NET, 13
  applying custom attributes, 347
  building an application using a C# type, 195
  handling overflows, 131
  Imports statement, 122
  Invoke method must be explicitly called, 374
  lack of support for the unsigned integer types, 193
  operator overloading not supported, 193
  terms for accessibility modifiers, 173
Visual Basic .NET type, offering an op_Addition method, 193, 194
Visual C++ compiler, /clr command-line switch, 32
Visual Studio .NET
  adding assemblies to projects, 52

debugger support for exceptions, 445
debugging applications, 448–50
multifile assemblies not supported, 49
volatile fields, 179
-Vr command-line switch of the SN.exe utility, 91
-Vu command-line switch of the SN.exe utility, 92
-Vx command-line switch of the SN.exe utility, 92

W

W format for universal time in the DateTime type, 276–77
WaitForPendingFinalizers method of GC, 499–500
warning. *See also* error
CS0108: The keyword new is required on..., 210
CS0114: ... hides inherited member..., 212
watchdog process for respawning a server, 405
weak references, 485–89
weakly named assemblies, 72–73, 522
codeBase elements with, 102–3
locating, 511
weakly referenced objects, 486
WeakReference objects, 487
"Web applications", security implications of, 36–37
Web Forms
applications, 21
types for, 24
Web sites, loading assemblies from, 517
Web.config files, 67
WebName property of Encoding-derived classes, 293
white-space characters, leading/trailing in a string, 286, 288
Win32 Exceptions selection, 445
Win32 ExitProcess function, 516
Win32 functions, returning FALSE, 394
Win32 resources, embedding into assemblies, 56
Win32 version resource, 60
/win32icon switch with AL.exe or CSC.exe, 56
/win32res switch with AL.exe or CSC.exe, 56
Windows
application types supported by, 38
callback functions in, 365
console applications, 22
GUI applications, 24
installing East Asian Language files, 261
reputation for instability, 36–37, 71
shutting down, 516
virtual address spaces, 20
Windows Forms
applications, 21, 516
CLS-compliant exceptions only, 440
types for, 24
unhandled exceptions and, 439–40
Windows Installer (MSI), installing assemblies into the GAC, 82
Windows Installer service, 63
Windows Management Instrumentation (WMI), 23
Windows .NET Server Family, 11
Windows PE file header. *See* PE file header
Windows process, shutting down, 516
Windows Service Control Manager. *See* SCM
Windows services
applications, 22
types for, 24
Windows XP, invoking managed assemblies, 11
WindowsCodePage property of Encoding-derived classes, 293
WndProc method, 439
worker process, creating in ASP.NET, 516
workstation version of the CLR COM server, 509
Write method of Console, 281
write-only fields, 25
WriteLine method, 37–38, 281

X

X format for hexadecimal in enumerated and numeric types, 277
x86 stub functions, 10, 11
XML configuration file
created by a publisher, 65
elements of, 101–4
identifying rollback changes to, 111–12
modifying to instruct the CLR to load a new assembly, 103
for a new assembly, 106
telling the shim to load a particular CLR version, 509
XML namespace, 23
XML Web service applications
configuration files for, 67
garbage collector works extremely well with, 498
XML Web services, 21, 24

Y

Y format for year/month date format in the DateTime type, 277
Yukon, 517–18

Z

zero-based arrays, 310
zero element arrays, 317

术 语 表

.NET Compact Framework	.NET 微缩框架
accessibility modifiers	访问限定修饰符
accessor methods	访问器方法
AppDomain	应用程序域
argument	参数
array	数组
assembly	程序集
attribute	特性(指.NET 中一种可声明的元数据信息) 属性(指 XML 中一种描述元素的名/值对信息)
base class	基类
base type	基类型
bind	绑定
bit flag	位标记
box	装箱
boxed	已装箱
build	生成
catch filter	捕获筛选器
callback function	回调函数
cast	转型
class	类
cleanup	(资源)清理
clone	克隆
code access security(CAS)	代码访问安全
command-line switch	命令行开关
Common Language Infrastructure(CLI)	通用语言基础构造

Common Language Runtime (CLR)	通用语言运行时
Common Language Specification (CLS)	通用语言规范
Common Type System (CTS)	通用类型系统
compile	编译
component	组件
concurrent	并发
console	控制台
constant	常数
constructor	构造器
conversion operator	转换操作符
copy	拷贝
culture	语言文化
culture-neutral	语言文化中性
custom attribute	定制特性
debug	调试
deep copy	深拷贝
delegate	委托
design pattern	设计模式
destructor	析构器
domain-neutral assembly	以中立域方式加载的程序集
dynamic link library	动态链接库
encapsulation	封装
enumerated type	枚举类型
enumeration type	枚举类型
event	事件
evidence	证据
exception	异常
explicit	显式
explicit interface member implementation	显式接口成员实现
expression	表达式
field	字段
finalization	终止化
finalizer	终止化器

Framework Class Library(FCL)	(.NET) 框架类库
garbage collection (GC)	垃圾收集
garbage collector	垃圾收集器
generation	代龄
Global Assembly Cache (GAC)	全局程序集缓存
handle	句柄
hash code	散列码
host	寄宿(作动词)
host	宿主(作名词)
implicit	隐式
imported type	导入类型
indexer	索引器
inheritance	继承
Inline	内联
instance	实例
instruction	指令
interface	接口
intermediate language (IL)	中间语言
interoperability	互操作
jagged array	交错数组
Just-In-Time(JIT) compiler	即时(JIT)编译器
lifetime	生存期
literal	文本常量
managed code	托管代码
managed heap	托管堆
managed module	托管模块
mangle	签名编码
manifest	清单
marshall	封送
metadata	元数据
method	方法
namespace	命名空间
native code	本地代码

nested type 嵌套类型

object 对象

object-oriented 面向对象

operand 操作数

operator 操作符

overload 重载

override 重写

paradigm 范式

parallel 并行

parameter 参数

persist 持久化

plug-in component 插件组件

point 指针

polymorphism 多态

portable executable (PE) file 可移植可执行(PE)文件

primitive type 基元类型

process 进程

property 属性

reference 引用

reference type 引用类型

reflect 反射(作动词)

reflection 反射(作名词)

remote 远程

runtime 运行时

response file 响应文件

satellite assembly 卫星程序集

sealed 密封

serialize 序列化

shallow copy 浅拷贝

side-by-side 并存

single-domain assembly 以独立域方式加载的程序集

stack 堆栈

stack frame 栈帧

stack trace	堆栈踪迹
static	静态
stream	流
string	字符串
strong name	强命名
structure	结构
thread	线程
thread pool	线程池线程
type	类型
unbox	拆箱
unboxed	未装箱
unmanaged code	非托管代码
unsafe code	非安全代码
value type	值类型
weak reference	弱引用
Web Forms	Web 窗体
Web Services	Web 服务
Windows Forms	Windows 窗体
window procedure	窗口过程
worker process	工作者进程
zero-based array	0 基数组