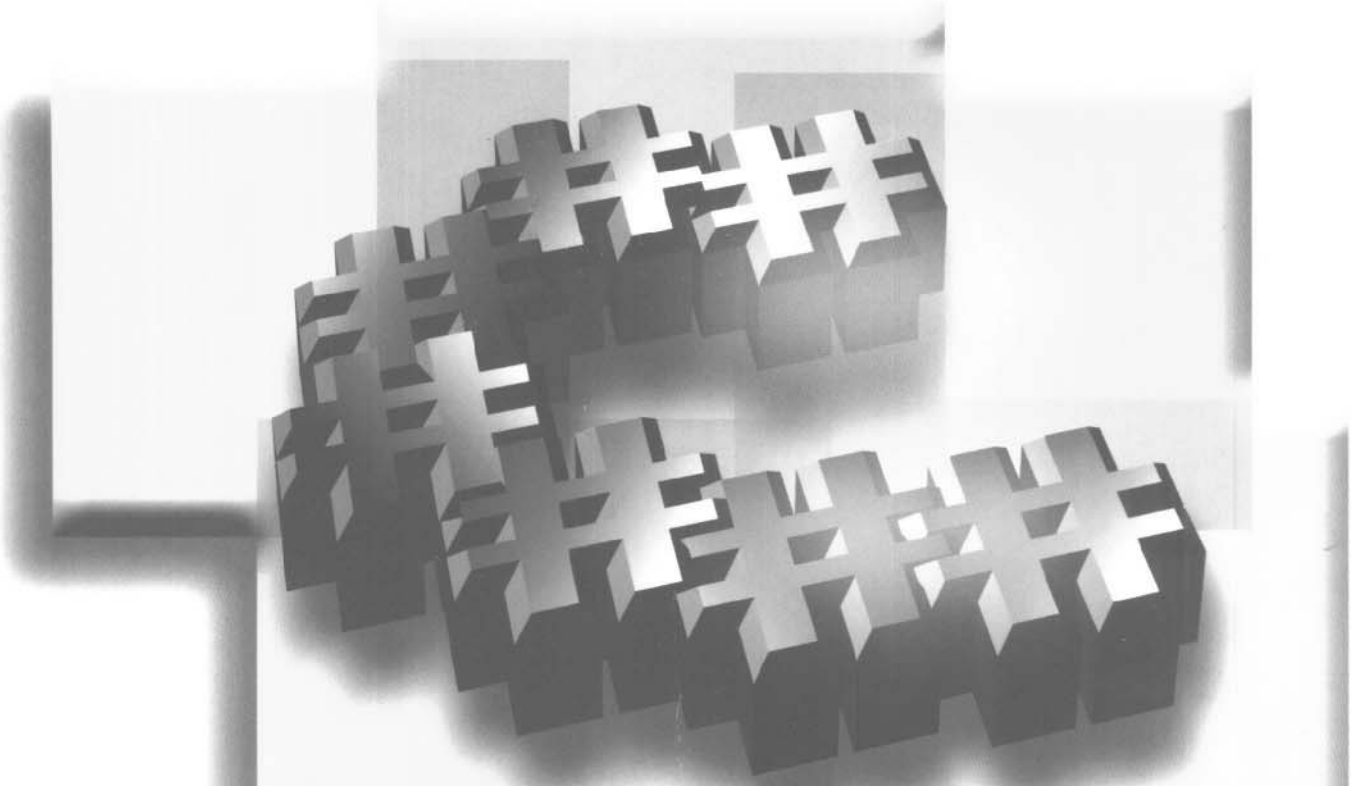
Microsoft .net 微软 Visual Studio.NET 程序开发系列丛书

Microsoft Press

C# 技术内幕



[美] Tom Archer 著
侯晓霞 柴洪辉 译

清华大学出版社
http://www.tup.tsinghua.edu.cn

微软 Visual Studio.NET 程序开发系列丛书

C# 技术内幕

[美] Tom Archer 著
侯晓霞 柴洪辉 译

清华大学出版社

(京)新登字 158 号

内 容 简 介

C#由 C 和 C++语言发展而来，是一种简单、现代、面向对象、类型安全的语言。本书分为基础篇、C#类基础、编写代码和高级 C# 4 个部分，分别描述了关于 C#和.NET 开发环境的一个特定方面的内容，其中包括 .NET 框架介绍、C#编程原则、类型系统、类、特性、接口、多线程编程和使用组合体等。

本书简明易懂，思路清晰，结合恰当的示例代码，特别适于希望学习 C#和.NET 开发环境的读者，以及已经具有 C 家族语言（C、C++或 Java）背景的读者阅读。

Inside C#
Microsoft Press
Copyright © 2001 by Microsoft Corporation
Original English language edition published by Microsoft Press, a Division of Microsoft Corporation
All rights reserved.
No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher. For sale in the People's Republic of China only.

本书中文简体版由 Microsoft Press 授权清华大学出版社出版发行，未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分内容。

北京市版权局著作权合同登记号：图字 01-2001-0120 号

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。

书　　名：C#技术内幕
作　　者：[美] Tom Archer
译　　者：侯晓霞　柴洪辉
责任编辑：孙　岩
出 版 者：清华大学出版社（北京清华大学学研大厦，邮编 100084）
http://www.tup.tsinghua.edu.cn
印 刷 者：北京广益印刷有限公司
发 行 者：新华书店总店北京发行所
开　　本：787×960 1/16　印张：23.25　字数：512 千字
版　　次：2002 年 1 月第 1 版　2002 年 1 月第 1 次印刷
书　　号：ISBN 7-900637-59-1
印　　数：0001～5000
定　　价：49.00 元（含 1 张光盘）

《微软 Visual Studio .NET 程序开发系列丛书》序

.NET 技术简介

.NET 技术是 2000 年微软公司推出的全新概念，对于它很难做出一个明确的定义，它代表了一个集合、一个环境、一个编程的基础结构。其目的是将互联网本身作为构建新一代操作系统的基础，对互联网和操作系统的设计思想进行延伸。具体地说，.NET 技术就是要在不同的网站之间建立起协定，促进网站之间的协同合作，实现信息的自动交流，从而帮助用户最大限度地获取信息、并对他们的数据进行简单、高效的管理。

对于最终用户来说，.NET 技术的实现将使计算机的功能得到大幅度的提升，而计算机的操作却变得更加简单。他们将完全摆脱人为的硬件束缚，利用任何设备、通过任何系统、在任何时间、任何地点访问互联网的多维时空，并对其进行跨应用程序的集成。用户对个人信息的任何修改——无论是通过个人电脑、便携设备还是灵通卡——将即时和自动地通知到所有需要这些信息的地方。

对于应用程序的开发人员来说，.NET 的意义也许更为重大。通过.NET 技术，程序开发人员不必再像过去那样通过集成本地系统服务来构建应用程序，花费大量的精力来考虑如何构建基本的结构，而无法专注于如何实现软件所特有的商业价值。现在，程序开发人员可以访问 Microsoft .NET 所提供的内容丰富的类库，直接使用各种各样来自于 Internet 的模块来构建自己的应用程序，而无需为不同的工作环境重新编写程序，快速、高效地开发应用程序。

对于网络开发人员来说，借助 .NET 技术所提供的、基于 XML 的松散耦合技术，他们终于可以真正地将应用程序逻辑分发到网络上，而不必顾及服务端与客户端是否属于同一种结构、执行机制是否相同、接口是否对应。Microsoft .NET 将把 ASP 和程序集成带到一个新的水平上，将集成程序和其他的集成或非集成程序整合；把这些程序风格化；形成针对这些应用程序编写新程序的能力；并可以在离线状态下运行它们。

Visual Studio.NET 简介

Visual Studio.NET 是 Microsoft 为实现其 .NET 技术而开发的一整套工具组件。它把开发功能强大、性能可靠的企业网络解决方案进行了简化。通过提供终端到终端的网络开发

能力以及可伸缩、可复用的服务器端组件，Visual Studio.NET 将会大大地提高生产率，并促使商务活动更加有效地关注快速多变而又充满竞争的市场需求。

Visual Studio.NET 具有许多令人激动人心的特性，其中的某些特性是对早期 Visual Studio®版本的增强，而另外一些特性则是全新的。这些新特性包括：一个单一的集成化开发环境、Visual Basic.NET 当中面向对象的新特性、以及大量在产品开发生命周期中使用的工具。

Visual Studio.NET 的主要作用在于：

- **提供加速开发过程的高效工具**

  Visual Studio.NET 提供了一个统一的、紧密集成的可视化编程环境，以帮助您简化开发网络应用程序的过程、缩短学习使用方法的时间。通过共享的 HTML，XML 和样式单编辑器，您可以轻轻松松地借助包括 C#在内的任何一种 Visual Studio 语言来开发网络应用程序。

- **提供对各种各样的网络应用程序的快速设计能力**

  借助于 Web Forms，程序员们可以使用他们在开发基于窗体的桌面应用程序时所使用的技巧来创建跨平台、跨浏览器的网络应用程序。Web Forms 被用于网络的服务器端，它能高速地运行，并生成符合 HTML 3.2 规范、能够在各种浏览器上阅读的文档。

- **利用 XML 和 Web Services 来简化分布式计算**

  Web Services 借助标准的 Internet 协议在网络上调用商务逻辑。HTTP 被作为 Web Services 传输的基础协议，该协议使得对功能的请求能够穿越各种团体所使用的防火墙。XML 被用来对上述功能请求的参数进行格式统一，从而使得这些请求能够适用于所有的软件和硬件。这样使得对 Web Services 的访问可以通过任何一种语言、使用任何一种组件模型、在任何一种操作系统上实现。

- **快速构建中间层商务组件**

  Visual Studio 的一个核心目标就是要为基于服务器的应用程序提供应用程序快速部署(Rapid Application Development，RAP)工具。Visual Studio.NET 在创建中层商务逻辑时所表现出的效率丝毫不逊于 Visual Basic®开发窗体程序时的效率。利用 Visual Studio.NET 创建的组件将为您的商务运作提供足够的功能和伸缩性。

- **构建可靠的可伸缩解决方案**

  利用 Visual Studio.NET，您可以非常轻松地创建具有自动伸缩能力的可靠的应用程序和组件。例如，如果使用 Visual C++来编写网络应用程序，新推出的 ALT Server 将会极大地提高性能。

Visual Studio.NET 的其他新性能包括：

- **一种新的语言：C#**
  C#语言是一种优雅、简单、类型安全、面向对象的编程语言。它在向 C 和 C++的用户提供 RAD 的同时，保持了 C/C++所特有的强大功能和控制能力。C#的关键特性包括：
- 具有与 C++十分相似的模型和语法。99%的语句、表达式和运算符都与 C++相同
- 具有与 COM+服务完全的互换性
- 提供对 COM 和平台的完全支持能力，使您能够轻松地迁移现有的代码
- 提供自动的无用单元收集能力
- 提供可扩展的分类元数据，使您可以声明新类型和各种元数据
- 对基于网络的组件交互提供 XML 支持

Visual Studio.NET 的新特性使之为基于微软 .NET 框架(Microsoft .NET Framework，微软下一代的网络应用程序开发平台)的应用程序提供了一个完整的开发环境。Visual Studio.NET 提供的关键性基础技术简化了安全、可伸缩、高度可用的网络应用程序和 Web Services 的创建、部署和更新的方法，并为程序员们施展他们的技巧提供了广阔的空间。

除此以外，Microsoft .NET Framework 还帮助网络开发人员使用自己所喜爱的语言，按照使用本地对象的方式来使用 Web Services。程序开发人员可以把注意力投向如何增强产品所特有的功能，以为公司赢得竞争优势。这样的结果是更快的产品开发速度、更高的开发效率和更高的产品质量。

本套丛书的内容与安排

本套丛书取材于由微软出版社提供、独家授权的大量第一手资料，本套丛书目前包括 5 册，分别为：

- 《C#语言参考手册》
  深入详细地介绍 C#语言的语法规则和例外情况，为学习者提供从 C/C++到 C#的快速入门指南。
- 《C#编程技术》
  用大量的程序代码来比较 C#与 C/C++的异同，并具体地说明如何利用 C#访问 Microsoft .NET 类库，编写功能强大的应用程序。
- 《C#技术内幕》
  精心组织成几个逻辑上有序的部分，分别描述了 C#和 .NET 开发环境的一个特定内容：基础篇、C#类基础、编写代码、高级 C#。
- 《Visual Studio .NET 开发环境使用指南》

详细地介绍新一代的应用程序开发环境的使用方法和技巧，为您对 Visual Studio.NET 开发环境的亲身体验提供全面详细的使用指南。

- 《XML 3.0 技术内幕》

  详细深入地介绍最新版本的 XML (Extensible Markup Language，可扩展标记语言)的语言规范和使用方法，帮助读者快速深入地了解这项当今网络的核心技术。

随着技术的发展，并根据读者的需要，我们将陆续增加新的书目。

本套书在编写的过程中面临着如何迅速地把最新的 IT 技术传播给国内的广大读者的巨大压力。迫于时限，在章节篇幅的安排上秉承“形式服从于内容”的宗旨，因而造成某些章节篇幅远远超出其他章节的情况。另外，由于本书所讨论技术的新颖性，使得可资借鉴的中文资料相当稀少，加上编者们的水平有限，故而某些术语的译称可能并不尽如人意，甚至可能还会有不少错误，对于这些问题，恳请广大读者批评指正。

编　者

2001 年 8 月

前　言

写作缘由

做了 20 年程序员之后——每思及此，我就觉得韶光不再！——我基本上已经达到这样一种境界：编程对于我来说，开始显得有些陈腐了。不要错解我的意思：如果我是一个亿万富翁并且不需要工作的话，我仍然可能会继续书写代码，因为这是我真正乐于从事的事情。但现实却是我常常会产生一种“穷途末路”的感觉。终于，Microsoft .NET 和 C#出现在我的面前，一个全新的世界打开了！我曾与其他几个朋友交谈过，他们跟我一样，都被.NET 的美好前景“重新唤醒”。我们终于拥有了这项激动人心的新技术，可以用来解决已经与我们纠缠多年的问题(比如：多语言开发环境、大型复杂系统的分发配置和版本问题等等)。我之所以写这本书，是因为书写代码再一次能够让我激动不已。每天清晨醒来，我都忍不住会想一想今天将要学到很棒的新东西。我希望在您学习这门语言的时候，也能够分享我这份狂热的心情。

目前任何写作有关 C#的人都不得不边写边学。如果你在学习 SDK 或者目前使用的编程语言期间曾经编写过应用程序的话——谁又没有这样做过呢？——你就会知道这事有点难。现在试着想象一下：如果你写了一点东西，而有千千万万的人将要阅读它的时候，你又会怎样呢？最大的问题在于做了一半的时候——一旦你醒悟你在做什么——你就会渴望重新设计整个过程并且重新来过！但是很显然，由于最后期限来临，这样的想法并不实际。我相信本书是一个学习 C#语言的好工具。因为我是边写边学，书中肯定会有一些不妥之处，也有一些地方我本来可以做得更好。但是，如果有机会发行第二版的话，我保证我们都会从我书写本书时我的个人学习过程中受益匪浅，因为，我所遇到的问题也就是您将要面临的问题。

最后，我想说的是我欢迎任何有关本书的反馈意见。我不是那种自认“我写书，所以我很牛气”的人，我只是一个很普通的人，幸运地得到了创作本书的机会而已。我总是乐于向他人学习并且乐此不疲。您可以通过http://www.TheCodeChannel.com与我联系。

本书的读者对象

本书适合于希望了解 C#和.NET 开发环境的读者阅读。我已经说过，.NET 是一个激动人心的平台，同时也是 Microsoft Windows 开发环境未来的潮流。本书假定读者已经具有 C 系列语言之一(C、C++或 Java)的背景知识。如果说还有什么其他的条件，那就是在编写应

用程序时学习和探索新思路的那种渴望。因为您正手握此书，您可能已经心存这种渴望！

本书的组织结构

本书精心组织成逻辑上有序的四个部分：每一部分包含了若干章，描述了 C#或.NET 开发的一大类特定内容。

本书以第 I 部分“基础篇”开始，这部分面向的读者是入门级别的 C#程序员和刚刚接触.NET 开发环境的读者。这部分中的章节提供了对.NET 的介绍，并且说明如何创建和测试您的第一个 C#应用程序。

在第 II 部分“C#类基础”中，介绍了在 C#中定义和使用类的一些基本要素。这部分中的章节着重于为您打下一个坚实的基础，以便了解在 C#中支持哪些成员(枚举、属性、数组等等)，以及在 C#应用程序中如何定义和使用它们。

尽管在学习定义类成员的几个章节中您已经写过代码了，但是直到第III部分“编写代码”，才开始介绍各种任务的不同侧面，诸如控制程序流、处理错误(包括异常)、以及使用代表来书写事件处理器等等。

本书以第IV部分“高级 C#”结束。作为一个典型的老程序员，我最喜欢书写这部分内容。其中包括了多线程编程、反射、使用非控代码(包括 COM 互用性)和版本管理等等。

关于本书配套光盘

本书包括一张配套光盘。如果您在 Windows 中启用了 AutoRun 功能，当您将这张光盘插入到光驱中时就会出现一个屏幕，为您提供安装选项。要想手动打开这个安装屏幕，可以从该光盘的根目录下运行 StartCD 程序。StartCD 程序为您提供到 MSDN 的连接，利用该连接您可以下载最新版本的 Microsoft .NET Framework SDK，在编译和运行示例程序时将用到它。

本书的示例程序位于光盘上的 Code 文件夹中。您可以直接从光盘上查看示例程序，也可以从 StartCD 上使用安装程序将它们安装到您的硬盘上。

注意： 如果您不能浏览 Samples 文件夹中的文件，有可能是因为您的光盘驱动程序太老，以至于不支持长文件名。如果是这样，要想浏览这些文件，您就必须使用安装程序将这些示例文件安装到您的硬盘上。

系统要求

您如果想从本书中最大程度地获益，我强烈建议您在阅读每一章时都一定要演练一遍示例程序。这样的话，您将需要安装最新的.NET Framework SDK。在写作本书期间，它包括.NET 运行环境和 C#编译器。此外，我特意避免使用 Visual Studio.NET 产品，以便集中精力在编程语言和运行环境之上，而不对您特定的开发环境做任何限制。因此，本书中所

有的示例程序都是从命令行编译和运行的。

下面我们再来看一看组成 Microsoft .NET 的关键组件和一些相关的技术。

- **新的编程语言 C#**

  C#是 C 和 C++语言家族中第一种面向组件的编程语言。它从 C 和 C++派生而来，是一种使用简便、面向对象、类型安全的现代编程语言。C#兼具 Visual Basic 的高效性和 C++的强大功能。

- **通用运行语言**

  高性能的通用运行语言(Common Language Runtime, CLR)包含执行引擎、垃圾收集器、即时编译、安全系统和丰富的类框架(即 Microsoft .NET 框架)。CLR 从底层设计，能够支持多种语言。

- **通用语言规范**

  通用语言规范(Common Language Specification, CLS)描述了各种语言所共有的功能。CLS 这种相对很低的限制有助于创建许多与 CLS 兼容的语言。这些语言可获得的好处包括：对 Microsoft .NET 框架功能的完全访问和与其他兼容语言之间充分的互用性。比如，一个 Visual Basic 的类可以从 C#类中继承而来并且覆盖掉其虚拟方法。

- **以 CLR 为目标的大量语言**

  由 Microsoft 提供的以 CLR 为目标的编程语言包括：Visual Basic、带有受控扩展的 Visual C++、Visual C#和 JScript。至于由第三方所提供的编程语言则数不胜数。

- **Web 服务**

  目前的全球互联网主要是由单独的站点所组成。如果用户要访问多个站点以便货比三家，比如为一组人员做旅行计划，那么他所访问的这些 Web 站点之间通常并没有合作。下一代的互联网则是基于 Web 站点之间的合作的网络。这样做的理由很简单：合作的 Web 站点能够更好地满足用户的需求。Microsoft 的 Web 服务技术鼓励 Web 站点之间的合作，因此允许站点之间通过基于 XML 的标准协议进行通信，即使这些 Web 站点在编程语言和应用平台上都相互独立，也不对此造成影响。还有很多重要的 Web 服务将基于 C#和 Windows 上运行的 CLR，但这个框架结构却的确是开放的。

- **Visual Studio.NET**

  Visual Studio.NET 将上述所有的部分联系在一起，以便在各种各样的编程语言中更简便地创建各种各样的组件、应用程序和服务程序。

既然我们已经涉及到了一些与 C#相关的重要技术，那么就对 C#本身进行一些详细的探讨吧。开发人员已经为他们所选择的编程语言投入了大量的心血，因此新语言的责任就是通过保留简单、持续改进和谨慎创新的组合来证明自己的价值。

简单保留

希波克拉底说过："要养成这样的习惯：要么有所帮助，要么至少没有伤害"。在我们设计 C#的过程中，"没有伤害"这个原则起到了非常重要的作用。如果一个 C 或 C++的功能将问题描述得很清楚，我们就会全盘保留。更重要的是，C#在很多关键的领域都借用了 C 和 C++中的要素，如表达式、语句和整个的语法。由于在很多典型程序中都会包含这些要素，因此 C 和 C++开发人员定会对 C#一见倾心。

持续不断的改进

持续改进的地方简直太多了，以至于在这个简短的前言中无法一一陈述。但以下这几个改变的地方值得引起关注，因为它们消除了一些在 C 和 C++中很常见并且很耗时的错误：

- 变量在使用之前必须初始化，因此消除了由于变量未初始化而导致的故障。
- 类似 if 和 while 的语句要求使用布尔值。开发人员如果意外地使用了赋值操作符(=)代替相等操作符(==)的话，在编译时就能发现错误。
- 在 switch 语句中不允许出现贯行(fall-through)。开发人员如果意外地省略了 break 语句的话，在编译时就会发现错误。

深思熟虑的创新

在 C#的类型系统中可以发现一些深层次的创新，包括以下这些部分：

- C#类型系统采用了自动内存管理，因此将开发人员从耗费时间和容易出错的手动内存管理中解放出来。与大多数类型系统不同，C#类型系统还允许对指针类型和对象地址进行直接操作(这些手动内存管理技术只允许在某些特定的安全环境下使用)。
- C#类型系统是统一的——所有的类型都是对象。通过对"装箱"和"卸箱"这些概念的创新使用，C#在值类型和引用类型之间架起了一道桥梁，允许将任何数据都当作对象看待。
- 特性、方法和事件是基本要素。很多语言都省略了对特性和事件的支持，因此造成在编程语言和相关框架之间不必要的不匹配。比如，如果框架支持特性而编程语言并不支持特性，那么给特性做增量就会很麻烦(如：o.SetValue(o.GetValue()+1))。如果该语言也支持特性，操作就简单多了(o.Value++)。
- C#支持属性，允许对组件的描述性信息进行定义和使用。能够定义新种类的描述性信息对于语言设计人员来说总是一个强大的工具。现在所有的 C#开发人员都拥有这个能力了。

亲爱的读者，衷心地希望你们能愉快地写下第一个 C#程序，并且在以后的日子里写出更多更好的 C#程序。

目　录

第Ⅰ部分　基础篇 1

第 1 章　面向对象编程的基础知识 1

1.1　一切都是“对象” 2

1.1.1　对象和类 5

1.1.2　实例化 6

1.2　面向对象编程语言的三大原则 7

1.2.1　封装 8

1.2.2　继承 10

1.2.3　多态性 13

1.3　本章小结 16

第 2 章　Microsoft .NET 简介 17

2.1　Microsoft .NET 平台 17

2.2　.NET 框架 18

2.2.1　Windows DNA 和.NET 18

2.2.2　公共语言运行环境 18

2.2.3　.NET 框架类库 19

2.2.4　Microsoft 中间语言和 JITters 21

2.2.5　统一的类型系统 22

2.2.6　元数据和反射 23

2.2.7　安全问题 24

2.2.8　部署 24

2.2.9　与非控代码的互用性 25

2.3　本章小结 26

第 3 章　C#应用程序简介 27

3.1　编写第一个 C#应用程序 27

3.1.1　选择编辑器 27

3.1.2　“Hello, World” 29

3.1.3　使用命令行编译器 30

3.1.4　运行应用程序 31

3.2　代码详解 32

3.2.1　“直达式”编程 32

3.2.2　类和成员 32

3.2.3　Main 方法 33

3.2.4　System.Console.WriteLine 方法 33

3.2.5　名称空间和指示符 using 33

3.2.6　骨架代码 35

3.3　出错情况 36

3.4　ILDASM 研究 37

3.5　C#编程原则 41

3.5.1　何时定义您自己的名称空间 41

3.5.2　命名原则 41

3.5.3　命名规范标准 42

3.6　本章小结 44

第Ⅱ部分　C#类基础 47

第 4 章　类型系统 47

4.1　一切都是对象 47

4.2　数值类型和引用类型 48

4.2.1　数值类型 48

4.2.2　引用类型 48

4.3　装箱与开箱 49

4.4 所有类型的根：System.Object 50
4.5 类型和别名 51
4.6 类型之间的转换 52
4.7 名称空间 54
4.8 CTS 的好处 56
4.8.1 语言互用性 56
4.8.2 单根的对象层次结构 56
4.8.3 类型安全 57
4.9 本章小结 57

第 5 章 类 58

5.1 定义类 58
5.2 类的成员 58
5.3 访问限定符 59
5.4 Main 方法 61
5.4.1 命令行参数 61
5.4.2 返回值 62
5.4.3 多个 Main 方法 63
5.5 构造函数 64
5.5.1 静态成员和实例成员 66
5.5.2 构造函数的初始化函数 67
5.6 常量和只读字段 70
5.6.1 常量 71
5.6.2 只读字段 71
5.7 对象清除和资源管理 73
5.7.1 历史回顾 74
5.7.2 确定性结束 74
5.7.3 性能 76
5.7.4 完美的解决方案 81
5.7.5 近乎完美的解决方案 82
5.7.6 Dispose 设计模式 83
5.8 继承 84
5.8.1 多接口 86
5.8.2 封装类 87
5.9 本章小结 88

第 6 章 方法 89

6.1 方法参数“ref”和“out” 89
6.2 方法重载 94
6.3 可变的方法参数 96
6.4 虚拟方法 98
6.4.1 方法覆盖 98
6.4.2 多态性 99
6.5 静态方法 106
6.6 本章小结 108

第 7 章 属性、数组和索引器 109

7.1 属性——智能字段 109
7.1.1 定义和使用属性 110
7.1.2 编译器的工作原理 111
7.1.3 只读属性 113
7.1.4 继承属性 114
7.1.5 属性的高级使用 114
7.2 数组 115
7.2.1 声明数组 115
7.2.2 一维数组 116
7.2.3 多维数组 117
7.2.4 查询秩 119
7.2.5 锯齿状数组 120
7.3 使用索引器将对象当作数组对待 122
7.3.1 定义索引器 123
7.3.2 索引器示例程序 123
7.3.3 设计规则 125
7.4 本章小结 126

第 8 章 特性 127

8.1 特性的介绍 127
8.2 特性的定义 128

8.3 对特性进行查询 130
8.3.1 类的特性 130
8.3.2 方法的特性 132
8.3.3 字段的特性 134
8.4 特性的参数 136
8.4.1 定位参数和命名参数 136
8.4.2 使用命名参数时的常见错误 138
8.4.3 有效的特性参数类型 138
8.5 AttributeUsage 特性 139
8.5.1 定义一个特性目标 139
8.5.2 单次特性和多次特性 141
8.5.3 指定继承特性的规则 142
8.6 特性标识符 143
8.7 本章小结 144
第 9 章 接口 145
9.1 接口的应用 145
9.2 声明接口 147
9.3 实现接口 148
9.3.1 使用 is 来查询实现 150
9.3.2 使用 as 来查询实现 153
9.4 显式的接口成员名字限定 157
9.4.1 接口的名字隐藏 157
9.4.2 避免名字模糊性 160
9.5 接口和继承 164
9.6 合并接口 167
9.7 本章小结 169
第III部分 编写代码 171
第 10 章 表达式和操作符 171
10.1 已定义的操作符 171
10.2 操作符的优先级 172
10.2.1 C#如何判断优先级 172
10.2.2 左联合性和右联合性 173
10.2.3 实际使用 174
10.3 C#操作符 175
10.3.1 初级表达式操作符 175
10.3.2 数学操作符 179
10.3.3 关系操作符 188
10.3.4 简单赋值操作符 190
10.4 本章小结 193
第 11 章 程序流程控制 194
11.1 选择语句 194
11.1.1 if 语句 194
11.1.2 多个 else 子句 195
11.1.3 C#如何强制使用 if 规则 197
11.1.4 switch 语句 199
11.1.5 组合情况标签 201
11.1.6 在 switch 语句中没有“贯行” 203
11.2 迭代语句 204
11.2.1 while 语句 204
11.2.2 do/while 语句 206
11.2.3 for 语句 208
11.2.4 嵌套循环 209
11.2.5 使用逗号操作符 210
11.2.6 foreach 语句 211
11.3 跳转语句 213
11.3.1 break 语句 213
11.3.2 中断无穷循环 214
11.3.3 continue 语句 216
11.3.4 “声名狼藉”的 goto 语句 217
11.3.5 return 语句 223
11.4 本章小结 223

第 12 章 用异常进行错误处理 224
12.1 异常处理概览 224
12.2 异常处理的基本语法 225
12.2.1 抛出异常 225
12.2.2 截获异常 226
12.2.3 重抛异常 227
12.2.4 用 finally 进行清除 228
12.3 错误处理技术之比较 229
12.3.1 异常处理相对于返回代码的优点 229
12.3.2 在正确的上下文环境中处理错误 231
12.3.3 提高代码的可读性 232
12.3.4 从构造函数抛出异常 234
12.4 使用 System.Exception 类 234
12.4.1 构造一个 Exception 对象 ... 235
12.4.2 使用 StackTrace 属性 237
12.4.3 截获多个异常类型 238
12.4.4 派生出自己的 Exception 类 239
12.5 设计具有异常处理功能的代码 241
12.5.1 使用 try 块时的设计注意事项 241
12.5.2 使用 catch 块时的设计注意事项 243
12.6 本章小结 245
第 13 章 操作符重载和用户定义的转换 246
13.1 操作符重载 246
13.1.1 语法和例子 247
13.1.2 可重载操作符 250
13.1.3 操作符重载的限制 250
13.1.4 设计准则 250
13.2 用户定义的转换 251
13.3 本章小结 257
第 14 章 代表和事件处理器 258
14.1 将代表用作回调方法 258
14.2 把代表定义为静态成员 261
14.3 仅在需要时创建代表 263
14.4 代表构成 265
14.5 定义具有代表的事件 272
14.6 本章小结 276
第Ⅳ部分 高级 C# 277
第 15 章 多线程编程 277
15.1 线程的基本知识 277
15.1.1 线程与多任务 278
15.1.2 上下文切换 278
15.2 C#语言中的多线程应用程序 278
15.3 线程处理 280
15.3.1 AppDomain 280
15.3.2 Thread 类 280
15.3.3 线程调度 284
15.4 线程安全和同步 288
15.4.1 使用 Monitor 类保护代码 ... 288
15.4.2 用 C#中的 lock 语句使用 Monitor 锁 293
15.4.3 通过使用 Mutex 类同步化代码 294
15.4.4 线程安全和.NET 类 297
15.5 使用线程的原则 297
15.5.1 何时使用线程 297
15.5.2 何时不应该使用线程 298
15.6 本章小结 299

第 16 章 利用反射查询元数据 300
16.1 反射 API 层次 300
16.2 Type 类 301
16.2.1 检索一个实例的类型 301
16.2.2 根据名字检索类型 301
16.2.3 查询类型 302
16.3 使用组合体和模块 304
16.3.1 迭代组合体的类型 305
16.3.2 列出组合体的模块 307
16.4 用反射进行后期绑定 309
16.5 在运行时刻创建并执行代码 312
16.6 本章小结 316
第 17 章 与非控代码互操作 317
17.1 平台启用服务 317
17.1.1 声明导出的 DLL 函数 317
17.1.2 与 C#共用回调函数 320
17.1.3 汇集和 PInvoke 321
17.2 编写非安全代码 322
17.2.1 在 C#中使用指针 323
17.2.2 fixed 语句 324
17.3 COM 互操作性 326
17.3.1 一个全新的领域 326
17.3.2 入门 326
17.3.3 从 COM 类型库生成元数据 328
17.3.4 早期绑定 COM 组件 330
17.3.5 利用动态类型探索选择 COM 接口 332
17.3.6 后期绑定 COM 组件 333
17.3.7 COM 线程模型 335
17.4 本章小结 336
第 18 章 使用组合体 338
18.1 组合体概览 338
18.2 组合体的优点 339
18.2.1 组合体的打包 340
18.2.2 组合体的部署 340
18.2.3 组合体的版本管理 340
18.3 建立组合体 341
18.4 创建共享组合体 343
18.5 使用全局组合体缓存区 345
18.6 组合体的版本管理 347
18.6.1 QFE 和默认版本管理策略 350
18.6.2 创建安全模式部署文件 350
18.6.3 使用组合体的特定版本 351
18.7 本章小结 352

第Ⅰ部分　基础篇

第1章　面向对象编程的基础知识

本章的目的是带领您熟悉面向对象编程(Object-Oriented Programming, OOP)的各种术语，并使您认识到面向对象的概念对于编程的重要性。许多编程语言，诸如 C++和 Microsoft Visual Basic，都声称自己是“支持对象”的，但实际上只有少数几种语言真正完全支持构成面向对象编程的所有原则。C#就是其中一种，它从最底层开始设计，是一种真正面向对象、基于组件的编程语言。因此，为了最大程度地从本书获益，您需要牢固掌握本章提出的概念。

读者通常会跳过类似这样概念性的章节而一头扎入代码之中。但是，除非您认为自己已经是“对象通”，否则，建议您还是先认真阅读本章。对于那些对面向对象编程似是而非的读者来说，肯定能从本章中学有所获。另外，别忘了本章后面的章节也会引用本章讨论的术语和概念。

前面已经说过，许多语言都被称为是面向对象或基于对象的，但实际上只有少数几种才真正是。C++不是，因为它的根深植于 C 语言之中，这是不可否认、无法回避的事实。在 C++中为了支持传统的 C 代码，已经牺牲了太多的 OOP 理想。即便是 Java 语言，尽管它已经够好了，但作为一种面向对象的语言，它仍然是有限的。比如，作者可以指出这样一个事实：在 Java 中存在简单类型和对象类型，它们的处理方式和行为模式都非常不同。但本章的重点并不是比较不同的语言对 OOP 原则的忠诚度，而是提供一个关于 OOP 原则本身客观的、与语言无关的指南。

在开始之前，需要指出的是：面向对象编程并非一种营销的招牌(尽管对于某些人来说是)，也不是一种新的语法或一种新的应用编程接口(Application Programming Interface, API)。面向对象编程是一套全新的概念和想法。它是一种用计算机程序来描述实际问题的新思路，也是一种更直观、效率更高的解决问题的方法。

作者的第一份工作是用 Pascal 语言为《冰上假日》的票房报告和巡演路线编程。在后来其他的工作中，又用到了 PL/I 和 RPG III(以及 RPG/400)编程。又过了几年，作者开始使

用C语言编写应用程序。在这些经历中，作者可以非常轻松地运用以前学到的知识。后续每种语言不论其复杂程度，学习过程都是越来越短。这是因为直到开始用C++编程为止，作者用到的所有语言都是一种结构化语言，主要的区别仅仅在于语法不同而已。

不过，如果您是初次接触面向对象编程的话，您应当注意：您以前学过的非面向对象语言在这里不会有所帮助！面向对象编程在如何设计和实现问题的解决方案时，采取的是一种完全不同的思路。事实上有研究表明，在学习面向对象语言时，编程新手比学过诸如BASIC、COBOL和C这些结构化语言的人要快得多。因为他们不需要克服任何有碍于理解OOP的结构化积习，他们就像一张白纸，是从头学起。如果您已经具有使用结构化语言编程的多年经验，而C#又是您接触的第一种面向对象语言的话，您能得到的最好的建议就是保持开放的心胸，并且努力实现作者在这里提出的想法，而不是在心中暗想："哼，我也能做到(当然是使用结构化语言)！"任何一个从结构化背景转向面向对象编程的人都会经历这样一个过程，而这也是值得的。使用面向对象语言编程的好处是无法估量的，无论是更有效地书写代码方面，还是构造一个更易修改和扩展的系统方面。这样的好处一开始可能看不到，但以作者将近20年的软件开发经验(包括过去8年使用面向对象语言在内)而言，作者认为只要正确运用，OOP的概念的确能够实现它们的承诺。

1.1 一切都是"对象"

在真正的面向对象语言中，所有的问题域(problem domain)实体都是通过对象(Object)的概念表达出来的(注意在本书中将采用Coad/Yourdon定义来描述"问题域"——也就是说，问题域就是您试图去解决的问题，包括每个问题的特殊复杂性、描述用语和解决难度等等)。您也许已经猜到，对象就是面向对象编程的核心所在。在面向对象编程中，我们不会拐着弯儿地去想什么结构、数据包、函数调用和指针，通常只考虑对象这个概念。先来看一个例子。

假设您在编写一个开发票的应用程序，您需要对发票上的明细栏进行计算。从用户的角度出发，以下哪一种思路是更直观的呢？

- 非面向对象思路

  用一个数据结构来表示发票的抬头，这个发票的抬头结构包括一个指向发票明细栏结构的双向链表(linked list)，每个发票明细栏结构包含该明细栏的小计。这样，要计算出发票的总计，需要再声明一个变量，就把它叫做totalInvoiceAmount吧，并将它初始化为0；使用一个指针指向发票的抬头结构，得到明细栏链表的头指针，然后遍历这个链表；访问每个明细栏结构时，将包含该明细栏小计的成员变量值加入到totalInvoiceAmount变量之中。

- 面向对象思路

  使用一个发票对象，发送一条消息给该对象，要求得到发票总计。我们并不需要考虑在对象内部信息是怎样存储的，而在非面向对象数据结构中却必须考虑这一点。我们只需以一种自然的方式来对待这个对象，也就是用发送消息的方式来向它做出请求(对象能处理的消息综合起来叫做该对象的接口，下面将解释在面向对象思路中为什么只考虑接口而不是具体实现是合理的)。

显然，面向对象思路更直观，也更接近于大多数人解决问题的方式。在面向对象思路中，发票对象可能会包含多个发票明细栏对象，并对每个明细栏对象发送消息要求得到其小计。但是，尽管您需要得到总计，但却不必关心它到底是怎样得到的。这是因为面向对象编程的一个主要原则就是封装——对象能够将其内部的数据和方法隐藏起来并且提供一个接口，这个接口就是能够访问该对象的重要组成部分。只要对象能够完成任务，在其内部究竟怎样完成任务就不重要了。您只需了解对象的接口部分，然后利用接口使对象执行给定的任务即可(本章的后续部分还将进一步解释封装和接口的概念)。上述的例子表明了这样一道理：编程时如果能够对问题域的现实世界对象加以模拟，程序编写起来将会十分容易，因为我们的思路更加贴近正常的思维方式。

注意在第二种思路中要求对象执行给定的任务——也就是计算出明细栏的总计。一个对象并不只包含数据，而结构却只包含数据。从定义上来看，对象由数据和加工数据的方法组成。这意味着对于一个问题域，我们能做的就不只是设计必要的数据结构了。我们还可以看看哪些方法可以附加在给定的对象上面，以便该对象成为功能齐全的封装结构。下面的示例以及后面章节中的示例都有助于说明这个概念。

注意： 本章中的代码片断用于展示面向对象编程的概念。要注意书中出现 C#代码片断时，这些概念本身对于 OOP 来说都是通用的，并不只特定于某一种编程语言。在本章中为了便于比较，也列出了 C 语言代码，而它并不是面向对象的。

比如说您正在编写一个应用程序，用来计算您的新公司中惟一雇员 Amy 的薪酬。如果使用 C 语言，您可能这样书写代码以将相关的数据与雇员联系起来：

```
struct EMPLOYEE
{
    char szFirstName[25];
    char szLastName[25];

    int iAge;

    double dPayRate;

};
```

下面是使用 EMPLOYEE 结构来计算 Amy 薪酬的代码：

```
void main()
{
    double dTotalPay;

    struct EMPLOYEE* pEmp;
    pEmp = (struct EMPLOYEE*)malloc(sizeof(struct EMPLOYEE));

    if (pEmp)
    {
        pEmp->dPayRate = 100;

        strcpy(pEmp->szFirstName, "Amy");
        strcpy(pEmp->szLastName, "Anderson");
        pEmp->iAge = 28;

        dTotalPay = pEmp->dPayRate * 40;
        printf("Total Payment for %s %s is %0.2f",
               pEmp->szFirstName, pEmp->szLastName, dTotalPay);
    }

    free(pEmp);
}
```

在这个示例中，代码包含了结构中的数据和一些使用该结构的外部代码(相对于该结构而言)。这样会出现什么问题呢？主要问题之一就在于抽象：EMPLOYEE 结构的使用者必须了解很多有关于雇员的数据。为什么？比如说以后如果您想要修改计算 Amy 薪酬的税率的话，假设您想将美国联邦社会保险捐款 FICA 和其他各种税赋考虑进来以便决定税后工资，这时您不仅必须修改所有使用 EMPLOYEE 结构的代码，而且需要做出文档记载——为您公司将来任何一位程序员——说明该程序在使用中已经发生过改变。

下面我们来看看 C#是怎样解决这个问题的：

```
using System;

class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
```

```
            this.payRate = payRate;
        }

        protected string firstName;
        protected string lastName;
        protected int age;
        protected double payRate;

        public double CalculatePay(int hoursWorked)
        {
            // Calculate pay here.
            return (payRate * (double)hoursWorked);
        }
    }

    class EmployeeApp
    {
        public static void Main()
        {
            Employee emp = new Employee ("Amy", "Anderson", 28, 100);
            Console.WriteLine("\nAmy's pay is $" + emp.CalculatePay(40));
        }
    }
```

在上面这个 C# 版本的 EmployeeApp 示例中，使用对象的程序员只需简单地调用该对象的 CalculatePay 方法就能计算出该对象的薪酬。这种思路的优点就是使用者不再需要担心具体薪酬是怎样计算出来的。如果将来您决定修改计算薪酬的方法，那么做出的修改对于已存在的代码不会有任何影响。这种程度的抽象就是使用对象的一个基本好处。

可能有人会说，创建一个访问 EMPLOYEE 结构的函数也可以实现 C 语言代码的抽象。然而问题就在于创建的这个函数与该函数要处理的数据结构是完全分离的。而使用 C#这样的面向对象语言时，对象的数据和加工数据的方法(即对象的接口)总是在一起。

另外要记住只有对象的方法才能修改对象的变量。如前面的示例中所见，每个 Employee 成员变量声明时都带有“protected”访问限定符，除了定义为“public”的方法 CalculatePay 以外。访问限定符用来指定派生类和客户代码可以访问给定类成员的级别。如果指定“protected”限定符，则派生类可以访问该成员，而客户代码则不能访问。限定符“public”表示派生类和客户代码都可以访问该成员。在第 5 章中我们还将详细讨论访问限定符，现在要记住的关键问题就是限定符可以保护关键的类成员不被误用。

1.1.1 对象和类

类和对象之间的区别常常是刚接触面向对象编程的程序员们容易混淆的地方。为了说

明这两个术语之间的不同，我们使 EmployeeApp 示例更现实一些，假设不是处理一个雇员的数据，而是处理整个公司所有雇员的数据。

使用 C 语言，我们可以定义一个基于 EMPLOYEE 结构的雇员数组。因为我们并不清楚将来公司到底会有多少雇员，只好创建一个包含固定个数元素的数组，假设其大小是 10000。但是，如果目前公司只有 Amy 这一个雇员的话，这样使用资源就显得很浪费。因此，我们通常会创建一个 EMPLOYEE 结构的链表，以便在应用程序中动态分配内存。

但是，我们现在正在做的可能恰恰是不应该由我们做的。我们在语言和机器——即分配多大的内存和何时分配内存——上绞尽脑汁，而不是专注于问题域本身。使用对象则有助于我们将精力集中在解决问题必需的逻辑思考上。

要定义一个类并将它与对象区别开来有多种方法。您可以将类简单地看作是一个拥有相关方法的类型(就像 char、int 或 long 一样)，而对象就是类型或类的一个实例。但作者本人最喜欢的定义却是：类是对象的一个蓝图。作为开发者的您描绘出这幅蓝图，就像工程师描绘出房子的蓝图一样。一旦完成蓝图，您就可以按照蓝图建造多幢房子。同时，别人也可以购买您的这幅蓝图从而建造相同的房子。同样的道理，类就是给定功能集的蓝图，基于特定类创建的对象就拥有该类集成的所有功能。

1.1.2 实例化

实例化(instantiation)是面向对象编程的专用术语，表示创建类的一个实例。这个实例就是一个对象。在下面的例子中，我们所做的一切就是为对象创建一个类。换句话说，这个阶段不涉及内存分配，因为我们只拥有对象的一个蓝图，而没有实际的对象。

```
class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }

    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public double CalculatePay(int hoursWorked)
```

```
        {
            // Calculate pay here.
            return (payRate * (double)hoursWorked);
        }
    }
```

为了实例化这个类并使用它，我们还必须在方法中声明一个该类的实例：

```
public static void Main()
{
    Employee emp = new Employee ("Amy", "Anderson", 28, 100);
}
```

在这个例子中，emp 被声明为 Employee 类型并且使用操作符“new”将它实例化。变量 emp 就表示了 Employee 类的一个实例并被看作是一个 Employee 对象。实例化之后，我们就可以通过其公共成员来与这个对象进行通信了。比如，我们可以调用 emp 对象的 CalculatePay 方法，而在没有实际对象之前我们不能这样做(有一个例外，那就是处理静态成员时。在第 5 章和第 6 章中我们将讨论静态成员)。

下面我们来看看以下 C#代码：

```
public static void Main()
{
    Employee emp = new Employee();
    Employee emp2 = new Employee();
}
```

这里我们有同一个 Employee 类的两个实例——emp 和 emp2。从表面上看，每个对象的功能相同，每个实例都可以包含它自己的实例数据并被单独处理。同样的道理，我们可以创建一个包含这些 Employee 对象的数组或集合。第 7 章中将详细讨论数组。作者在这里想说的是大多数面向对象的语言都能够定义对象数组。这使得您能够很轻松地组织对象、调用对象数组的方法或者使用数组的下标。将这与链表比较一下吧：在链表中您必须自己将链表中的每个条目与它前后的条目一一连接起来。

1.2　面向对象编程语言的三大原则

C++编程语言的作者 Bjarne Stroustrup 认为，一种语言必须支持对象、类和继承这三个概念，才可以称得上是面向对象的。但实际上，面向对象语言已经被更普遍地认为是建立在封装、继承和多态性这三者之上的。之所以发生这种观念上的转变，是因为这么多年以来，我们逐渐认识到封装和多态性与类和继承一样，是构成面向对象系统不可缺少的组成部分。

1.2.1 封装

在前面提到过，“封装”，有时也叫做“信息隐藏(information hiding)”，能够将对象的内部隐藏起来，只提供一个接口给对象的使用者，使得只有那些能被直接操作的成员才能被使用。但同时也提到了“抽象”这个概念，在这一节里将澄清有关这两个相似概念之间容易混淆的地方。封装提供的是一道屏障，将类的外部接口(也就是类的使用者能看到的公共成员)与它的内部实现细节分隔开来。封装的优点在于类的开发者可以将静态的、不变的成员公开，而将那些动态的、容易改变的类成员隐藏起来。如本章先前所示，在C#中通过赋予每个类成员访问限定符public、private或者protected就可以实现封装。

1.2.1.1 抽象设计

“抽象”指的是在程序空间中一个给定的问题是如何表示出来的。编程语言本身就提供抽象。试想：您上一次考虑如何使用CPU的寄存器和堆栈是什么时候？即使您最初学的是用汇编语言编程，您也一定有很长时间没有考虑过这样底层而且是针对特定机器的细节了。这是因为大多数编程语言都已将这些细节为您抽象出来了，使得您可以集中精力于要解决的问题域本身。

面向对象语言使得声明的类所具有的名称和接口都类似于现实世界中的问题域实体，因此使用对象时就有了更加自然的“感觉”。去掉那些与当前解决问题没有直接关联的因素之后，您就可以专注于问题本身，工作效率自然也就更高。但实际上，如Bruce Eckel在Thinking in Java (Prentice Hall Computer Books, 2000)一书中所说，解决大多数问题的能力通常会由所采用的抽象质量决定。

当然，那是另外一个层次上的抽象。更进一步，作为一个类开发人员，您必须考虑到如何设计出最好的抽象，以便用户在使用类时能够集中精力于当前要解决的问题，而不陷入您的类怎样工作这样的细节之中。从这一点上说，“类的接口怎样与抽象关联”是个很好的提问。类的接口实际上就是抽象的实现。

下面将讲一个大家熟悉的编程课上的类比，以便明确这些概念。这个类比就是自动售货机的内部工作原理。自动售货机的内部实际上非常复杂，为了完成它的职责，自动售货机必须能够接受现金和货币，找零钱，找到并送出顾客想要的东西。然而，自动售货机需要展示给顾客的却只是有限的几个功能。通过一个硬币槽、选择货品的按钮、要求找零的拉杆、放零钱的槽和发送选定物品的出口，即可表达出自动售货机展现给顾客的接口。以上每一项都表示了该机器接口的一部分。自动售货机自发明以来，大体上都保留了这样相同的功能。尽管随着技术的进步，自动售货机的内部也发生了很大的变化，但是其基本的接口部分却并不需要改变太多。设计类的接口时很重要的一步就是要深入彻底地理解问题

域本身。这种理解将有助于您创建的接口既能让用户访问他们需要的信息和方法，同时也能将他们隔离在类的内部工作之外。设计出来的类不仅要能解决今天的问题，同样也要从类的内部中充分地抽象出来，以便私有类成员能够在不影响已存在代码的前提下自由修改。

设计类的抽象概念时另一个同等重要的方面是始终考虑客户程序员的方便。假设您设计了一个通用的数据库引擎，如果您是一个数据库方面的专家，您可能对游标、委托控制和元组这样的术语会很熟悉。但是，大多数不熟悉数据库编程的开发人员可能对这些术语就不甚了解。使用这些对类的使用者而言很陌生的术语的行为，其实完全违背了抽象的目的——即通过用自然术语代表问题域以便提高程序员的工作效率。

另外一个需要考虑到用户的地方是决定哪些类成员应该被公共访问。同样，这时应该多一点对问题域的理解和多为类的用户想想。仍以数据库引擎为例，您可能不会希望用户直接访问表示内部数据缓冲器的成员，这样将来就能轻易地更改这些数据缓冲器的定义。另外，由于这些缓冲器对于引擎整个的操作至关重要，您可能就会确保只能通过您的方法来修改它们，这样就能保证采取必要的预防措施。

注意： 您可能会想到面向对象系统主要是为方便创建类而设计的。尽管这个特点的确能够带来生产效率的短期提高，但是只有认识到引入 OOP 的目的在于为使用类的程序员提供方便以后，才能带来长期效益。因此在设计类的时候，始终不要忘记考虑那些将要实例化您所创建的类或从您创建的类中派生类的那些程序员。

1.2.1.2　良好抽象的益处

开发可复用(reusable)的软件过程中，将类抽象成一种对将使用这些类的程序员最有用的方式是极其重要的。如果开发的接口在实现方式改变之后仍能保持稳定和不变的话，那么随着时间的推移，您的应用程序也就很少需要修改。还是以先前的计算薪酬的示例代码为例，在 Employee 对象和计算功能中，只有少数几个方法是相关的，如：CalculatePay、GetAddress 和 GetEmployeeType。如果您熟悉实际的计算薪酬方法，在很大程度上就能轻松地确定使用这个类的用户将需要哪些方法。既然如此，如果在设计这个类的时候，您把对问题域的理解与远见融合在设计之中的话，当然就有理由保证在这个类的实际实现中，不管将来如何变化，类的大部分接口仍然可以保持不变。总之，从用户的角度看来，它就是一个 Employee 类而已。而站在用户的立场上，当然也是希望在版本更迭过程中类最好不要有什么变化。

将用户和实现细节分离还能使得整个系统易于理解和维护。在结构化语言，如 C 中，每个模块都需要一个明确的名称并且要访问一些给定结构的成员，这样，每当这些结构的成员发生变化时，引用这些结构的每一行代码都必须随之改变。

1.2.2 继承

“继承”指的是程序员指定的一个类与另一个类之间的某种关系。通过继承，您可以创建(或派生)一个新的类，而它以已存在的类为基础。然后您可以随心所欲地修改这个新类并且创建这个派生类型的新的对象。这个功能就是构建类层次结构的精华所在。除了抽象，在系统整体设计中继承就是最重要的部分了。派生类是被创建的新类，基类就是新类从其派生的那个类。新创建的派生类继承了基类的所有成员，因此您可以重新利用以前的工作成果。

注意： 在C#中，哪些基类成员将被继承取决于成员定义时所用的访问限定符。在第5章中我们将详细讨论。这里为了便于讨论，您可以暂时认为派生类将继承其基类的所有成员。

还是以先前的EmployeeApp为例，我们来看看何时且如何使用继承。在那个示例中，我们可能会雇用不同类型的雇员，如正式工、合同工和小时工。虽然所有这些Employee对象的接口一样，但在内部它们的功能却会有很大的不同。比如，CalculatePay方法对正式工和合同工的计酬方法就不一样，但我们还是希望不管雇员是何种类型，他们的CalculatePay接口仍然一样。

如果您是刚接触面向对象编程，您可能就会想到：“这里我为什么要用对象？为什么不用一个带有雇员类型成员的EMPLOYEE结构和一个类似的函数？”

```
Double CalculatePay(EMPLOYEE* pEmployee, int iHoursWorked)
{
    // Validate pEmployee pointer.

    if (pEmployee->type == SALARIED)
    {
        // Do W-2 employee processing.
    }
    else if (pEmployee->type == CONTRACTOR)
    {
        // Do 1099 processing.
    }
    else if (pEmployee-> == HOURLY)

    {
        // Do hourly processing.
    }
    else
```

```
        {
            // Do corp-to-corp processing.
        }

        // Return the value from one of the
        // compound statements above.
    }
```

这段代码存在几个问题。首先，CalculatePay 函数是否能成功调用与 EMPLOYEE 结构关系紧密。前面已经说过，这样紧密的关系会导致 EMPLOYEE 结构的任何变化都会影响到这段代码。作为面向对象的程序员，最不想做的事情就是给类的用户增加深入了解设计类的细节这样的负担，这就好比是自动售货机生产厂家要求您在购买汽水之前了解自动售货机的内部工作原理一样。

其次，这段代码不便于复用。一旦您开始了解继承是如何为复用提供方便之后，您就会意识到类和对象确实不错。在这里，您只需简单地定义基类的所有成员，而不用考虑雇员的类型。派生类会继承这些成员然后做出必要的修正。下面来看看 C#是怎样解决这个问题的：

```
class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }

    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public double CalculatePay(int hoursWorked)
    {
        // Calculate pay here.
        return (payRate * (double)hoursWorked);
    }
}

class SalariedEmployee : Employee
{
```

```
        public string SocialSecurityNumber;

        public void CalculatePay (int hoursWorked)
        {
            // Calculate pay for a W-2 employee.
        }
    }

    class ContractEmployee : Employee
    {
        public string FederalTaxId;

        public void CalculatePay (int hoursWorked)
        {
            // Calculate pay for a contract employee.
        }
    }
```

这个例子中的三个特点值得关注：

- 基类 Employee 中定义了一个名为 EmployeeId 的字符串，它被 SalariedEmployee 和 ContractEmployee 两个类所继承，但这两个派生类与这个成员一点关系也没有——它们只是从 Employee 类中派生时顺带地自动继承了它。
- 两个派生类都实现了它们自己的 CalculatePay。但是，您会注意到它们都继承了同样的接口，因此，尽管它们根据自己的需要修改了内部的代码，但是用户的代码却无需改变。
- 两个派生类都向从基类继承的成员中添加了新的成员。SalariedEmployee 类定义了一个 SocialSecurityNumber 字符串，而 ContractEmployee 类则包含了一个对 FederalTaxId 成员的定义。

从这个小小的示例中您已经看到：继承使得您可以通过从基类中继承功能而实现代码复用；更进一步，它使您可以通过添加自己的变量和方法来扩展类。

定义适当的继承

为了说明适当继承的重要性，我们将引用 Marshall Cline 和 Greg Lomow 合著的《C++FAQs》(Addison-Wesley,1998)中的一个术语：可替代性(substitutabiolity)。可替代性表示派生类的已通告行为对于基类来说是可替代的。仔细想想这句话——这就是您在学习创建有效的类层次结构时最重要的一个规则(“有效”表示经受得住时间的考验并且实现 OOP 可以复用和扩展代码的承诺)。

在创建类层次结构时另一个需要牢记的重要规则是：派生类在对待继承的接口时应该不比它的基类要求多而承诺少。不遵守这条规则将破坏已存在的代码。类的接口就是类和

使用这个类的程序员之间的一个捆绑合同。当程序员引用派生类时，总是可以将该派生类当作其基类一样看待。这就叫做“向上转换”(upcasting)。在我们先前的例子中，如果用户引用了一个 ContractEmployee 对象，那么它实际上也隐性引用了该对象的基类——Employee 对象。因此，按照定义，ContractEmployee 总是应该完成其基类能够完成的功能。请注意这条规则只应用于基类的功能。派生类可以选择添加功能，而随着它的要求和承诺增加，这些功能可能会越来越有限。因此，这条规则只应用于继承的成员，因为已存在代码只与这些成员之间有合同限制。

1.2.3　多态性

作者所知的关于多态性(polymorphism)最好和最简洁的定义是：多态性就是允许老代码调用新代码的能力。这当然是面向对象编程最大的好处了，因为这样就可以在不修改或破坏已存在代码的前提下扩展或加强您的系统。

现在假设您编写了一个方法需要遍历一组 Employee 对象，调用每个对象的 CalculatePay 方法。在您的公司只有一种雇员类型的时候，这个方法很有效，因为每当增加雇员时您只需将确切的对象类型加入到集合中即可。但是，如果您开始雇用别的类型的雇员时情况将会怎样呢？比如，如果您使用了 Employee 类并且它实现了正式工的计酬方法，那么在您开始雇用合同工时又该怎么办呢？毕竟合同工的工资计算方法是不同的。在结构化语言中，您可能就需要修改函数以便处理新的雇员类型，因为老代码不可能知道如何处理新代码。而面向对象的解决方案就可以通过多态性来处理类似这样的问题。

还是以前面的例子为例，您定义了一个名为 Employee 的基类，然后为每一种雇员类型定义了一个派生类(如前所见)，每一个派生的雇员类都有其自己对 CalculatePay 方法的实现。这时奇迹发生了。利用多态性，当您使用一个指向对象的已向上转换的指针调用该对象的方法时，程序运行时会自动保证调用正确版本的方法。可以用下面的代码加以说明：

```
using System;

class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }
```

```
    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public virtual double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("Employee.CalculatePay");
        return 42; // bogus value
    }
}

class SalariedEmployee : Employee
{
    public SalariedEmployee(string firstName, string lastName,
                            int age, double payRate)
    : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("SalariedEmployee.CalculatePay");
        return 42; // bogus value
    }
}


class ContractorEmployee : Employee
{
    public ContractorEmployee(string firstName, string lastName,
                              int age, double payRate)
    : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("ContractorEmployee.CalculatePay");
        return 42; // bogus value
    }
}

class HourlyEmployee : Employee
{
    public HourlyEmployee(string firstName, string lastName,
```

```
                        int age, double payRate)
    : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("HourlyEmployee.CalculatePay");
        return 42; // bogus value
    }
}

class PolyApp
{
    protected Employee[] employees;

    protected void LoadEmployees()
    {
        Console.WriteLine("Loading employees...");

        // In a real application, we'd probably get this
        // from a database.
        employees = new Employee[3];

        employees[0] = new SalariedEmployee ("Amy", "Anderson", 28, 100);
        employees[1] = new ContractorEmployee ("John", "Maffei", 35, 110);
        employees[2] = new HourlyEmployee ("Lani", "Ota", 2000, 5);

        Console.WriteLine("\n");
    }


    protected void CalculatePay()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay(40);
        }
    }

    public static void Main()
    {
        PolyApp app = new PolyApp();

        app.LoadEmployees();
```

```
            app.CalculatePay();
        }
    }
```

编译并运行这个程序将产生以下输出结果：

```
c:\>PolyApp

Loading employees...

SalariedEmployee.CalculatePay
ContractorEmployee.CalculatePay
HourlyEmployee.CalculatePay
```

多态性至少提供了两个好处。第一，它让您可以将具有共同基类的对象组成一组并对它们进行一致处理。在上面的例子中，尽管从技术上我们已经拥有三个不同的对象类型——SalariedEmployee、ContractorEmployee 和 HourlyEmployee —— 但我们仍可将它们都作为 Employee 对象对待，这是因为它们都是从 Employee 基类中派生出来的。这样就可以把它们都放入定义为 Employee 对象数组的数组之中。由于多态性的存在，当调用这些对象的方法时，程序运行时就会保证调用正确的派生对象的方法。

第二个好处是曾在本节开始时提到的：老代码可以调用新代码。注意 PolyApp.CalculatePay 方法反复调用了由 Employee 对象组成的成员对象数组。由于这个方法调用对象时实际上隐性调用了已向上转换的 Employee 对象，而在运行时程序的多态性又保证会调用正确的派生类的方法，因此，我们可以向系统添加别的雇员类型，将它们插入到 Employee 对象数组之中即可，而应用程序却可以照常运行，无需更改任何源代码！

1.3 本章小结

本章带领您快速浏览了面向对象编程中的术语和概念。关于这个主题我们还有七章的篇幅加以深入研究，因此，对于本书的重点而言，它可能有点喧宾夺主了。但是，牢固掌握面向对象编程的基础知识之后，您才可以更轻松地学习 C#语言。

在本章中我们接触到了很多新观念。理解面向对象系统的关键就是要弄明白类、对象和接口之间的区别，以及这些概念与有效的解决方案之间的联系。好的面向对象解决方案同样有赖于深刻理解并实现面向对象编程的三大原则：封装、继承和多态性。本章提出的概念为下一章打下了一个基础，下一章将介绍 Microsoft .NET 框架和公共语言运行环境(CLR，Common Language Runtime)。

第 2 章　Microsoft .NET 简介

如果不了解.NET 和 C#语言在其中所起的作用，您就无法充分理解.NET 运行环境所支持的一些 C#语言的关键元素。本章对.NET 的综合回顾将有助于您不仅理解贯穿本书的术语，而且明白 C#语言的某些特定功能的设计原理。

如果您阅读过关于.NET 主题的新闻组或邮件列表的话，就会发现有很多用户都曾被这项技术的术语弄糊涂过。一些不断出现的不明确、有时甚至前后矛盾的名称会把您搞得不知所措。当然，这项技术本身就是全新的这个事实也是出现这些问题的一个原因。因此，本书要讲的首要内容就是解释关于.NET 的一些术语。

2.1　Microsoft .NET 平台

Microsoft .NET 提出的想法是：由.NET 将计算的重点从一个由单独的设备和 Web 站点通过 Internet 简单相连的世界，转变成一个由设备、服务程序和计算机协同工作的世界，以便为用户提供更加丰富广泛的解决方案。Microsoft .NET 方案由以下四个关键部分组成：

- .NET 构建块服务(Building Block Service)，即对某些特定服务程序的访问，如用于文件存储的服务、日历管理或 Passport .NET(一种身份鉴别服务)。
- .NET 设备软件，运行于新型 Internet 设备之上的软件。
- .NET 用户体验，包括如自然界面、信息代理和智能标签这样的功能，这些技术可以自动建立超级链接，这些链接指向与用户创建的文档中的单词或短语相关的信息。
- .NET 基础结构，由.NET 框架(Framework)、Microsoft Visual Studio .NET、.NET 企业服务器(Enterprise Server)和 Microsoft Windows .NET 组成。

大多数开发人员提到.NET 时，他们实际上指的是.NET 的一部分，即.NET 基础结构。您可以认为本书每次提到.NET 时(如果前面没加限定语的话)，指的就是.NET 基础结构。.NET 基础结构包括组成这个新的开发环境的所有技术，在这个开发环境中可以创建和运行强大的、可升级的、分布式的应用程序。在.NET 基础结构中，能让我们开发这样的应用程序的工具就是.NET 框架。

.NET 框架由公共语言运行环境(CLR)和.NET 框架类库组成，类库有时也叫做基础类库(Base Class Library, BCL)。可以把 CLR 看作是一台虚拟机器，所有的.NET 应用程序在这台

机器中起作用。所有的.NET 语言都可以使用.NET 框架类库。如果您熟悉 Microsoft 基础类(Microsoft Foundation Classes, MFC)或 Borland 公司的对象窗口库(Object Windows Library, OWL)的话，就不会对类库感到陌生。.NET 框架类库支持从文件 I/O 和数据库 I/O 到 XML 和 SOAP 的一切。实际上，.NET 框架类库非常庞大，甚至可以单独用整本书来简单介绍一下它所支持的所有类。

这里再多说一句，当本书使用“虚拟机器”这个术语时，并不是表示 Java 中的虚拟机器(Java Virtual Machine, JVM)，它指的是这个术语的传统定义。几十年前，当 Java 还只是一种深色热饮的名字时，IBM 就首先使用了“虚拟机器”这个词。虚拟机器是一种高级操作系统的抽象概念，别的操作系统可以在这个完全封闭的环境下运行。之所以将 CLR 看作是一种虚拟机器，是基于这样一个事实：在 CLR 中运行的代码也是在一个封装的、受控的环境下运行，与机器上别的进程是隔离开来的。

2.2 .NET 框架

现在我们来看一看什么是.NET 框架以及它的功能。首先我们要把.NET 与早些时候的分布式应用程序开发环境相比较，然后列出一个.NET 提供的功能列表，这些功能使得应用程序开发人员可以更快地创建强大的分布式应用程序。

2.2.1 Windows DNA 和.NET

读者对本书在前面对.NET 的描述——“用于创建和运行强大的、可升级的、分布式应用程序的环境”——可能并不陌生。其实.NET 基本上就是早些时候试图达到这些目标的尝试的产物，那个平台叫做 Windows DNA。但是.NET 又远远不止于 Windows DNA 想达到的目标。Windows DNA 是一个解决方案的平台，它关注的是如何使用 Microsoft 服务器产品来解决业务问题。术语“胶水”(glue)有时与 Windows DNA 一起使用，如：“DNA 定义了把强大的、可升级的、分布式系统连接为一体的胶水”。然而，除了仅仅是一个技术规范之外，Windows DNA 没有任何切实的东西。这就是 Windows DNA 和.NET 之间的几个主要区别之一。Microsoft .NET 不仅是一套技术规范，它还包括了几个实实在在的产品，如编译器和类库，甚至还包括整个的最终用户应用程序。

2.2.2 公共语言运行环境

公共语言运行环境(CLR)正是.NET 的核心所在。顾名思义，CLR 就是一个运行期环境，

使用不同语言编写的应用程序都可以在这里运行并且互不干扰——即所谓的“跨语言互用”(cross-language interoperability)。那么 CLR 又是怎样为跨语言互用提供的这样一个理想的环境呢？公共语言规范(CLS)提供了一套规则，任何一种编程语言的编译器只要遵循这套规则，就可以创建出能在 CLR 中运行的.NET 应用程序。任何人要想写一个.NET 兼容的编译器的话，只需遵循这些规则即可。这样，从我们的编译器中产生的应用程序同样可以和其他的.NET 应用程序一起运行并具有同样的互用性。

与 CLR 相关的一个重要概念是受控代码(managed code)。受控代码就是在 CLR 之下运行的代码，因此由 CLR 进行管理。可以这样来理解这个概念：在现今的 Microsoft Windows 环境下可以运行各种各样不同的应用程序，这些应用程序需要遵循的惟一规则就是在 Windows 环境下正常运行，而创建这些应用程序时可能采用的是完全不同的编译器。换句话说，这些应用程序只要遵循在 Windows 下运行这条最基本的规则即可。

在 Windows 环境下，在应用程序如何互相通信、分配内存、甚至请求 Windows 操作系统为自己服务等方面，都只有少数几条全局规则。但在受控代码环境下，却有许许多多的规则用来确保所有的应用程序以一种全局的统一方式运行——不管这些应用程序是用什么语言编写的。这种.NET 应用程序的统一行为就是.NET 的本质而并非其他。幸运的是，这些全局规则主要只影响那些编译器的开发人员。

2.2.3 .NET 框架类库

.NET 框架类库对于提供语言互用性来说非常重要，因为它们允许开发人员对于所有由 CLR 公布的功能使用单一的编程接口。如果您曾经在 Windows 开发环境下使用过几种不同的编程语言的话，您就会深深喜欢上这个功能。实际上，.NET 框架类库在编译器开发领域中正在形成一个革命性的趋势。在.NET 出现之前，大多数编译器开发人员开发的语言都能够自己完成大部分工作。即使是像 C++这样的语言，尽管设计时是作为一种与类库协同工作的功能集，它也至少包含一些自己的功能。但是，在.NET 的世界里，编程语言就只是通向.NET 框架类库的语法接口而已。

下面我们首先来看看 C++中标准的“Hello, World”应用程序，然后与 C#中同样的应用程序进行比较：

```
#include <iostream.h>

int main(int argc, char* argv[])
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

注意应用程序首先包含了一个头文件，这个头文件中包含了对 cout 函数的声明。应用程序的 main 函数——每个 C/C++应用程序的入口——使用 cout 函数来将字符串“Hello, World”写入标准输出设备。这里要着重指出的是：没有.NET 框架类库，您不能用任何.NET 语言来编写这个应用程序。这是因为.NET 语言根本没有最基本的编译器功能，如向控制台输出字符串之类。不过现在我们知道其实 cout 函数是在 C/C++运行的时候实现的，而 C/C++运行的时候本身就是一个库。诸如字符串格式化、文件 I/O 和屏幕 I/O 这样的 C++基本任务，逻辑上至少被认为是基本语言的一部分。而对于 C#或任何.NET 语言而言，语言本身几乎没有能力完成这样最简单的任务——如果没有.NET 框架类库的话。

下面来看看在 C#中是如何实现“Hello, World”这个程序的：

```
using System;

class Hello
{
    public static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

这下您能看出这套通用的类库对您来说意味着什么了吗？它是不是很不错？当然，这取决于您自己的看法。一套通用的类库意味着理论上所有的编程语言都具有相同的功能，因为它们都必须使用同样的类库来完成除了变量声明之外的任何任务。

作者在论坛上曾看到过这样的“牢骚”：“如果所有的语言都具有相同的功能，那为什么还有这么多不同的语言呢？”作者不明白为什么会有这样的抱怨。作为曾在很多多语言开发环境下工作过的程序员，作者可以证明如果程序员不必牢记什么语言对于系统可以做什么事情以及怎么做那将带来多大的好处。总之，作为开发人员，我们的任务就是生产代码，而不必去考虑自己最喜欢的语言有这个优点或是那个优点。

作者经常看到的另一个问题是：“如果所有这些.NET 语言能做同样的事情，我们为什么还需要其他的语言呢？”答案与这样一个事实有关，那就是程序员们都有各自不同的习惯。Microsoft 当然不希望在这么多可用的语言中只挑选出一种，然后强迫数百万的程序员们放弃他们多年来使用别的语言编程的经验。这样，程序员们不仅必须熟悉一种新的 API，还可能被迫去掌握一种完全不同的语法。而现在，开发人员还能够继续使用最适合自己工作的编程语言。总而言之，我们的目的是提高工作效率，而不是改变那些不需要改变的地方。

注意： 尽管理论上.NET 框架类库能够让编译器把所有的 CLR 功能都提供给使用语言的用户，但实际上并非总是如此。在 Microsoft 公司内部.NET 框架类库

小组和其他不同编译器小组之间的一个争议就是：尽管.NET 框架类库小组努力将其所有的功能都展示给不同的语言，但是除了要求达到最低限度的 CLS 标准之外，他们并没有要求各个编译器小组实现每一种功能。当作者就这个问题咨询几个 Microsoft 的开发人员时，他们认为其实每一个编译器小组都已经决定只实现那些他们觉得对用户最有用的功能，而不会实现.NET 框架展示的每一个功能。但幸运的是，C#正好提供了访问几乎所有.NET 框架功能的接口。

2.2.4 Microsoft 中间语言和 JITters

为了帮助语言开发者将他们的语言转向.NET，Microsoft 开发了一种类似于汇编语言叫做 Microsoft 中间语言(Microsoft Intermediate Language, MSIL)的语言。要编译出在.NET 上运行的应用程序，可以将源代码作为编译器的输入，而编译器将产生 MSIL 作为输出。MSIL 本身就是一种完备的语言，您也可以用它来编写应用程序。但是，作为一种汇编语言，您可能永远也不会用它来编写程序，除非情况十分特殊。因为 MSIL 是各个编译器小组所开发的语言，每一个编译器小组都会自行决定在多大程度上支持 MSIL。不过，如果您是一位编译器开发人员并且希望创建一种与别的语言互用的语言的话，您应该严格限制在 CLS 指定的那些功能之内。

当您编译一个 C#应用程序或其他任何用 CLS 兼容的语言编写的应用程序时，应用程序会被编译成 MSIL。当该应用程序首次在 CLR 中执行时，MSIL 就被进一步编译成真正的 CPU 指令(实际上，函数只有在首次调用时才会被编译)。不过，既然本书叫做《C#技术内幕》，让我们仔细看看实际上究竟发生了些什么：

1. 您使用 C#书写了源代码。
2. 然后您使用 C#编译器(csc.exe)将它编译成一个 EXE 文件。
3. C#编译器输出 MSIL 代码和一个清单，组成 EXE 文件的只读部分，这个 EXE 文件带有一个标准的 PE(Win32-Portable executable)头。

   注意这里有个关键地方：当编译器创建输出时，它同时也从.NET 运行环境导入了一个名为_CorExeMain 的函数。
4. 应用程序被执行时，操作系统载入 PE，与任何独立的动态链接库(DLL)一样，如导出_CorExeMain 函数的 mscoree.dll，操作系统把它当作合法的 PE。
5. 操作系统载入器跳转到 PE 内部的入口处，这个入口是由 C#编译器放置的。别的 PE 在 Windows 中也是这样执行的。

   但是，由于操作系统并不能执行 MSIL 代码，因此入口处仅仅是一个跳转，直接跳转到 mscoree.dll 中的_CorExeMain 函数。

6. _CorExeMain 函数开始执行放置在 PE 中的 MSIL 代码。
7. 由于不能直接执行 MSIL 代码——因为它不是一种可执行的机器码，CLR 在处理 MSIL 的时候，就使用 JIT(just-in-time，即时)编译器(或者叫做 JITter)把是 MSIL 编译成真正的 CPU 指令。只有当程序中的方法被调用时才会进行即时编译。编译后的可执行代码在机器上缓存起来，只有当源代码发生改变时才会重新编译。

根据不同的情况，有三种不同的 JITter 可以用来将 MSIL 转换成真正的代码：

- 安装时代码生成(Install-time code generation)

  安装时代码生成将把整个的组合体(assembly)编译成特定CPU的二进制代码，就像 C++编译器一样。一个组合体就是一个发送给编译器的代码包(本章稍后的“部署”部分还将详细讨论组合体)。这个编译过程在安装时就全部完成，因此最终用户也不易注意到这个组合体是被即时编译过的。安装时代码生成的好处就是您能在运行之前只编译一次整个的组合体。由于整个的组合体已被编译，您就不必担心每次代码中首次执行一个方法时会断断续续地出现性能问题。这就像是一个分时度假计划，您一开始就付清了所有的款项。虽然为度假计划付钱时会很痛苦，但好处是您再也不必为食宿操心了。是否使用这个工具及何时使用有赖于您的系统和部署环境。通常，如果您打算为您的系统创建一个安装应用程序的话，您就应该使用这个 JITter，以便用户“第一次使用时”就能得到一个完全优化的系统。

- JIT

  每当程序首次调用一个方法时，就会调用默认的 JITter，调用方式如前所述。这有点像一个“边度假边付钱”的计划。如果没有显式运行 PreJIT 编译器的话，它就是默认的 JIT 编译器。

- EconoJIT

  EconoJIT(经济型 JIT)是另一种运行环境的 JITter，它是专为那些资源有限的系统而设计的，比如，只有少量内存的手持式设备。这种 JITter 和常规的 JITter 最大的不同就在于它包含了所谓的代码扔掷(code pitching)。代码扔掷允许 EconoJIT 在系统内存不够用时丢弃那些已经产生出来的、编译过的代码。这样做的好处是内存可以回收，但缺点是如果再次调用被丢弃的代码时，它必须被重新编译，就好像从来没被调用过一样。

2.2.5 统一的类型系统

任何开发环境的关键性能之一就是其类型系统。一个只具有有限数量类型的系统或者

限制程序员扩展由系统提供的类型的系统，其生命期都不会长久。.NET 运行环境不仅为开发人员提供了一个能在所有兼容 CLS 语言中使用的单一的、统一的类型系统，而且允许语言开发人员通过添加新的类型来扩展这个类型系统，添加的新类型看起来和用起来都与系统内置的类型一样。这意味着开发人员可以用一种统一的方式来使用所有的类型，而不管这些类型是.NET 预定义类型也好，还是用户自己创建的类型。在第 4 章中我们将详细讨论类型系统以及 C#编译器如何支持这个类型系统。

2.2.6 元数据和反射

在上一节“Microsoft 中间语言和 JITter”中我们提到过，兼容 CLS 的编译器将您的源代码作为输入，然后产生 MSIL 代码用于在运行时编译(通过 JITter)和执行。除了将源代码映反射为 MSIL 指令序列之外，兼容 CLS 的编译器还有另外一项同等重要的任务：将元数据嵌入到产生的 EXE 文件中。

元数据(Metadata)是一种描述数据的数据。在这里，元数据就是组成 EXE 文件的程序条目的集合，如声明的类型和实现的方法等。如果这听起来有点熟悉，那就对了。这个元数据与组件对象模型(Component Object Model, COM)组件产生的类型库(typelib)相似。从.NET 编译器输出的元数据不仅比我们熟悉的 COM 类型库更加完整和丰富，而且它总是嵌入在 EXE 文件之中。这样，就永远不会丢失应用程序的元数据或者出现文件匹配错误了。

使用元数据的理由很简单。它允许.NET 运行环境在运行时刻知道要分配怎样的类型以及要调用怎样的方法。这样运行环境就可以设置得更有效，以便应用程序运行。查询元数据的这种方式就叫做反射(reflection)。事实上，.NET 框架类库提供了一整套反射方法，以便任何应用程序——不仅是 CLR——可以查询另一个应用程序的元数据。

Visual Studio.NET 这样的工具使用这些反射方法实现了诸如 IntelliSense 这样的功能。有了 IntelliSense，当您键入一个方法名时，该方法的参数值就会出现在屏幕上的一个列表框内。Visual Studio.NET 将这个功能做了进一步的增强，它可以显示出一个类型的所有成员。在第 15 章中，我们将讨论反射 API。

利用反射的另一个十分有用的.NET 工具是 Microsoft .NET 框架 IL 解析器(IL Disassembler, ILDASM)。这个强大的实用程序可以解析目标应用程序的元数据，然后以一种树状层次来显示有关该应用程序的信息。图 2.1 说明了在 ILDASM 中，C#应用程序“Hello, World”是怎样表示的。

图 2.1 背景中的窗口就是 IL 解析器的主窗口。双击树状视图中的 Main 方法，就可以在前端看到显示 Main 方法细节的窗口。

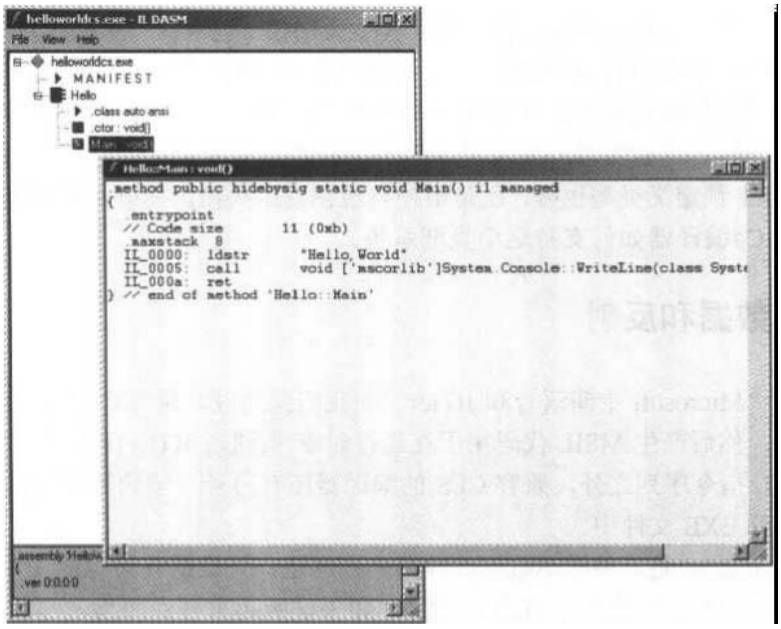


图 2.1 ILDASM 中显示的 C#应用程序“Hello, World”

2.2.7 安全问题

任何分布式应用程序开发环境最重要的方面就是它如何处理安全问题。对于那些长期以来抱怨 Microsoft 没有认真考虑在服务器方面的企业解决方案中提出全新安全措施的人来说，.NET 带来了很多的新概念。实际上，CLR 一载入类，安全措施就开始生效了，因为类载入器本身就是.NET 安全计划的一部分。比如，在.NET 运行环境中载入一个类，与安全相关的因素如访问性规则和一致性要求就被验证了。此外，安全检查还确保了每一段代码都拥有适当的资格，以便访问一定的资源。安全代码保证了角色认定和身份信息。这些安全检查甚至还对进程和机器的界限进行了扩展，以确保敏感数据在分布式计算环境中不会受到危害。

2.2.8 部署

到目前为止，在开发超大型的分布式系统时，部署仍然是最让人讨厌的任务。实际上，任何一位 Windows 开发人员在处理不同的二进制码文件、注册表冲突、COM 组件和诸如开放式数据库连接(Open Database Connectivity, ODBC)及数据访问对象(Data Access Object, DAO)这些产品的支持库安装时，可能都会因为厌烦而重新考虑自己的职业选择。值得庆幸

的是，.NET 设计小组显然在部署这个领域已经花费了不少的时间。

.NET 应用程序部署的关键就是组合体这个概念。组合体其实就是包含单个文件或多个文件实体的数据包。怎样部署应用程序将由您开发的是 Web 服务器应用程序还是传统的 Windows 桌面应用程序来决定。不过，由于装配被作为一个完整的、封装的功能集而引入，部署就简单得好像将必要的组合体复制到目标位置一样。

在.NET 框架出现之前困扰程序员们的许多问题因此迎刃而解。比如，再也不必按照 COM 组件和 Microsoft ActiveX 控件的使用方法来对组件进行注册了，这是因为有了元数据和反射之后，所有的组件都是自描述的了。.NET 运行环境同样保持着文件的记录和与应用程序相关的文件的版本。因此，所有安装的应用程序就自动与它组合体中的文件相关联。如果一个安装程序试图覆盖一个另外的应用程序需要的文件，.NET 运行环境会允许这个安装程序安装它需要的文件，但同时 CLR 不会删除这个文件的先前版本，因为其他应用程序仍然需要它。

2.2.9 与非控代码的互用性

您可能已经猜到，非控代码(unmanaged code)就是不受.NET 运行环境控制的代码。但是这一点要弄清楚：非控代码仍然是在.NET 运行环境中运行。不过，非控代码不具备受控代码的一些优点，比如：垃圾收集、统一的类型系统和元数据。您可能会产生这样的疑惑：为什么会有人想在.NET 环境下运行非控代码呢？当然了，除非万不得已，没有人会希望这样做，只有在无法选择别的方式时，才会考虑使用它。以下就是一些这样的情形，到那时您就会感谢 Microsoft 在.NET 中加入了这个功能。

- 受控代码调用非控 DLL 函数

  比如您的应用程序需要与类似 C 的 DLL 接口，但是编写这个 DLL 的公司在采用.NET 方面落后于您的公司。而这时您仍然需要从.NET 应用程序中调用那个 DLL。在第 16 章中我们将讨论这个例子。

- 受控代码使用 COM 组件

  同样的原因，您可能还是需要从.NET 应用程序中调用类似 C 的 DLL 函数，您可能还需要继续支持 COM 组件。这种情形下，您可以为 COM 组件创建一个.NET 外包装，这样，受控的客户代码就会以为它是在同.NET 类打交道。第 16 章中我们也会讨论这个问题。

- 非控代码使用.NET 服务

  这正好是个相反的问题——您希望从非控代码中访问.NET。使用一个相反的思路可以解决这个问题：让 COM 客户代码误以为它使用的是 COM 服务器，而实际上却是某个.NET 服务程序。在第 16 章中您也会看到这个例子。

2.3 本章小结

Microsoft .NET 代表了计算模型的一个转变，在这个新的计算模型中，所有的设备、服务程序和计算机在一起协同工作为用户提供解决方案。这个转变的中心在于.NET 框架和 CLR 的开发，如图 2.2 所示。.NET 框架包含那些编译成在 CLR 中运行的语言所共享的类库。因为 C#是为 CLR 而设计的，没有 CLR 和.NET 框架类库的话，在 C#中甚至连最简单的任务也不能执行。理解这些技术的特点对于掌握 C#语言和理解本书非常有用。

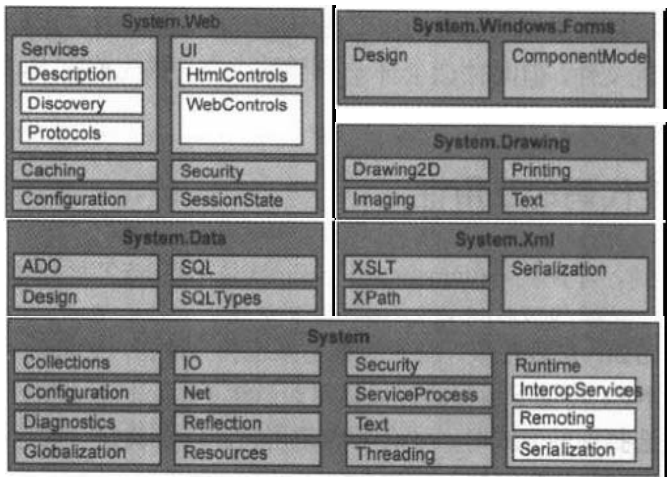


图 2.2 .NET 框架包含的类库便于服务程序、设备和计算机之间的互用

第 3 章　C#应用程序简介

在我们深入到主题——第Ⅱ部分“C#类基础”和第Ⅲ部分“编写代码”——之前，最好安排一个入门介绍性的章节。在这一章中，我们将快速浏览一个简单的 C#应用程序的开发过程。首先，我们将比较各种可以用来编写 C#代码的编辑器的优缺点；选定编辑器后，我们就来编写这个规范的“Hello, World”示例程序，以便了解编写 C#应用程序的基本语法和结构。您将会发现与大多数语言一样，C#的语法也是很公式化的，您可以将它作为一个模板来编写大多数基本的 C#应用程序。最后您将学习如何使用命令行编译器来编译程序以及如何运行这个新的应用程序。

3.1　编写第一个 C#应用程序

下面我们就来一步一步地编写第一个 C#应用程序，然后运行它。

3.1.1　选择编辑器

在编写 C#应用程序之前，您需要选择一个编辑器。以下的章节描述了一些最流行的编辑器，以及为您的 C#开发环境提供了关于选择编辑器的一些相关意见。

3.1.1.1 记事本

在 C#的早期，Microsoft 记事本是开发人员使用.NET 框架 SDK 编写 C#应用程序时最常用的编辑器。在本书中之所以选择了记事本还有一些其他原因，将在稍后具体解释。不过，我们并不推荐使用记事本，原因如下所述。

- C#文件应该被保存为带有扩展名.cs。但是，在记事本的“文件保存”对话框里，如果您不够仔细，忘了将“保存类型”下拉列表框的设置改变为“所有文件”的话，您想命名为“Test.cs”的文件就会被保存为“Test.cs.txt”。
- 记事本不会显示行号。如果编译器报告在某一行出现错误时，没有行号会很麻烦。
- 记事本为每一个 tab 键插入 8 个空格，这意味着任何难度超过“Hello, World”的应用程序的可读性将会很低。
- 记事本在您敲回车键后不会自动缩进。因此，您必须手动定位到希望输入代码的

位置。

当然还存在其他一些原因，但仅仅这些也许已经能让您感觉到记事本的确不是一个开发 C#应用程序的好选择。

3.1.1.2 Visual Studio 6

因为我们在 Microsoft Windows 环境下做开发时用的是 Microsoft Visual C++语言，因此 Microsoft Visual Studio 6 自然就是我们选择的编辑器。Visual Studio 是一个全能的编辑器，它包含了编辑和保存 C#文件所需的所有功能。

程序员们使用的编辑器的最大优点之一就是能够突出显示不同的语法成份。不过，由于 Visual Studio 6 在 C#出现之前就已发行多年，并且是为开发 Visual C++应用程序而设计的，所以在 Visual Studio 6 能够正确地高亮度显示 C#代码之前，您还得做一点小小的改动。第一步是更改 Visual Studio 注册表表项。使用 Regedit.exe 或其他的注册表编辑器在您的注册表中定位到下面的表项：

HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\

Text Editor\Tabs/Language Settings\C/C++\FileExtensions

其值将包含一个类似以下的字符串：

cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2

在该字符串的末尾添加上.cs 扩展名(注意最后一个分号是可选的)。新的注册表值如下：

cpp;cxx;c;h;hxx;hpp;inl;tlh;tli;rc;rc2;cs

这样，在 Visual Studio 6 中打开一个带有.cs 扩展名的文件时，Visual Studio 就会将这个文件看作一个它能支持的文件。

下一步，您需要告诉 Visual Studio 哪些文件是 C#中的关键字。您要创建一个名为 usertype.dat 的文件并将它放到 msdev.exe 所在的那个文件夹中。这是一个 ASCII 文本文件，它包含所有应该高亮度显示的关键字，一行放一个关键字。当 Visual Studio 启动时就会载入这个文件。这样，每次修改过这个文件之后，都必须重新启动 Visual Studio，所做的改动才会生效。在本书配套光盘中已经包含了这个 usertype.dat 文件，它列出了 C#语言的所有关键字。图 3.1 表示采取这些步骤之后 C#代码的结果：

3.1.1.3 Visual Studio .NET

很显然，如果您想在.NET 环境下获得最高的工作效率，您就应该使用 Visual Studio .NET。它不仅为创建 C#应用程序提供了所有的集成工具和向导，而且包含了诸如 IntelliSense 和动态帮助(Dynamic Help)这样的高效功能。有了 IntelliSense，当您键入名称空间或类的名称时，屏幕上就会自动显示出它们的成员，这样，您就不必记住每个类的所有成员了。当您键入方法名和一个括号时，IntelliSense 就会自动显示出该方法的所有参数值

及其类型。Visual Studio 6 也能提供这个功能，但显然它不会支持.NET 类型和类。动态帮助是 Visual Studio 的一项新功能。当您在编辑器中输入代码时，屏幕上会出现一个单独的窗口，该窗口中会显示出与光标所在位置的单词相关的主题。比如，如果您键入关键字“namespace”，则该窗口中会显示出关于名称空间用法的帮助主题的超级链接。

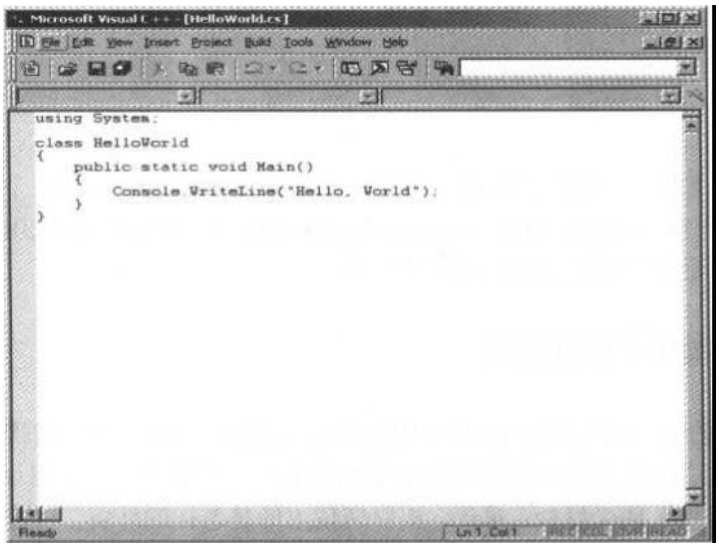


图 3.1　使用这种提供语法高亮度显示的编辑器(如 Visual Studio 6)的一个优点就是能够立即反馈出关键字是否合法

3.1.1.4　第三方编辑器

我们别忘了还有很多流行的第三方编辑器，如 Starbase 公司的 CodeWright 和来自 MicroEdge 的 Visual SlickEdit。本书将不深入讲述这些编辑器的细节，但您完全可以使用任何第三方的编辑器来编写 C#应用程序。

3.1.1.5　本书中使用的编辑器

在作者开始写本书时，由于 Visual Studio .NET 仍在测试阶段，因此使用的是 Visual Studio 6。作者希望使用这个不是为 C#而设计的环境有助于证明：本书会对任何开发 C#应用程序的人有用，不管他选择的是何种开发环境。前面已经说过，您甚至可以使用记事本来编写本书中的示例程序，而您仍可得到希望的结果。

3.1.2　“Hello, World”

假设您已经决定使用何种开发环境，下面我们来看看您的第一个 C#应用程序——规范

的“Hello, World”应用程序。将下面的代码写入一个文件，将它以文件名“HelloWorld.cs”保存：

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, World");
    }
}
```

您无需急于了解以上每行代码的作用。目前我们关心的是写出第一个应用程序、编译它并且执行它。一旦我们顺利完成——这将确认您拥有创建和运行 C#应用程序的正确环境——我们就会深入到这些代码的内部细节之中。

3.1.3 使用命令行编译器

如果您使用的编辑器带有构建 C#应用程序的内置功能，那么这一步骤对于您来说可能显得多余。不过，为了证明 C#独立于编辑器的特点，在本书中将使用 C#命令行编译器(csc.exe)。这样提供了两个好处：第一，不管您使用的是怎样的环境，构建示例程序的步骤总是能够进行下去；第二，学习不同的编译器选项之后，若以后您的编辑器在功能上有所缺陷的话，学到的这些方法将最终有助于您处理这样的情形。

键入“Hello, World”代码之后，打开一个命令窗口，进入 HelloWorld.cs 文件所在的文件夹，键入下列内容：

```
csc helloworld.cs
```

如果一切正常的话，您将会看到类似图 3.2 所显示的结果。随同编译器名称和版本号一同显示的是错误信息或警告信息。如图所示，作者在写本书时使用的编译器是一个测试版本。不过，不管您使用的是哪种版本的编译器，在这个简单示例的结果中您都不应看到任何错误或警告信息。

如果您看到了这样的错误信息“‘csc’ is not recognized as an internal or external command, operable program or batch file”，可能就表示您还没有安装包含 C#编译器的.NET SDK。

如果您调用 C#编译器的方式不正确，比如传递给它一个非法的文件名、或者调用时使用了“/?”参数值，那么编译器就会列出编译您的应用程序时可以设置的每一个可能的选项选项。本书将不停下现在的讨论转而描述编译器可以设置的所有选项，因此只着重于几个与当前讨论相关的选项选项，如有兴趣您可以深入研究余下的部分。

图 3.2　编译 HelloWorld.cs 不应产生任何错误或警告

3.1.4　运行应用程序

既然您已经构建了“Hello, World”应用程序，现在就来运行一下，以便确认您的.NET 运行环境设置是否正确。本书中所有的示例程序都是“控制台应用程序”，表示我们专注于 C#，而不包含任何特定于 Windows 开发环境的因素。因此，您可以从命令行来运行这些示例程序；如果您的编辑器支持命令行应用程序运行的话，您也可以从编辑器来运行这些示例程序。

如果您使用了命令行选项，现在就打开一个命令窗口并在构建应用程序的路径下在命令提示符后键入 **HelloWorld**，输出的结果应该如：

```
d:\>HelloWorld
Hello, World
```

另外，当您试图从编辑器运行这个应用程序时，如果没有看见任何东西，或者突然出现一个命令提示符运行这个应用程序并很快消失的话，可以按如下修改您的代码：

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, World");
        String str = System.Console.ReadLine();
    }
}
```

调用 System.Console.ReadLine 会引起应用程序暂停，直到您敲回车键为止。这样，您就有机会看清楚应用程序是否打印了您所希望的东西。从这里开始，本书中的示例程序都不再包含这行代码。如果您使用编辑器来运行这些示例程序的话，您必须记住在每个应用

程序结束之前插入这行代码。

3.2 代码详解

既然您已经会编写和执行 C#应用程序了，下面我们就来对这些代码进行详细的讨论，并且看看 C#应用程序的基本结构。

3.2.1 “直达式”编程

也许您已经从前面几章和“Hello, World”应用程序中发现，这些示例程序中显示的每个类的方法都是在类定义之内被定义的。这样处理并不仅仅是为了省事，尽管 C++程序员们可能会这样猜想。在 C++中编程时，有两种选择：既可以将类的成员函数的实现直接写入类声明之中，如内联编程(inline programming)，也可以将类声明和它的成员函数定义分别放在不同的文件中。在 C#中，您没有这样的选择。

在 C#中当您定义一个类时，您必须同时在其中定义所有的方法——因为不存在任何头文件。这样做有什么好处呢？它允许类的编写人员可以创建移植性很好的代码，这也是.NET 开发环境的关键概念之一。当您以这种方式编写 C#类时，得到的是一个完全封装好的功能集，这样您就可以轻易地将它嵌入别的开发环境，而不必担心这种语言处理如何引用的文件或它能否引入包含在文件当中的文件。采用这种“直达式”的编程(one-stop programming)思路，打个比方说，您可以将整个类放进一个活动服务器页面(Active Server Pages, ASP)中，而这个类仍然会正常工作，犹如把它编译进 Windows 桌面应用程序中一样！

3.2.2 类和成员

在一个基本的 C#应用程序中，您首先看到的是类或名称空间的名称。如您在第 1 章中学到的一样，您应该为类选择一个能够描述问题域的名称，如 Invoice、PurchaseOrder 或 Customer。类定义开始和结束的地方分别以大括号“{”和“}”标识，大括号之间的一切都被认为是该 C#类的组成成份。注意在上面的程序中我们将类命名为 HelloWorld，而应用程序中的一切都是在该类的上下文范围内来定义的。

类的所有成员都是在类的括号之内被定义的，包括方法、字段、特性、索引器、属性和接口等。在后续章节中我们会详细讨论怎样定义这些不同的 C#元素。

3.2.3 Main 方法

每一个 C#应用程序都必须包含一个名为 Main 的方法，这个方法可以在应用程序的任一个类中定义。具体哪个类包含这个方法是无关紧要的，在一个给定的应用程序中您可以拥有任意多的类，只要其中一个类包含名为 Main 的方法就可以了。另外，这个方法必须被定义为“public”和“static”。关键字“public”是一个访问限定符，用于告知 C#编译器任何人都可以调用这个方法。在第 1 章中我们已经看到，关键字“static”告诉编译器方法 Main 是一个全局方法。对于被调用的这个方法来说，它所在的类不需要被实例化。否则，编译器就不知道怎样以及何时实例化您的类。由于这个方法是静态的，编译器就把它所在的地址保存为入口地址，这样.NET 运行环境就知道从哪里开始执行您的应用程序了。

注意： *本章中的示例程序显示 Main 方法没有返回值，也没有接受任何参数值。不过，您可以将您的 Main 方法定义为有返回值，也可以带有一个参数值数组。这些选项以及如何将参数值传递给应用程序的 Main 方法，将在第 5 章中介绍。*

3.2.4 System.Console.WriteLine 方法

方法 System.Console.WriteLine 将带有行结束符的字符串写入到标准输出设备中。大多数情况下，这表示字符串将输出到一个控制台窗口之中，除非您做了什么改动，或者使用了编辑器将输出重定向到另外的窗口中了。

3.2.5 名称空间和指示符 using

在第 2 章中，您已经了解到.NET 框架类库被组织成一个名称空间的层次结构。当一个希望使用的类或类型嵌套了几层时，这就会造成一些冗长的名称。因此，为了节约键入时间，C#提供了指示符(directive)using。下面来看一个例子，了解一下这个指示符是怎样起作用的。在我们的“Hello, World”应用程序中，有这样一行代码：

```
System.Console.WriteLine("Hello, World");
```

键入这行代码一次当然不是什么大问题，但是想想看：如果在一个大型应用程序中必须使用每个类或类型的全称时会怎么样？指示符 using 使您可以列出一种搜索路径，在编译器不能理解您键入的某个定义时，它就会在列出的名称空间中搜索该定义。我们将指示符 using 用在我们的示例中，程序代码如下：

```
using System;

class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

当编译器解析方法 Console.WriteLine 时，它会认为该方法未定义。然后，它会搜索指示符 using 指定的名称空间，在名称空间 System 中找到这个方法后，就会继续编译代码，而不会产生错误信息了。

注意指示符 using 只应用于名称空间，而不应用于类。在我们的例子中，System 就是一个名称空间，而 Console 是一个类，WriteLine 是一个属于 Console 的静态方法。因此，下面的代码是非法的：

```
using System.Console; //错误：不能对类使用
                      //指示符 using

class HelloWorld
{
    public static void Main()
    {
        WriteLine("Hello, World");
    }
}
```

尽管在指示符 using 中您不能指定类，但利用下面对指示符 using 的变化您可以创建类的别名：

using *alias* = *class*

使用这种形式的指示符 using，您就可以按照以下的形式来编写代码了：

```
using ouput = System.Console;

class HelloWorld
{
    public static void Main()
    {
        output.WriteLine("Hello, World");
    }
}
```

这样，您就可以为那些在.NET 层次结构中嵌套了多层的类取一个有意义的别名，以便

您的代码更易编写和维护。

3.2.6 骨架代码

下面我们来粗略看看哪些部分可以看作是大多数 C#应用程序的骨架代码，这些代码表示了一个简单、无修饰成分的 C#应用程序的基本轮廓。您可能希望将它们写入一个文件并保存下来以便将来作为模板使用。注意尖括号表示您需要补充信息的地方。

```
using <namespace>
namespace <your optional namespace>
class <your class>
{
    public static void Main()
    {
    }
}
```

类的多义性

如果一个类型在多个被引用的名称空间中定义，则编译器会发出错误信息，指示出这种多义性。因此，下面的代码不会编译成功，因为类 C 在两个名称空间中都被定义了，而这两个名称空间都使用了指示符 using 来引用。

```
using A;
using B;

namespace A
{
    class C
    {
        public static void foo()
        {
            System.Console.WriteLine("A.C.foo");
        }
    }
}

namespace B
{
    class C
    {
        public static void foo()
        {
```

```
            System.Console.WriteLine("B.C.foo");
        }
    }
}

class MultiplyDefinedClassesApp
{
    public static void Main()
    {
        C.foo();
    }
}
```

为了避免出现这种错误，您最好给您的类和方法取一个独特的、描述性的名字。

3.3 出错情况

到目前为止，我们已经知道了如何编写、编译和执行C#应用程序，我们有了一个构建C#应用程序的基本大纲。那么，好的C#应用程序什么时候出现“故障”呢？首先，我们来为“故障”下个定义：故障就是任何我们不希望发生的事情。在编程上，故障分为两类：编译期错误和运行期错误。下面来看几个包含这些错误的示例程序，以便了解怎样纠正这些错误。

编译期错误

当编译器，包括C#编译器，不能解读您试图传递的信息时，它就会打印出错误信息，您的应用程序也就无法构建了。将下面的代码输入一个文件，取名为HelloErrors.cs，然后编译它。

```
using Syste;

class HelloErrors
{
    public static void Main()
    {
        xConsole.WriteLine("Hello, World");
        Console.WriteLinex("Hello, World");
    }
}
```

编译器运行将会产生如下输出结果：

```
HelloErrors.cs(1,7):error CS0234:The type or namespace name 'Syste' does
not exist in the class or namespace ''
```

要记住总是存在默认的全局名称空间，显然编译器不能从叫做“Syste”的名称空间中定位任何东西。不过，在此值得说明的是：在编译器遇到了您代码中的语法错误时，您将会看到什么样的提示信息。首先，您将看到当前正在编译的文件的名称，接着是错误所在位置的行号和列号；然后，您将看到由编译器定义的错误码——在这里，错误码是 CS0234。

最后，在错误码之后，您将看到对该错误的简要描述。很多时候，这个描述对于您弄清楚错误就足够了。不过，如果您嫌这个描述不够的话，可以在.NET Framework SDK Documentation 中搜索错误码以便得到对该错误更详细的描述，这个文档是与.NET Framework SDK 一同安装的。与错误码 CS0234 相关的联机帮助如图 3.3 所示：

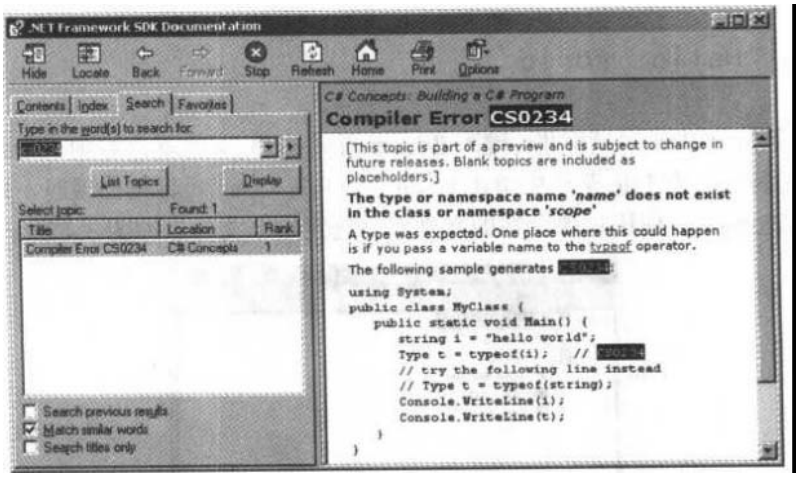


图 3.3　在联机帮助文档中您可以使用由 C#编译器提供的错误码来查找对该错误的详细描述

尽管我们引入了三个错误——名称空间 System 拼错、类 Console 拼错、以及对方法 WriteLine 的调用拼错——但是编译器只报告了一个错误。这是因为编译器遇到某些错误时，就会中止编译过程并且打印出产生错误的地方。在上面这个例子中，编译器一旦不能解决指示符 using 的问题时，它就会停止编译过程，因为已发生的这个错误有可能导致很多与它相关的错误发生。在指示符 using 中将名称空间 System 拼写正确之后，编译器就会报告剩余两个错误所在位置的行号和列号。

3.4　ILDASM 研究

在第 2 章中您已经看到，当您使用.NET 编译器创建一个 EXE 或 DLL 文件时，这个文

件不是通常意义上的可执行文件。相反，它是由一个清单和 MSIL(Microsoft 中间语言)代码组成的。清单上列出了文件中所包含的类型和类，而 MSIL 代码则稍后通过您的安装程序或者.NET 运行环境使用 JIT 编译器(JITter)来进行编译和执行。

这样做的一大好处就是产生的 MSIL 代码看起来就像汇编语言一样，而且可以作为一个很好的教学工具来说明编译器是怎样处理我们的代码的。因此，在本书中常常会停下来使用 C#编译器的 MSIL 输出结果来说明有些语句到底是如何工作，或者为什么您应该以某种特定的方式来使用 C#语言的某个特定功能。为了查看.NET 编译器产生的 MSIL 输出结果，Microsoft 提供了一个名为 Microsoft .NET Framework IL Disassembler (ILDASM)的解析器，以便您能够打开.NET 可执行文件(EXE 或 DLL)并细读它的名称空间、类、类型以及代码等。在下一节里我们将开始熟悉 ILDASM。

MSIL 中的“Hello, World”

在 Start 菜单的 Run 对话框里，键入 **ildasm** 并单击 OK，您就会看到一个带有菜单选项的普通的应用程序。在 File 菜单中单击 Open，当 File Open 对话框出现时，定位到您先前创建的 HelloWorld.exe 应用程序所在的文件夹并选定这个文件。如图 3.4 所示。

图 3.4 IL DASM 允许您深入探究组成.NET 应用程序的清单和 IL 代码

注意 IL DASM 使用树状视图来列出受控二进制代码的内容。图 3.5 显示了 IL DASM 树状视图中使用的各种各样的图标，这些图标用来描述.NET 应用程序的不同组成部分。结合图 3.5 中的图标和 IL DASM 中的“Hello, World”程序可以看出，HelloWorld.exe 由一个清单、一个类(HelloWorld)、两个方法(类构造函数和静态方法 Main)以及一些类信息组成。

名称空间		(蓝色盾牌)
类		(带三个接头的蓝色矩形)
接口		(带三个接头、标有“I”的蓝色矩形)
数值类		(带三个接头的棕色矩形)
枚举		(带三个接头、标有“E”的棕色矩形)
方法		(紫色矩形)
静态方法		(标有“S”的紫色矩形)
字段		(青色菱形)
静态字段		(标有“S”的青色菱形)
事件		(绿色向下三角形)
属性		(红色向上三角形)
清单或类信息		(红色向右三角形)

图 3.5　在 IL DASM 中，不同的图标用于标示.NET 应用程序不同的组成部分

“Hello, World”中最有意思的部分位于方法 Main 当中。在 IL DASM 树状视图中双击方法 Main，就会出现一个窗口显示出方法 Main 的 MSIL 代码，如图 3.6 所示：

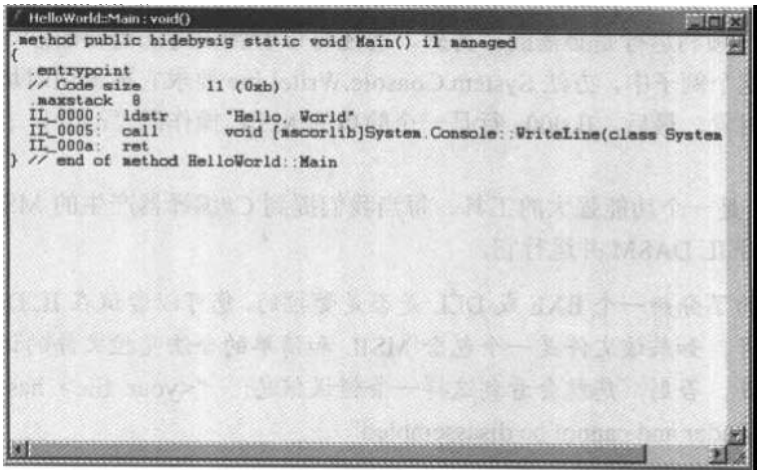


图 3.6　要查看某个方法产生的 MSIL 代码，在 IL DASM 中打开二进制文件并双击该方法

即便是在 MSIL 中，“Hello, World”也没有多少新鲜的内容，但是您可以对生成的 MSIL 形成一个大概的了解，从而举一反三，应用到别的.NET 应用程序中。下面我们就来逐行研究这个方法。

```
.method public hidebysig static void Main() il managed
{
    .entrypoint
    // Code size    11 (0xb)
    .maxstack 8
    IL_0000:        ldstr     "Hello, World"
    IL_0005:    call void [mscorlib]System.Console::WriteLine
                        (class System.String)
} // end of method HelloWorld::Main
```

上面第一行使用了 MSIL 关键字“.method”定义了方法 Main。同时我们还看到该方法被定义为“public”和“static”——方法 Main 的默认限定符。另外，这个方法还被定义为“managed”，这是个很重要的区别，因为在 C#中还可以编写“非控”或“不安全”代码。第 17 章将讨论非控 C#代码。

下一行使用了 MSIL 关键字“.entrypoint”来表明将这个特定的方法作为该应用程序的入口。当.NET 运行环境执行这个应用程序时，将在此把控制权传递给该程序。

接下来有意思的东西就是位于 IL_0000 和 IL_0005 上的真正的 MSIL 代码了。第一行代码使用了操作码“ldstr”(Load String)来将一个固定编码的文字“Hello, World”装载到堆栈上，下一行代码调用了方法 System.Console.WriteLine。注意 MSIL 使用了定义该方法的集合的名称为该方法名加上了前缀。了解了 MSIL 中这个级别上的细节之后，意味着您可以更轻松地编写类似“Dependency Walkers”这样的工具，用它来审查应用程序以便决定哪些文件对于程序顺利运行是必需的。另外，您还可以看到该方法需要的参数值的个数和它们的类型。在这个例子中，方法 System.Console.WriteLine 要求它被调用时堆栈上存在一个 System.String 对象。最后，IL000a 行是一个简单的 MSIL 操作码“ret”，表示从该方法返回。

IL DASM 是一个功能强大的工具。每当我们提到 C#编译器产生的 MSIL 代码时，您就可以立即打开 IL DASM 并运行它。

注意： 为了分辨一个 EXE 或 DLL 是否是受控的，您可以尝试在 IL DASM 中将它打开。如果该文件是一个包含 MSIL 和清单的合法受控文件的话，它就会被打开。否则，您就会看到这样一条错误信息：“<your file> has no valid CLR header and cannot be disassembled”。

3.5 C#编程原则

下面将以介绍编写 C#应用程序的一些原则来结束本章。

3.5.1 何时定义您自己的名称空间

在“Hello, World”应用程序中，我们使用了 System 名称空间中定义的方法 Console.WriteLine。实际上，所有的.NET 类型和类都是在名称空间中定义的。虽然在这个应用程序中我们并没有创建名称空间，但也不妨来看看这个问题。

使用名称空间是一个很好的途径，它将您的类型和类分门别类，以避免发生名称冲突。Microsoft 将所有的.NET 类和类型定义都放置在特定的名称空间，因为它不希望自己使用的名称与其他使用编译器的开发人员所用的名称发生冲突。不过，您是否应该使用名称空间又导致了另外一个问题：您创建的那些类型和类是否会在不受您控制的环境中使用？换句话说，如果您编写的类只是在您自己的小组成员间使用，您当然可以轻松地创建命名规则以保证不会发生名称冲突。但是，如果您编写的类会被第三方开发人员使用，而您对命名又毫无控制的话，这时您就应该确定使用名称空间。另外，Microsoft 建议使用您公司的名称作为顶层名称空间，而作者还建议在有人会看到您代码的任何时候都使用名称空间。

3.5.2 命名原则

据统计，与应用程序开发相关的最大开销总是维护问题。在我们继续之前，先简单地谈一谈有关命名的规范。选择一种稳定的、易于理解的命名规范将帮助您将代码写得更容易阅读，也因此更容易维护。

大多数人都知道，命名规范是一个比较棘手的问题。当 Visual C++和 MFC 首次出现时，这个问题就很明显了。作者还记得当自己在 Peachtree Software 公司做一个开发小组的领导时就面临过这样的问题。当时我们的任务是为公司编写第一个在 MFC 中运行的会计软件，项目刚开始时每个人都雄心勃勃，发誓要解决所有的难题，但这与后来项目进行一段时间以后人人都只希望交付打补丁的软件形成了鲜明的对比。当开发人员在一起讨论时，从他们眼睛里流露出来的神情和他们胳膊下夹着的文件就可以感受到这样一种气氛：这些人准备要好好大干一场。面对这样的场面，该怎么办？当然了，肯定是要坚定立场。因为大多数 MFC 开发工作都与 Microsoft 的代码有关，我们就应当使用 Microsoft 在编写 MFC 时使用的命名规范。总之，如果源代码中存在两套命名系统(一套 MFC 使用的，一套我们自己

使用的)的话，肯定就会影响工作效率。当然，作者自己喜欢用匈牙利(Hungarian)标记的事实并没有带来什么麻烦。

不过，斗转星移，时代不同了。现在在 C#这种新的语言中，我们又遇到了新的挑战。在这个开发环境下，我们看不到上述的 Microsoft 代码。虽然如此，经过与 Microsoft C#设计小组的多次讨论之后，作者发现有一个标准正在形成。它的结局可能会与现在所说的稍微有所不同，但至少能给您一点启示。

3.5.3 命名规范标准

在解释如何为应用程序的不同元素命名之前，我们先来看看目前使用的几种不同的标准。

3.5.3.1 匈牙利标记

匈牙利标记是大多数 C 和 C++程序员(包括在 Microsoft 工作的程序员)使用的命名系统。它是由 Microsoft 的杰出工程师 Charles Simonyi 创建的一套完整的命名系统。早在 1980 年，Microsoft 就采用了这套著名的——或者不著名的，取决于您的观点——命名系统，它是以 Simonyi 的博士论文“Meta-Programming: A Software Production Method”为基础而建立的。

匈牙利标记规定在每个变量前加上表示其类型的前缀。不过，并不是每一种类型都被指定一个标准前缀。另外，随着别的语言的加入和新类型的创建，也就必须再创建新的前缀。这就是当您走进一家采用匈牙利标记的商店时，可能会发现您所看到的某些前缀您并不熟悉的原因(顺便说一句，术语“匈牙利标记”取自这样一个事实：那就是前缀使得变量看起来不符合英语的拼写规则；另外，Simonyi 先生正是匈牙利人)。

也许鼓励使用匈牙利标记的最重要的出版物就是 Microsoft 出版社出版、Charles Petzold 所著的《Programming Windows》，书中的所有示例程序都使用了匈牙利标记，而几乎所有的 Windows 和 OS/2 开发人员都阅读过此书。另外，Microsoft 在它自己的开发中也采用了这种标记方法。当 MFC 发行时，一些针对 C++开发的新前缀也随着源代码一同发行，这样就保证了继续使用匈牙利标记。

那为什么又不继续使用匈牙利标记了呢？因为匈牙利标记在这样的情形下会很有用：那就是知道要使用的变量的类型或作用域会有益处的时候。但是，在下一章您了解更多细节之后就会发现，C#中的所有类型都是对象，都是基于.NET 的 System.Object 类之上的。因此，所有的变量都具有一套基本功能和行为特征。出于这个原因，在.NET 环境下就减少了对匈牙利标记的需要。

注意： 对于那些喜欢寻根究底、或者深受失眠困扰的人来说，提出匈牙利标记的原

始论文可以在 http://msdn.microsoft.com/library/techart/hunganotat.htm 中找到。

3.5.3.2 Pascal 大小写规则和 Camel 大小写规则

尽管已经不可能使 C#设计小组固定使用一种“硬”标准，但从他们编写的代码中可以很明显地发现：其实他们遵循的是 Microsoft 的同事 Rob Caron 提出的一套标记方法，这套方法在为变量命名时混合使用了 Pascal 大小写规则(即各个单词的首字母都大写)和 camel 大小写规则(即除了第一个单词各个单词的首字母都大写)。在 Rob Caron 的论文“Coding Techniques and Programming Practices”(http://msdn.microsoft.com/library/ techart/cfr.htm)中，他建议为方法名使用第一个字符大写的 Pascal 大小写规则，为变量名使用 camel 大小写规则。因此，在本书的示例程序中也采用了这套命名规范。不过，由于 C#包含的元素不止方法和变量，因此下面将列出作者所见过的在 Microsoft 内部使用的不同元素及其命名规范，也同样使用这套命名规范。

注意： 关于本主题的更多信息，您可以参考.NET Framework SDK 文档中包含的 .NET Framework Guidelines，它位于.NET Framework Developer Specifications\ .NET Framework Design Guidelines\Naming Guidelines。

3.5.3.3 名称空间

使用您的公司或产品的名称，并采用首字母大写的大小写混合规则，比如：Microsoft。如果您所在的公司出售软件组件的话，最好创建一个与公司同名的顶层名称空间，然后为每一种产品创建一个嵌套的带嵌入式类型的名称空间，以免与别的产品发生名称冲突。在.NET Framework SDK 中可以找到这样一个例子：Microsoft.Win32。这种策略可能会造成更冗长的名称，但记住使用您代码的开发人员只需使用指示符 using 就可以减少击键次数。如果您的公司名叫 Trey Research，出售两种产品——一个电子表格(grid)和一个数据库(database)，那么就可以将名称空间命名为 TreyResearch.Grid 和 TreyResearch.Database。

3.5.3.4 类

因为对象被看成是有生命、会呼吸的功能实体，所以最好采用能描述出类的问题域的名词来为类命名。在类很通用(也就是说，问题域不那么明确)的情况下——比如，一个代表 SQL 字符串的类型——这时就可以采用 Pascal 大小写规则。

3.5.3.5 方法

对于所有的方法都采用 Pascal 大小写规则。方法就是为解决问题而生的，因此，就让方法的名称来描述它们要做什么吧。比如：PrintInvoice(打印发票)和 OpenDatabase(打开数据库)。

如果方法将用在布尔表达式中，就为方法加上一个表示该方法将做什么的动词作为前缀。比如，如果您有一个方法根据工作站是否锁定来返回一个布尔值，那么就可以将该方法命名为 IsWorkStationLocked，这样，如果在一个 if 语句中使用这个方法，语句的意思就会明确得多，如下所示：

```
if (IsWorkStationLocked) …
```

3.5.3.6 方法的参数值

对于所有的参数值使用 Pascal 大小写规则。最好为所有的参数值取上有意义的名字，以便在使用 IntelliSense 时，用户可以立即知道每个参数值的用途。

3.5.3.7 接口

对于所有的接口使用 Pascal 大小写规则。在接口的名称前通常加上大写的“I”作为前缀，如 IComparable (这是作者所知的在 C#中惟一的一处匈牙利标记)。

很多开发人员在为接口命名时采用与类命名相同的规则。但是，这两者之间存在一个基本的区别。类代表的是一个封装了数据及对这些数据进行处理的函数的实体，而接口则代表了行为。在实现某种接口的时候，实际上表明的是类可以展示那种行为。因此，为接口命名时通常采用形容词。比如，一个声明了串形化数据的方法的接口可以叫做 ISerializable(可串形化的)，这样就明确表示了类实现所具有的功能。

3.5.3.8 类的成员

这部分内容对于 C#开发人员来说，可能是最棘手的问题。那些来自 C++和 MFC 的开发人员已经习惯于在成员名称前加上前缀“m_”。不过，作者仍然推荐使用首字母不大写的 camel 大小写规则。这样，如果您有一个方法，它有一个参数值叫做“Foo”，您就可以将创建的内部成员取名为“foo”，以便将这个参数值与它的内部表示区分开来。

为成员名加上类的名称作为前缀是多余的。比如，有一个类叫做“Author”，创建它的成员时，给成员取名采用“Name”而不要用“AuthorName”，以免将来编写这个成员时要写作“Author.AuthorName”。

3.6 本章小结

在探索 C#语言的过程中，重要的第一步就是在 C#中编写、编译和执行一个程序。尽管通常说来，您使用哪种工具来编辑 C#源文件并不重要，但使用功能更强大一些的编辑器和专为 C#程序开发而设计的开发环境总是有益的。了解 C#编译器的选项选项之后，您就

可以控制编译器如何产生 MSIL 代码了。使用 ILDASM 这样的工具，您可以深入研究 MSIL 代码，这个工具包含在 Microsoft .NET Framework SDK 中。

C#程序的结构提供了很多功能用于使程序更安全、更易编写以及更少产生错误。这些功能包括了名称空间和指示符 using 等。最后，包含了特定大小写规范的命名规范可以使程序增强可读性并更容易维护。

第II部分　C#类基础

第4章　类型系统

Microsoft .NET 框架的核心是一个公共的类型系统，叫做.NET“公共类型系统”(Common Type System, CTS)。除了定义所有的类型之外，CTS 还规定了公共语言运行环境(CLR)在有关应用程序声明和类型使用方面应该遵循的规则。在本章，我们将来看一看这个新的类型系统，学习 C#开发人员可以使用的类型，并理解在 C#应用程序中不同类型的区别。首先我们将讨论在.NET 中每个编程元素都是对象这样一个概念，然后看一看.NET 是如何将类型分为两类的(数值类型和引用类型)，以及发现装箱会使完全面向对象的类型系统更有效地运行，最后介绍在 C#中类型转换是如何进行的并了解一下名称空间。

4.1　一切都是对象

大多数面向对象编程语言都有两种截然不同的类型：一种是语言本身所固有的类型(基本类型)，一种是可以由语言的使用者来创建的类型(类)。您可能已经猜到，基本类型就是通常的简单类型，如字符、字符串和数字等；而类则是倾向于更精巧的类型。

使用两套完全不同的类型系统会导致很多问题产生，其中一个就是兼容性的问题。比如，如果您希望在一个传统系统中使用这两套类型来建立一个int类型的集合，您就需要创建一个专为存储int类型的值的类。如果您又需要一个存储double类型集合的类，您就必须再为double类型做同样的事。这样做的原因就在于这些基本类型之间通常没有多少共同点，它们不是真正意义上的对象，因此不能从一个公共的基类中派生出来。它们更像是必须按照它们自己的方式来单独处理的奇妙类型。这样的传统系统中另一个类似的问题就是：当您希望指定一个方法使用该语言支持的“任何”类型作为参数值时，由于这些基本类型互不兼容，您就不能指定这样的参数值，除非为每一个基本类型都编写一个包装类。

值得庆幸的是，在.NET和C#的世界中这些都不再成为问题，因为CTS中的一切都是对象。实际上，不仅仅一切都是对象，更重要的是，所有的对象都是从一个基类中隐性派生出来的，这个基类被定义为CTS的一部分，叫做“System.Object”，有关这方面的内容将

在“装箱和开箱”这一节中详细介绍。

4.2 数值类型和引用类型

创建一种所有一切都是对象的语言并不是什么新鲜的概念。其他语言已经做过这样的尝试，其中最著名的是 SmallTalk。将所有的一切都当作对象的最大缺点就是降低了性能。比如，如果您试图在 SmallTalk 中把两个 double 类型的数值相加，那么实际上就在堆中分配了一个对象。毋庸多言，如果您仅仅为两个数求和而分配一个对象，这种方式效率极低。

CTS 的设计者们面临着这样一个任务：创建一个所有的一切都是对象的类型系统，但在应用这个类型时，该类型系统要以高效的方式工作。他们提出的解决方案就是将 CTS 类型分为两类：数值类型和引用类型。您很快就会看到，这些术语是如何反映变量在内部的分配和工作方式的。

4.2.1 数值类型

当您拥有一个数值类型的变量时，实际上就拥有了一个包含了实际数据的变量。因此，数值类型的首要规则就是它们不能为空(null)。下面的例子中我们将通过创建一个 C#中 CTS 类型 System.Int32 的变量来分配内存，声明实际上就是在堆栈上分配了一个 32 位的空间。

```
int i = 32;
```

此外，给“i”赋值的行为就导致该 32 位的值被移动到这个已分配的空间中。

在 C#中定义了多种数值类型，包括枚举、结构和基本类型等。不管何时您声明了一个以上类型的变量，都会在堆栈上分配与该类型相关的字节空间，并且直接与这些已分配的位打交道。此外，当您传递一个数值类型的变量时，您传递的是该变量的值，而不是对包含其对象的引用。

4.2.2 引用类型

引用类型与 C++中的引用类似，即它们都是类型安全的指针。类型安全的引用所引用的不仅仅只是一个准确无误的地址，而且它(非空时)总能保证指向指定类型的对象，并且该对象已经被分配在堆上。这里值得注意的是引用也可以为空这个事实。

在下面的例子中，分配了一个引用类型(string)。事实上是在堆上分配了一个值，并且返回了一个指向该值的引用。

```
string s = "Hello, World";
```

与数值类型一样，在 C#中也定义了多种引用类型，如类、数组、代表和接口等。不管何时您声明了一个以上类型的变量，都会在堆上分配与该类型相关的字节空间，并且直接与该对象的引用而不是那些已分配的位“打交道”。

4.3　装箱与开箱

那么，这些不同种类的类型是如何提高系统效率的呢？这就是通过神奇的“装箱”(boxing)来实现的。最简单的理解是：装箱就是将数值类型转换为引用类型。相对应的情况就是引用类型通过“开箱”(unboxing)转换回数值类型。

这项技术之所以如此“伟大”，是因为一个对象在它需要是一个对象的时候，它就仅仅是一个对象。比如，如果您声明了一个数值类型，如 System.Int32，这个变量的内存是从堆栈分配的。您可以将这个变量传递给任何方法，只要这个方法被定义为使用 System.Object 类型的变量即可；您还可以访问这个类型的任何成员。因此，对于您来说，它看起来和感觉起来就像一个对象，但事实上它只是堆栈中的 4 个字节而已。只有当您试图以一种与 System.Object 基类接口相匹配的方式使用这个变量时，系统才会将这个变量自动装箱，使其成为一个引用类型并可以像使用任何对象一样使用它。C#正是通过装箱来使所有的一切看起来都是对象，从而避免了当所有一切都是真正的对象时对系统要求过高。让我们通过一些例子来开始研究。

```
int foo = 42;        // Value type.
object bar = foo;    // foo is boxed to bar.
```

在这段代码的第一行，我们创建了一个 int 类型的变量 foo。我们知道，int 类型是一种数值类型(因为它是一种基本类型)。在第二行，编译器看到变量 foo 被复制到一个叫做 bar 的引用类型变量。这时，编译器就生成将这个值装箱所需的 MSIL 代码。

现在，要将 bar 转换为数值类型，就可以执行一个显式的转换：

```
int foo = 42;        // Value type.
object bar =foo;     // foo is boxed to bar.
int foo2 =(int)bar;  // Unboxed back to int.
```

请注意，装箱(重复一遍，装箱就是将数值类型转换为引用类型)的时候不需要进行显式转换。但是，开箱(从引用类型转换为数值类型)时却必须进行显式转换。这是因为在开箱的时候，一个对象可以被转换为任何类型，所以必须进行显式转换以便编译器确定指定变量类型的转换是否合法。由于类型转换有着严格的规则，并且这些规则是由 CTS 支配的，所以我们将在本章稍后的“类型之间的转换”中更详细地讨论这个问题。

4.4 所有类型的根：System.Object

我们已经说过，所有的类型最终都是从 System.Object 类型中派生出来的，因此保证了系统中的每一种类型都至少有一套公共的功能。表 4.1 描述了所有类型都能自由使用的 4 个公共方法。

表 4.1 System.Object 类型的公共方法

方法名	描 述
bool Equals()	这个方法在运行的时候比较两个对象引用以确定它们是否是完全相同的对象。如果两个变量指的是同一个对象，则返回值为 true。对于数值类型(这将在下一节中讨论)，如果两个变量类型相同并且值也相等，这个方法也返回 true。
int GetHashCode()	取得一个指定对象的散列代码。当类的实现程序出于性能的考虑要将对象的散列代码放入散列代码表中时，就会用到散列函数。
Type GetType()	在反射方法(这将在第 16 章中讨论)中用来获得给定对象的类型信息。
string ToString	默认情况下，这个方法是用来获取对象名字的。它可以由派生类覆盖，以便返回代表对象的对用户更友好的字符串。

表 4.2 描述了 System.Object 的受保护方法。

表 4.2 System.Object 类型的受保护方法

方法名	描 述
void Finalize()	这个方法由运行环境来调用，以便在收集垃圾之前进行清除。注意这个方法也许会被调用，也许不会。因此，不要把必须要执行的代码放在这个方法中。这条规则涉及到所谓的确定性结束，将在第 5 章中详细说明。
Object MemberwiseClone	这个成员表示对对象的“浅复制”(shallow copy)。浅复制的意思是复制一个包含引用的对象到另一对象时，不会复制其中被引用对象。如果您需要让您的类支持“深复制”(deep copy)，也就是在复制时包含被引用对象的话，您就必须实现 ICloneable 接口，并且自己手动进行复制。

4.5 类型和别名

既然由CTS负责定义可在.NET各种语言中使用的类型，大多数语言就选择了实现这些类型的别名。比如，一个4字节的整数值在CTS类型中是用System.Int32来表示的，C#就为这个类型定义了一个叫做int的别名，这是因为在一种技术之上再使用另一种技术并没有什么好处。表4.3列出了不同的CTS类型和它们的C#别名。

表4.3 CTS类型和别名

CTS类型名称	C#别名	描 述
System.Object	Object	所有CTS类型的基类
System.String	String	字符串
System.SByte	Sbyte	有符号8位字节
System.Byte	Byte	无符号8位字节
System.Int16	short	有符号16位数值
System.UInt16	ushort	无符号16位数值
System.Int32	int	有符号32位数值
System.UInt32	uint	无符号32位数值
System.Int64	long	有符号64位数值
System.UInt64	ulong	无符号64位数值
System.Char	char	16位Unicode字符
System.Single	float	IEEE 32位浮点数值
System.Double	double	IEEE 64位浮点数值
System.Boolean	bool	布尔数值(true/false)
System.Decimal	decimal	128位数据类型，可准确到28或29位数字，主要用于有高精度要求的财务应用程序

4.6 类型之间的转换

下面我们来看一看有关类型的最重要的一个方面：转换。假设一个基类叫做 Employee，一个派生类叫做 ContractEmployee，由于派生类总是能够向上转换为其基类，所以下面这段代码可以正常工作。

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample1
{
    public static void Main ()
    {
        Employee e = new ContractEmployee();
    }
}
```

但是，下面这段代码却是非法的，因为编译器不能提供隐式的向下转换：

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample2
{

    public static void Main ()
    {
        ContractEmployee ce = new Employee(); // Won't compile.
    }
}
```

之所以会有不同结果的原因请看第 1 章以及可替代性概念。记住可替代性原则是指派生类可以用来替代其基类。因此，ContractEmployee 类型的对象就总是可以用来替代或当作一个 Employee 对象的。这就是为什么第一个例子可以通过编译的原因。

但是，您不能将一个Employee类型的对象向下转换为一个ContractEmployee类型的对象，因为不能保证该对象能支持ContractEmployee类所定义的接口。因此，在向下转换的时候，就要用到如下的显式转换：

```
class Employee { }
```

```
class ContractEmployee : Employee { }

class CastExample3
{
    public static void Main ()
    {
        //Downcast will fail.
        ContractEmployee ce = (ContractEmployee)new Employee();
    }
}
```

但是，如果我们“撒谎”并通过将基类显式转换为一个派生类以“欺骗”CTS，又会怎样呢？

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample4
{
    public static void Main ()
    {
        Employee e = new Employee();
        ContractEmployee c = (ContractEmployee)e;
    }
}
```

这个程序被编译通过了，但运行这个程序的时候会产生一个运行异常。这里有两点值得注意：首先，这个结果并不是一个编译错误，因为对象 e 可能真的已经是一个向上转换了的 ContractEmployee 对象，因此，直到运行的时候才能知道经过向上转换的对象的真实情况；其次，CLR 在运行的时候才确定对象的类型。当 CLR 辨认出一个非法的类型转换时，它就抛出一个 System.InvalidCastException 异常。

还有另外一种对象转换的途径，就是使用关键字“as”。使用这个关键字代替类型转换的好处是：如果转换是非法的，您不用担心会产生异常。取而代之的是产生一个值为 null 的转换结果。例如：

```
using System;

class Employee { }

class ContractEmployee : Employee { }

class CastExample5
```

```
{
    public static void Main ()
    {
        Employee e = new Employee();
        Console.WriteLine("e = {0}",
                    e == null ? "null" : e.ToString());

        ContractEmployee c  = e as ContractEmployee;
        Console.WriteLine("c = {0}",
                        c == null ? "null" : c.ToString());
    }
}
```

如果您运行这段代码，就会看到如下结果：

```
c:>CastExample5
e =Employee
c =null
```

注意，将对象与 null 进行比较意味着您不用冒险使用一个空对象。实际上，如果这个例子试图在对象 c 上调用一个 System.Object 的方法的话，CTS 就会抛出一个 System.NullReferenceException 异常。

4.7 名称空间

名称空间用于在 C#应用程序中定义作用域。通过名称空间的声明，开发人员就可以为 C#应用程序提供一个层次结构，这个层次结构以语义相关的类型组和其他(嵌套的)名称空间为基础。多个源代码文件可以使用同一个名称空间。推广开来说，如果您将几个类都放进一个给定的名称空间，就可以在这些类各自所属的源代码文件中定义它们。使用您的类的程序员通过使用关键字“using”就可以访问这个名称空间内所有的类。

注意： 最好尽可能地使用公司的名称来作为根名称空间的名称，这样有助于确保名称空间的惟一性。更多的命名规则请参见第 3 章。

关键字 using

有时，您可能希望使用给定类型的全称，如：namespace.type。不过，这种方式非常乏味并且有时是不必要的。在下面的例子中，我们使用一个存在于 System 名称空间中的 Console 对象。

```
class Using1
{
    public static void Main()
    {
        System.Console.WriteLine("test");
    }
}
```

但是，如果我们不知道 Console 对象只存在于 System 名称空间会怎样呢？关键字“using”允许我们指定一个名称空间的搜索顺序，这样，当编译器遇到一个非全称的类型时，它就可以查看列出的名称空间来搜索这个类型。在下面的例子中，编译器可以将 Console 对象定位在 System 名称空间中，无需每次都由程序员指定。

```
using System;

class Using2
{
    public static void Main()
    {
        Console.WriteLine("test");
    }
}
```

在您开始构建真正的应用程序，而这个程序中包含了对各种各样的 System 对象进行了数百次调用的时候，您就会很快体会到无需预先为每一个对象引用指定名称空间的好处。

您不能使用关键字“using”来指定类的名称。因此，下面的语句将是非法的：

```
using System.Console; // Invalid.

class Using3
{
    public static void Main()
    {
        WriteLine("test");
    }
}
```

不过，您还是可以变相使用关键字“using”来创建一个别名：

```
using console = System.Console;

class Using4
{
    public static void Main()
    {
```

```
            console.WriteLine("test");
        }
    }
```

当存在嵌套的名称空间与长的类名以至于编写代码很冗长乏味时，这种方法特别有用。

4.8 CTS 的好处

任何语言环境或运行环境的关键特征之一就是它对类型的支持。只能提供有限的类型或者限制程序员扩展语言内置类型的那些编程语言的生命期都不会长久。而拥有一种统一类型系统的编程语言还有许多其他的好处。

4.8.1 语言互用性

CTS 在允许语言互用的过程中扮演了一个集成的角色，因为它定义了一套.NET 编译器必须支持的类型，以便与其他语言进行互用。CTS 本身是在公共语言规范 CLS 中定义的。CLS 为每个.NET 编译器定义了一套单独的规则，以保证每个编译器输出的代码与 CLR 一致。CLS 对编译器的要求之一就是编译器必须支持 CTS 中定义的一定的类型。这样做的好处是因为所有的.NET 语言都使用一个单一的类型系统，确保了在不同语言中创建的对象和类型可以无缝连接。正是这种 CTS/CLS 的组合使得不同语言的互用不再仅仅是程序员们的梦想了。

4.8.2 单根的对象层次结构

前面我们已经提到过，CTS 的一个重要特征就是对象层次结构的单根性。在.NET 框架中，系统中的每一个类型都是从 System.Object 基类中派生出来的。这一点是与 C++语言的重要的不同，在 C++语言中没有一个暗含的所有类的基类。这种单一基类的思路已经被 OOP 理论所采用，并在大多数面向对象的主流语言中得到实现。在您开始学习面向对象编程时，单根层次的好处不会马上显现出来，但随着时间的流逝，您就会产生这样的问题：在采用这种层次结构的类型之前，编程语言都是怎样设计出来的？

单根的对象层次结构是统一类型系统的关键，因为它保证了层次结构中的每个对象都拥有一个共同的接口，因此层次结构中的一切到最后都可以变成相同的基本类型。C++的一个主要缺陷就是不支持这样的层次结构。下面来考虑一个简单的示例以更好地理解这层意思。

比如说您已经在 C++中构建了一个基于您自己创建的基类的对象层次结构，我们将这

个基类叫做“CFoo”。现在您希望将它与另外一个对象层次结构集成起来，这个对象层次结构中的所有对象都是从一个叫做“CBar”的基类中派生出来的。在这个例子中，对象层次结构的接口并不兼容，因此要集成这两个层次结构的话，需要付出相当多的努力。您可能必须使用某些带聚集的包装类或多继承才能完成这样的任务。而在单根的层次结构中，兼容性不再是个问题，因为每个对象都有相同的接口(从 System.Object 中继承的)。这样您就知道，在您的层次结构中以及每一个对象和第三方.NET 代码的层次结构中的每一个对象都至少拥有一些共同的功能。

4.8.3 类型安全

作者想提及的最后一个CTS的好处就是类型安全。类型安全保证了类型总是名副其实，并且在特定的类型上只能进行适当的操作。类型安全提供了很多好处和功能，如下所述，其中大多数都得自于对象层次结构的单根性。

- 每一个对对象的引用都具有某种类型，它所引用的对象也具有某种类型。CTS 保证了引用总是指向希望它指向的内容。
- 因为 CTS 跟踪系统中的每一个类型，因此不能欺骗系统使它认为的某种类型实际上是另外一种类型。在需要保证安全性的分布式应用程序中，这是一个特别受到关注的方面。
- 每一种类型都可以通过指定访问限定符来为它的各个成员逐一地确定其能否被访问。访问的类型包括任意访问(public)、只限继承的类访问(protected)、禁止访问(private)以及只限当前编译单元的类型访问(internal)。下一章我们将更详细地讨论访问限定符。

4.9 本章小结

公共类型系统(CTS)是.NET 框架的一个重要特性。CTS 定义了应用程序要想在 CLR 中正常运行所必须遵循的类型系统规则。CTS 类型分为两类：引用类型和数值类型。名称空间用于在应用程序中定义作用域。公共的类型系统的好处包括语言互用性、单根的对象层次结构以及类型安全。在 C#中类型可以通过装箱和开箱进行转换，以便兼容的类型共享某些特性和功能。

第5章 类

类是每一种面向对象编程语言的灵魂。我们在第1章中已经提到，类就是数据和加工数据的方法的一个封装。这是在任何面向对象语言中都“颠扑不破的真理”。而一种面向对象语言与其他面向对象语言的不同之处就在于作为成员存储的数据的类型和每一种类的功能。C#在有关类和其他许多功能上都从C++和Java处借用了一些概念，但同时又以它的独创性为一些旧的问题提出了新颖的解决方案。

在本章，我们将首先介绍在C#中定义类的一些基本知识，包括实例成员、访问限定符、构造函数以及初始化列表，然后将介绍静态成员和常量与只读字段之间的区别；之后，我们将介绍析构函数(destructor)和所谓的“确定性结束”(deterministic finalization)；最后，本章以对继承和C#类的简短讨论作为结束。

5.1 定义类

在C#中定义类的语法很简单，特别是对于经常在C++或Java中编程的人来讲。您只需在类的名称前加上关键字“class”，然后在大括号之间插入该类的成员即可，如下所示：

```
class Employee
{
    private long employeeId;
}
```

如您所见，这个类的确很简单。它的名称叫做“Employee”，包含了单个成员“employeeId”。注意在该成员之前有关键字“private”，这叫做“访问限定符”。C#中定义了四种合法的访问限定符，稍后我们就会涉及到它们。

5.2 类的成员

在第4章中，我们讨论了由公共类型系统(Common Type System, CTS)定义的不同的类型。这些类型就是C#所支持的类的成员，包括以下内容：

- 字段

  字段就是用于保持一个值的成员变量。用 OOP 的用语来说，字段有时也称为对象的数据。根据您希望怎样使用一个字段，您可以对该字段应用多个限定符。这些限定符包括“static”、“readonly”和“const”，稍后我们将了解这些限定符的含义以及如何使用它们。

- 方法

  方法就是加工对象的数据(或字段)的真正的代码。在本章我们只着重介绍定义类的数据，在第 6 章中，将更加详细地介绍方法。

- 属性

  属性有时也叫做“智能字段”，因为对于使用类的客户而言，它们就是看起来像字段的真正的方法。这使得客户可以拥有更高程度的抽象，因为客户不需要知道它访问时是直接访问了字段还是访问了被调用的存取器方法。第 7 章将详细介绍属性。

- 常量

  如其名所示，常量就是不能改变其值的字段。本章稍后将讨论常量并与所谓的只读字段进行比较。

- 索引器

  与属性是智能字段一样，索引器就是智能数组。也就是说，索引器可以让对象通过“get”和“set”存取器方法来被索引。索引器允许您在对象中轻松实现索引，以便设置或浏览当前值。索引器和属性都将在第 7 章中讨论。

- 事件

  事件就是导致一些代码运行的东西。事件是 Microsoft Windows 编程中不可缺少的组成部分。比如，当鼠标移动或窗口被选中或调整大小时，就会触发事件。C#事件采用了标准的“发布/预订”设计模型(请参见 Microsoft Message Queuing, MSMQ)和 COM+异步事件模型(这使得应用程序具备了异步事件处理功能)，但是在 C#中，这种设计模型是语言中内置的“头等”(first-class)概念。第 14 章将介绍如何使用事件。

- 操作符

  C#通过操作符重载允许您可以将标准数学运算操作符添加到类中，以便使用这些操作符编写出更直接的代码。第 13 章将详细介绍操作符重载。

5.3 访问限定符

既然我们已经提到可以被定义为 C#类成员的不同类型，下面就来看看一些重要的限定

符，它们用于指定一个给定成员对于它所在的类之外的代码而言是可见的、还是不可见的。这些限定符叫做“访问限定符”，如表 5.1 所示：

表 5.1 C#访问限定符

访问限定符	说 明
Public	表示该成员可以从它所在的类定义和派生类的层次结构之外来访问
protected	表示该成员在它所在的类之外是不可见的，只允许派生类访问
Private	表示该成员在类定义的范围以外不可访问，即使派生类也不能访问
Internal	表示该成员只在当前编译单元内可见，根据代码所在的位置，访问限定符“internal”代表了“public”和“protected”的访问性组合

除非您希望某个成员拥有默认的访问限定符“private”，否则，您就必须为它指定一个访问限定符。这一点与 C++有所不同：在 C++中，一个不带显式访问限定符的成员会延续先前声明的访问限定符指明的访问性特征。比如，在下面的 C++代码中，成员 a、b 和 c 被定义为“public”，成员 d 和 e 被定义为“protected”成员：

```
class CAccessModsInCpp
{
   public:
   int a;
   int b;
   int c;
   protected:
      int d;
      int e;
}
```

为了在 C#中达到同样的目的，以上代码必须修改为如下所示：

```
class AccessModsInCSharp
{
   public int a;
   public int a;
   public int a;
   protected int d;
   protected int d;
}
```

下面这段 C#代码会使成员 b 被声明为“private”：

```
public MoreAccessModsInCSharp
```

```
{
    public int a;
    int b;
}
```

5.4 Main方法

每一个 C#应用程序都必须在它的某个类中定义一个方法 Main。此外，这个方法还必须被定义为“public”和“static”(我们马上就会知道“static”的含义)。对于 C#编译器来说，究竟哪个类定义了方法 Main 无关紧要，您选中哪个类来定义方法 Main 也不会影响到编译的顺序。这一点与 C++中构建应用程序时必须密切关注类之间的依赖性有所不同。C#编译器具备足够的智能来“穿越”您的源文件以便独立定位方法 Main。此外，这个如此重要的方法也是所有 C#应用程序的入口。

尽管您可以将方法Main放在任何一个类中，但还是建议您创建一个专门管理方法Main的类。下面是我们(到目前为止)简单的 Employee 类如何处理方法 Main 的代码：

```
class Employee
{
    private int employeeId;
}

class AppClass
{

    static public void Main()
    {
        Employee emp = new Employee();
    }
}
```

您可以看到，这个例子包含两个类。在 C#编程中，即便是最简单的应用程序，这也是一种通常的思路。第一个类(Employee)是问题域类，第二个类(AppClass)包含了应用程序所需的入口(Main)。这种情况下，方法 Main 将实例化对象 Employee，如果这是一个真正的应用程序的话，方法 Main 还将使用 Employee 对象的成员。

5.4.1 命令行参数

将方法 Main 声明为只带有单个字符串数组类型的参数，您就可以访问应用程序的命令

行参数了。这时，参数可以按您所希望的处理成任何数组。尽管要到第 7 章我们才会介绍数组，但这里还是给出了一段比较简单的代码，用来说明应用程序的命令行参数以及如何将它们写入到标准输出设备：

```
using System;
class CommandLineApp
{
    public static void Main(string[] args)
    {
        foreach (string arg in args)
        {
            Console.WriteLine("Argument: {0}", arg);
        }
    }
}
```

下面就是使用几个随机选择的值来调用这个应用程序产生的结果：

```
e:>CommandLineApp 5 42
Argument: 5
Argument: 42
```

命令行参数以字符串数组的形式提供给您。如果希望处理标志或选项这样的参数，您必须自己编程实现这样的功能。

注意： Microsoft Visual C++的开发人员早已熟悉应用程序中代表命令行参数值的数组，但与 Visual C++不同的是，C#中的命令行参数值数组不包含应用程序的名称，也不将它作为数组中的第一个元素。

5.4.2 返回值

本书中大部分示例程序定义方法 Main 的方式如下：

```
class SomeClass
{
    ...
public static void Main()
{
...
}
...
}
```

不过，您也可以定义Main返回一个int类型的值。尽管这在GUI应用程序中不常用，但当您编写要批处理运行的控制台应用程序时这会很有用。Return语句结束对方法的执行，返回值就可以作为上一级调用的应用程序或批处理文件的错误级别，用于表示用户自定义的成功或失败。要想这样，您可以使用下面这个原型：

```
public static int Main()
{
...
    // Return some value of type int
    // that represents success or value.
    return 0;
}
```

5.4.3 多个Main方法

C#的设计者们在C#中包含了这样一种机制：您可以定义一个以上的带有Main方法的类。为什么会这样做呢？原因之一就是为了在类中放置测试代码。然后，您就可以使用C#编译器所带的选项选项“/main:<className>”来指定使用哪个类的Main方法。下面这个例子就包含了两个带有Main方法的类：

```
using System;

class Main1
{
    public static void Main()
    {
        Console.WriteLine("Main1");
    }
}

class Main2
{
    public static void Main()
    {
        Console.WriteLine("Main2");
    }
}
```

在编译这个应用程序时，如果想使用Main1.Main方法作为该应用程序的入口，您就应该使用如下的选项选项：

```
csc MultipleMain.cs /main:Main1
```

将选项选项改为“/main:Main2”就表示使用 Main2.Main 方法作为应用程序的入口。

显然，由于 C#语言对大小写敏感，您在选项选项中必须注意使用类的名称时大小写一定要正确。另外，如果试图编译包含多个定义了 Main 方法的类的应用程序而不指定选项选项“/main”的话，就会发生编译错误。

5.5 构造函数

像 C#这样的面向对象编程语言最大的好处之一就是：您可以定义一些特殊的方法，无论何时创建类的实例时，总可以调用这些方法。这样的方法就叫做“构造函数”(constructor)。C#还引入了一种新的构造函数，叫做“静态构造函数”，在下一节中我们将讨论这种构造函数。

使用构造函数的关键好处在于：它能保证对象被使用之前经过合适的初始化过程。当用户实现一个对象时，该对象的构造函数首先被调用，并在用户对该对象执行进一步操作之前必须返回。这种保证有助于确保对象的完整性，也有助于使用面向对象语言编写的应用程序更加可靠。

不过，我们应该怎样为构造函数命名才能使编译器在实现对象时能够知道如何调用它呢？C#的设计者们效仿了 Stroustrup，规定在 C#中类的构造函数必须与类同名。下面这个简单的类包含了一个同样简单的构造函数：

```
using System;

class ConstructorlApp
{
    ConstructorlApp()
    {
        Console.WriteLine("I'm the constructor.");
    }

    public static void Main()
    {
        ConstructorlApp app = new ConstructorlApp();
    }
}
```

构造函数不返回任何值。如果您试图为构造函数加上一个某种类型的前缀，编译器就会发出错误提示，告诉您不能定义与成员所在类型同名的成员。

您还应该在 C#中指定对象被实例化的方式。使用关键字“new”时参照下面的语法就可以了：

<class> <object> = new *<class>(constructor arguments)*

如果您具有C++背景，在这一点上要特别注意。在C++中，实例化对象有两种方式。您可以在堆栈上声明它，如：

```
//C++code.This creates an instance of Cmyclass on the stack.
Cmyclass myClass;
```

也可以在自由存储区(或堆)上使用C++关键字“new”来实例化一个对象：

```
//C++code.This creates an instance of CmyClass on the heap.
CMyClass myClass =new CMyClass();
```

在C#中实例化对象有所不同，这也是C#开发新手容易发生混淆的地方之一。发生这种混淆的原因在于这两种语言创建对象时使用的关键字相同。尽管在C++中使用关键字“new”可以让您指定对象在何处创建，但是在C#中创建一个对象却取决于被实例化的类型。在第4章中我们已经知道，引用类型是在堆上创建的，而数值类型是在堆栈上创建的。因此，在C#中关键字“new”可以让您创建对象的新实例，但却不会决定该对象在何处创建。

既然如此，下面的代码就是合法的C#代码，但如果您是一位C++开发人员的话，这段代码的含义可能跟您想的不一样：

```
MyClass myClass;
```

在C++中，这行代码将在堆栈上创建一个MyClass的实例。在前面我们已经提到，在C#中只能使用关键字“new”来创建对象，因此，在C#中这行代码只是声明了myClass是MyClass类型的一个变量，但它并没有实例化这个对象。

作为这样的一个例子，如果您编译下面的程序，C#编译器就会警告您在这个应用程序中声明过的变量从没有使用：

```
using System;

class Constructor2App
{
    Constructor2App()
    {
        Console.WriteLine("I'm the constructor");
    }

    public static void Main()
    {
        Constructor2App app;
    }
}
```

因此，如果声明了一个对象类型，就应该在代码中某个地方使用关键字“new”来将它实例化：

```
Constructor2App app;
app = new Constructor2App();
```

为什么有时候会声明一个对象而不将它实例化呢？如果是在一个类中声明另一个类时，就应该在使用对象之前声明这些对象。类的这种嵌套就叫做“包容”(containment)，或者“聚集”(aggregation)。

5.5.1 静态成员和实例成员

在 C++中，您可以将类的成员定义为静态成员或实例成员。默认情况下，每一个成员都被定义为实例成员，表示类的每一个实例都会复制这些成员。如果成员被声明为静态成员，该成员就只被复制一次。静态成员是在载入包含类的应用程序时创建的，在应用程序的整个生命期内它都存在。因此，您甚至可以在类被实例化之前就访问这样的成员。不过，为什么会这样做呢？

有一点与 Main 方法有关。公共语言运行环境(CLR)需要一个通向应用程序的公共的入口。这样 CLR 就不需实例化任何一个对象，只要在某个类中定义了一个名为 Main 的静态方法即可。另外，从面向对象的角度看来，如果某个方法只在语义上属于某个类而并不需要实际的对象时，也需要使用静态成员——比如，如果您希望在应用程序的生命期内跟踪一个给定的对象有多少个实例，因为静态成员也存在于对象实例中，因此可以编写出下面的代码：

```
using System;

class InstCount
{
    public InstCount()
    {
    instanceCount++;
    }
     static public int instanceCount = 0;
}

class AppClass
{
    public static void Main()
    {
        Console.WriteLine(InstCount.instanceCount);
```

```
        InstCount ic1 = new InstCount();
        Console.WriteLine(InstCount.instanceCount);

        InstCount ic2 = new InstCount();
        Console.WriteLine(InstCount.instanceCount);
    }
}
```

这个例子的输出结果可能是：

```
0
1
2
```

关于静态成员的最后一个提示就是：静态成员必须有一个合法的值。在定义成员时就可以指定这个值，如：

```
static public int instanceCount1 = 10;
```

如果您没有初始化这个变量，CLR 就会在应用程序启动时使用默认值“0”对它进行初始化。因此，下面两行代码实际上是等价的：

```
static public int instanceCount2;
static public int instanceCount2 = 0;
```

5.5.2 构造函数的初始化函数

所有的 C#对象构造函数——除了 System.Object 构造函数以外——在执行构造函数的第一行代码之前都会调用其基类的构造函数。这些构造函数初始化函数允许您指定希望调用哪个类和哪个构造函数，它们具有两种形式：

- 形如“base(...)”的初始化函数允许调用当前类的基类的构造函数，也就是由发出调用指令的构造函数的名称所指定的特定的构造函数。
- 形如“this(...)”的初始化函数允许当前类调用它内部定义的另一个构造函数。当您重载了多个构造函数，而希望总是调用一个默认的构造函数时，就应该使用这种初始化函数。重载方法将在第 6 章讨论，这里给出一个粗略的定义：重载方法就是两个以上具有相同名称但参数值列表不同的方法。

为了看看实际操作中的顺序，注意下面的代码将首先执行类 A 的构造函数，然后执行类 B 的构造函数：

```
using System;

class A
```

```
{
    public A()
    {
        Console.WriteLine("A");
    }
}

class B : A

{
    public B()
    {
        Console.WriteLine("B");
    }
}

class DefaultInitializerApp
{
    public static void Main()
    {
        B b = new B();
    }
}
```

上面的代码在功能上与下面的代码相同，下面的代码中显式调用了基类的构造函数：

```
using System;

class A
{
    public A()
    {
        Console.WriteLine("A");
    }
}

class B : A
{
    public B() : base()
    {
        Console.WriteLine("B");
    }
}

class BaseDefaultInitializerApp
```

```
{
    public static void Main()
    {
        B b = new B();
    }
}
```

下面我们来看一个展示构造函数初始化函数作用的更好的例子。同样，我们有两个类：A 和 B。这次，类 A 有两个构造函数，一个没有参数值，一个带有一个 int 类型的参数值。类 B 有一个带有一个 int 类型参数值的构造函数。在类 B 的构造函数中就可能会发生问题。如果运行下面的代码，就会调用类 A 不带参数值的那个构造函数：

```
using System;

class A
{
    public A()
    {
        Console.WriteLine("A");
    }

    public A(int foo)
    {
        Console.WriteLine("A = {0}", foo);
    }
}

class B : A
{
    public B(int foo)
    {
        Console.WriteLine("B = {0}", foo);
    }
}

class DerivedInitializer1App
{
    public static void Main()
    {
        B b = new B(42);
    }
}
```

那么，怎样才能保证调用我们希望调用的类 A 的构造函数呢？只需告诉编译器我们希

望调用初始函数列表中的哪个构造函数即可，如：

```
using System;

class A
{
    public A()
    {
        Console.WriteLine("A");
    }

    public A(int foo)
    {
        Console.WriteLine("A = {0}", foo);
    }
}

class B : A
{
    public B(int foo) : base(foo)
    {
        Console.WriteLine("B = {0}", foo);
    }
}

class DerivedInitializer2App
{
    public static void Main()
    {
        B b = new B(42);
    }
}
```

注意： 与 Visual C++不同，在 C#中您不能使用构造函数初始化函数去访问当前类中除构造函数以外的其他实例成员。

5.6 常量和只读字段

您可能常常会将在应用程序执行过程中不希望改变的元素作为字段使用，比如，应用程序所依赖的数据文件、某个数学类中的 π 值、或者应用程序中任何您所知的永远不会改变的值。为了解决这些问题，C#允许定义两种极其相似的成员类型：常量和只读字段。

5.6.1 常量

从这个名字中就可以看出，由关键字“const”表示的常量就是在应用程序的生命期内保持不变的字段。在定义常量时只需记住两点：第一，常量的值是在编译时设定的，可以由程序员设定，默认状态是编译器设定；第二，常量成员的值必须是数值文字。

要将字段定义成常量，只需在被定义的成员前指定关键字“const”即可，如：

```
using System;

class MagicNumbers
{
    public const double pi = 3.1415;
    public const int answerToAllLifesQuestions = 42;
}

class ConstApp
{
    public static void Main()
    {
        Console.WriteLine("pi = {0}, everything else = {1}",
            MagicNumbers.pi, MagicNumbers.answerToAllLifesQuestions);
    }
}
```

注意这段代码中有一个关键的地方：客户代码不需要实例化 MagicNumbers 类，因为默认状态下常量成员是静态的。为了更清楚地认识这点，不妨看看下面这两个成员产生的 MSIL 代码：

```
answerToAllLifesQuestions:public static literal int32 =int32
(0x0000002A)
pi :public static literal float64 =float64(3.1415000000000002)
```

5.6.2 只读字段

定义为常量的字段是很有用的，因为它清楚地体现了程序员的意图：即该字段包含的是一个不变的值。不过，这只有在编译时您就知道常量值的情况下才起作用。那么，如果字段的值要到运行的时候才知道、而一旦被初始化后就不再改变，这时程序员该怎么办呢？这个问题在别的编程语言中通常都不会考虑到，但 C#语言的设计者们通过所谓的“只读字段”解决了这个问题。

当您使用关键字“readonly”定义一个字段时，您可以在这个地方设置该字段的值：构

造函数。除此之外，类本身或使用类的客户代码就都不能改变该字段的值了。比如，您希望在一个图形应用程序中追踪屏幕的分辨率，您不能使用常量来解决这个问题，因为直到运行的时候，应用程序才能决定终端用户的屏幕分辨率。因此，您可以使用类似下面这样的代码：

```
using System;

class GraphicsPackage
{
    public readonly int ScreenWidth;
    public readonly int ScreenHeight;

    public GraphicsPackage()
    {
        this.ScreenWidth = 1024;
        this.ScreenHeight = 768;
    }
}

class ReadOnlyApp
{
    public static void Main()
    {
        GraphicsPackage graphics = new GraphicsPackage();
        Console.WriteLine("Width = {0}, Height = {1}",
                          graphics.ScreenWidth,
                          graphics.ScreenHeight);
    }
}
```

初看起来，这段代码好像就是您所需要的。不过，这里还存在一个小问题：我们定义的只读字段都是实例字段，也就是说，用户必须实例化类之后才能使用这些字段。如果在实例化类的方式将决定只读字段值的情形下，这也许正是您所希望的。但是，如果您想要的是一个常量——按定义是静态的、在运行的时候才被初始化，又该怎么办呢？这时，您就应该给字段加上限定符“static”和“readonly”，然后再创建一个特殊类型的构造函数，叫做“静态构造函数”。静态构造函数就是用来初始化静态字段(不管只读与否)的构造函数。下面我们将前面那个例子中的屏幕分辨率字段修改为静态和只读，并添加了一个静态构造函数。注意在构造函数定义时添加的关键字“static”：

```
using System;

class GraphicsPackage
```

```
{
    public static readonly int ScreenWidth;
    public static readonly int ScreenHeight;

    static GraphicsPackage()
    {
        // Code would be here to
        // calculate resolution.
        ScreenWidth = 1024;
        ScreenHeight = 768;
    }
}

class ReadOnlyApp
{
    public static void Main()
    {
        Console.WriteLine("Width = {0}, Height = {1}",
                          GraphicsPackage.ScreenWidth,
                          GraphicsPackage.ScreenHeight);
    }
}
```

5.7 对象清除和资源管理

基于组件系统最重要的属性之一就是在组件终止之后能够进行资源清除和释放。通过“资源清除和释放”，即对其他组件引用的及时释放，以及有可能引起争夺的稀缺资源或受限资源的释放，如数据库连接、文件处理和通信端口等。“终止”表示到那时该对象不再被使用。

在C++中，清除是很直截了当的，因为清除是在对象的析构函数中完成的。每个C++对象都会定义这个函数以便在对象脱离作用域时自动执行。在Microsoft .NET环境下，对象的清除是由.NET垃圾收集器(Garbage Collector, GC)来自动处理的。这种策略还在引起争议，因为与C++中可以预测何时执行对象的终止代码相比，.NET的这种解决方案是一种“懒惰的”模式。GC利用在后台运行的线程来决定对象何时不再被引用，然后由其他的GC线程负责运行对象的终止代码。大多数情况下这就够了，但是在需要以及时和可预测的方式释放资源的情况下，它并不是最优化的解决方案。您可能已经感觉到了，这不是一个很容易解决的问题。在本节中将介绍对象终止和资源管理的问题，以及创建生命期可预测的对象时您该怎么做(注意本节的内容在很大程度上基于Brian Harry在在线公共论坛上对C#资

源管理的出色解释)。

5.7.1 历史回顾

几年前，当.NET 项目刚刚启动时，关于资源管理的争论就很激烈。.NET 的早期开发者们不仅来自 COM 和 Microsoft Visual Basic 开发小组，而且来自 Microsoft 的许多其他开发小组。最大的问题之一就是这些小组面临着如何处理有关引用计算的问题，包括循环和因为误用而产生的错误等等。这样的例子就是循环引用的问题：第一个对象包含了对第二个对象的引用，而第二个对象又包含了对第一个对象的引用。何时以及如何释放这样的循环引用可能会引起问题。不释放这样的引用或者只释放一个都会引起内存泄漏，而这种泄漏又特别难以跟踪。做过大量 COM 编程的人都曾有过这样的经历：为了找出由于引用计算导致的臭虫而不得不推迟工作进度。Microsoft 已经意识到引用计算问题在真正的基于组件的应用程序 (比如 COM 应用程序)中非常普遍，因此开始在.NET 中提供一种统一的解决方案。

针对这个问题的最初解决方案基于自动的引用计算——这样即使程序员忘了引用计算也不要紧，以及自动检测和处理循环的方法。此外，.NET 开发小组还考虑添加保守的引用收集、跟踪收集以及无需彻底追踪而只收集单个对象的 GC 算法。不过，由于各种各样的原因——我们将简短介绍这些不同的思路——最终的结果是：这些解决方案在通常情况下都不会起作用。

一个主要的障碍是：在.NET 开发早期，高度保持与 Visual Basic 的兼容性是一个主要目标。因此，解决方案对于 Visual Basic 语言本身来说，必须是完整而透明的，不做任何语义上的改动。经过许多争论之后，最终的解决方案是一个基于上下文的模型，存在于确定的上下文中的一切都将在 GC 中使用引用计算，而不在此上下文范围之内的就只使用 GC。这样的确有助于避免一些类型交叉问题，但对于在不同语言之间代码混合而言，这并不是一种好的解决方案。因此，设计者们决定对 Visual Basic 语言做出一系列改动，以使它“现代化”并更加强大。决定的一部分就是降低对 Visual Basic 生命期兼容性的要求。这个决定也结束了对确定性的生命期问题的调查。

5.7.2 确定性结束

回顾了一下历史之后，下面我们来介绍确定性结束。一旦确定不再使用某个对象之后，就执行该对象的终止代码，并释放该对象掌握的对其他对象的引用。这个终止过程很自然地贯穿过层叠对象的各个层次，从最上层的对象开始。理想情况下，对于共享的对象或单独使用的对象，这个解决方案都要起作用。

注意这里对于时间没做任何承诺。GC 的一个线程一旦发现某个引用不再被使用之后，这个线程不会执行任何操作，直到该对象的终止代码执行。不过，处理过程中总会发生 CPU 上下文切换，从应用程序的角度看来，在完成这个步骤之前总是会消耗一定的时间。先前我们已经提到过，实际上存在很多这样的情形：应用程序可能会关心执行终止代码的及时性和顺序。这样的情形通常与资源有关，这也是常常引起争议的地方。下面是一些对象不再使用时需要立即释放的资源：

- 内存

  释放对象的内存，将内存快速返还到池中以便使用。

- 窗口句柄

  窗口对象在 GC 中的足迹并没有反映出实际的资源开销。在操作系统中代表窗口的可以分配的窗口句柄的总数甚至比可用的内存还要有限。

- 数据库连接

  并发的数据库连接很频繁地发生，可用的连接数量可能会变得有限，因此及时将不用的连接返回到池中很重要。

- 文件

  由于给定的文件只有单个实例而很多操作又要求独占访问，因此在文件不再使用时关闭文件句柄很重要。

5.7.2.1 引用计算收集

引用计算(reference counting)在很多情况下都以合理的方式提供了确定性结束。但是要指出的是：在相当多的情况下，引用计算也不起作用，最常举出的例子就是循环。实际上，直截了当的引用计算从来不会收集循环中的对象。我们大多数人都曾“痛苦地”学习过手动管理这些的技术，但是使用这些技术又会成为不希望看到的产生错误的源头。同样，如果您在单元中使用远程对象，在重新进入单元时就会将许多不确定的因素导入到您的程序中。有人可能会说，您一旦将引用传递给控制之外的对象时，您就失去了最终决定权，因为您不知道这个“外来的”代码何时、甚至是否会将该引用释放。还有人认为依赖于层次对象终止顺序的复杂系统先天就不足，很可能随着代码的改进而引起很显著的维护问题。

5.7.2.2 跟踪收集

跟踪收集器(tracing collector)比引用计算还要稍逊一筹。对于终止代码执行而言，它是一种更懒惰的系统。对象都有结束符，结束符是一种当程序不再使用对象时要执行的方法。跟踪收集的优点就是循环不再成为问题，更大的优点是分配引用就是一个简单的移动操作。您付出的代价就是引用不再使用时没有承诺“立即”运行终止代码。那么，它都承诺了些什么呢？事实上，对于“行为端正”的程序——行为不端的程序就是将终止符线程强加入

有限的循环——系统将为对象调用这些终止符。联机文档在这方面的承诺总是倾向于过分小心谨慎，但如果对象中有这个 Finalize 方法的话，系统就会调用它。这样也并没有解决确定性结束的问题，不过重要的是理解资源将被收集，而结束符对于阻止程序中的资源泄漏是一条有效的途径。

5.7.3 性能

由于性能与确定性结束这个问题有关，因此性能本身也是一个很重要的争论焦点。.NET 开发小组认为必须存在某些跟踪收集器以便处理循环，因此跟踪收集器也就成了很大一部分的开销。他们同样认为代码执行的性能受引用计算花销的影响很大，不过好在应用程序运行时分配的所有对象中真正需要确定性结束的那些对象并不多。但是，通常很难将性能开销与那些对象脱离开来。下面我们来看看使用跟踪收集器进行简单的引用分配的一些伪代码，然后再看看使用引用计算的伪代码：

```
// Tracing.
a = b;
```

这就可以了。编译器会将上面的代码转换成单个的移动指令，在某些情形下，它甚至会对整个过程进行优化。

```
// Ref counting.
if (a != null)
   if (InterlockedDecrement(ref a.m_ref) == 0)
      a.FinalRelease();

if (b != null)
   InterlockedIncrement(ref b.m_ref);

a = b;
```

这段代码非常臃肿——工作区更大、代码执行的性能也不见更高，尤其是那两个互锁的指令。您可以把这些代码放进“更有帮助”的方法中以便减少代码臃肿，但这样又会增加代码的长度。此外，当您加进所有必要的 try 块时，最终也会影响代码的生成，因为优化器面对这么多的例外处理代码肯定会忙得不可开交，特别是在非控 C++中这种情形尤甚。值得指出的是：每一个对象由于额外的引用计算字段，都会多用四个字节，因此反过来又增加了内存的使用和工作区的面积。

接下来的两个示例程序展示了这种开销——它们由.NET 开发小组提供。这个测试程序首先执行一个循环，然后是分配对象、执行两个赋值语句、退出一个引用的作用域。与任何测试程序一样，这个程序也允许一些主观解释。有人甚至会说，这个程序中大多数的引

用计算都可以被优化掉。这可能是正确的，但是这个程序在这里的作用主要是展示一下效果。在真正的应用程序中，如果有可能进行优化，这种优化也是很难进行的。事实上，在C++中程序员都是手动进行这种优化，而这又可能导致产生有关引用计算的错误。因此，要注意在真正的程序中赋值与分配的比率都比这里高得多。

下面是第一个示例——ref_gc.cs，它是基于跟踪GC的：

```
using System;

public class Foo

{
    private static Foo m_f;
    private int m_member;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;

        for (int i=0; i < 10000000; ++i)
        {
            Foo f = new Foo();

            // Assign static to f2.
            f2 = m_f;

            // Assign f to the static.
            m_f = f;

            // f goes out of scope.
        }

        // Assign f2 to the static.
        m_f = f2;

        // f2 goes out of scope.

        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Ticks = {0}", ticks);
    }

    public Foo()
```

```
        {
        }
    }
```

下面是第二个示例程序——ref_rm.cs，它是为了线程安全而使用互锁操作的引用计算版本：

```
using System;
using System.Threading;

public class Foo
{
    private static Foo m_f;
    private int m_member;
    private int m_ref;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;

        for (int i=0; i < 10000000; ++i)
        {
            Foo f = new Foo();

            // Assign static to f2.
            if (f2 != null)
            {
                if (Interlocked.Decrement(ref f2.m_ref) == 0)
                    f2.Dispose();
            }
            if (m_f != null)
                Interlocked.Increment(ref m_f.m_ref);
            f2 = m_f;

            // Assign f to the static.
            if (m_f != null)
            {
                if (Interlocked.Decrement(ref m_f.m_ref) == 0)
                    m_f.Dispose();
            }
            if (f != null)
                Interlocked.Increment(ref f.m_ref);
            m_f = f;
```

```
            // f goes out of scope.
            if (Interlocked.Decrement(ref f.m_ref) == 0)
                f.Dispose();
        }

        // Assign f2 to the static.
        if (m_f != null)
        {
            if (Interlocked.Decrement(ref m_f.m_ref) == 0)
                m_f.Dispose();
        }
        if (f2 != null)
            Interlocked.Increment(ref f2.m_ref);
        m_f = f2;

        // f2 goes out of scope.
        if (Interlocked.Decrement(ref f2.m_ref) == 0)
            f2.Dispose();

        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Ticks = {0}", ticks);
    }

    public Foo()
    {
        m_ref = 1;
    }

    public virtual void Dispose()
    {
    }
}
```

注意在上面的代码中只有一个线程，因此不存在总线争夺，使得这成了一种“理想的”情况。您可能还可以对上面的代码进行加工，但能做的不多。同样值得指出的是：Visual Basic 传统上并不需要考虑为它的引用计算使用互锁操作(但 Visual C++需要)，在 Visual Basic 以前的发行版本中，一个组件在一个单线程的单元中运行，因而保证了每次只有单个的线程在执行。Visual Basic .NET 的目标之一就是多线程编程，另一个目标是去掉 COM 多线程模型的复杂性。对于希望看到不使用锁定前缀的读者来说，下面还有一个示例程序——ref_rs.cs，同样由.NET 开发小组提供。这是一个假定在单线程环境下运行的引用计算的版本，这个程序几乎不会像多线程版本那样慢，但仍然比 GC 版本慢得多。

```
using System;

public class Foo
{
    private static Foo m_f;
    private int m_member;
    private int m_ref;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;

        for (int i=0; i < 10000000; ++i)
        {
            Foo f = new Foo();

            // Assign static to f2.
            if (f2 != null)
            {
                if (--f2.m_ref == 0)
                    f2.Dispose();
            }
            if (m_f != null)
                ++m_f.m_ref;
            f2 = m_f;

            // Assign f to the static.
            if (m_f != null)
            {
                if (--m_f.m_ref == 0)
                    m_f.Dispose();
            }
            if (f != null)
                ++f.m_ref;
            m_f = f;

            // f goes out of scope.
            if (--f.m_ref == 0)
                f.Dispose();
        }

        // Assign f2 to the static.
```

```
            if (m_f != null)
            {
                if (--m_f.m_ref == 0)
                    m_f.Dispose();
            }
            if (f2 != null)
                ++f2.m_ref;
            m_f = f2;

            // f2 goes out of scope.
            if (--f2.m_ref == 0)
                f2.Dispose();

            ticks = Environment.TickCount - ticks;
            Console.WriteLine("Ticks = {0}", ticks);
        }

        public Foo()
        {
            m_ref = 1;
        }

        public virtual void Dispose()
        {
        }
    }
```

显然，这段代码中包含了很多变量。运行这三个应用程序的结果是：GC 版本的程序运行速度比单线程引用计算版本的程序快 2 倍，比带有锁定前缀的引用计算版本的程序快 4 倍。作者个人的经验如下——要注意这些数字代表的是在 IBM ThinkPad 570 上运行.NET Framework SDK Beta 1 版本的编译器时的平均值：

```
GC Version (ref_gc)                          1162ms
Ref Counting (ref_rm) (multi-threaded)       4757ms
Ref Counting (ref_rs) (single-threaded)      1913ms
```

5.7.4 完美的解决方案

大多数人都会同意这样一个观点，那就是对于这个问题的完美的解决方案就是创建这样的一个系统：在这个系统中每个对象的分配、使用和回收都很廉价。在这样的理想系统中，一旦程序员发现不再需要某个对象(不管是否存在循环)，那个对象就以一种确定的、有序的方式释放掉。.NET 开发小组为了解决这个问题花费了大量的时间和精力之后，他们

认为达到这样的目的的惟一方式就是：将跟踪 GC 与引用计算结合起来。测试程序的数据反映出在编程环境中为所有的对象以通用目的的方式使用引用计算的代价太昂贵了——代码更长、工作区更大。如果您将这个已经存在的高开销与为实现循环回收的跟踪收集器的额外开销组合起来的话，内存管理的代价就太大了。

应该指出的是，在.NET 开发早期，曾经研究过各种各种的技术，以便找到一种改进引用计算性能的方法。以前也的确出现过构建在引用计算之上的相对高性能的系统。但是，经过仔细的测试之后发现，这样的系统之所以能够改进性能是因为放弃了一些确定性。

说句“题外话”，C++/COM 程序就没有受到上述问题的困扰。大多数内部使用 COM、使用 C++类的高性能 C++程序都要求程序员显式管理内存。C++程序员通常使用 COM 只是为他们的客户创建接口。这是一个允许这类程序性能表现良好的关键属性。但这显然不是.NET 的目标，.NET 中内存管理的问题主要是由 GC 处理，而不是程序员。

5.7.5 近乎完美的解决方案

与生活中的大多数情况类似，我们不能够得到有关资源管理的完美的解决方案。但是，如果我们只希望在需要的地方得到确定性结束又会怎样呢？.NET 开发小组在这上面花了不少时间，要知道那时的情形是要在新系统内完全复制 Visual Basic 6 的语义。大多数当时的分析后来都应用了，但现在看来，当时放弃的一些想法与其说是透明的 Visual Basic 6 生命期语义，还不如说是不错的资源管理技术。

第一种尝试是试图简化需要确定性结束的类的标记，实现方法是使用属性或者从一个“特别的”类来继承。这将导致对对象进行引用计算。人们也研究了很多不同的设计，这些设计中既包含了 System.Object 类的子类，也包含将类层次结构的根改变为其他另外的类，这样就可以将引用计算与非引用计算捆绑在一起了。遗憾的是，这种方案仍然存在许多的问题未能解决，我们将在下面的章节中对其进行介绍。

5.7.5.1 组成

当您使用一个要求确定性结束的对象而把它存储在不能确定性结束的对象中时，就会丢失这种确定性(因为确定性是不能传递的)。这个问题在类的层次结构中非常重要。比如数组，如果您想把能确定性结束的对象放进一个数组，那么该数组也需要能够确定性结束。还比如集合和散列表等等，在您意识到之前，整个类库就已经被引用计算了，因此就达不到确定性结束的目的。

有一种想法是将.NET 框架类库拆分为两部分(或者是分为两个分支)，使大多数类型的类都有两个版本。比如，确定性结束的数组和非确定性结束的数组。这种想法曾经被认真考虑过，但是，后来发现使用整个框架的两个副本会引起混淆，执行起来相当麻烦——每

个类都将被载入两个副本，因此这个方案最终很不实际。

针对特定类的特定解决方案也被仔细研究过，但没有一种方案能够在整个框架范围内成为一种通用的解决方案。想想这个最基本的问题——如果一个非确定性对象包含了对确定性对象的引用，系统仍然是不能确定性结束的——同样会发生错误，但如果做出“非确定性对象不能包含对确定性对象的引用”这样的约束的话，就很难编写真正的程序了。

5.7.5.2 转换

还有一个相关的问题涉及到转换。比如，我们是否可以将一个确定性对象转换为System.Object呢？如果能，它会被引用计算吗？答案如果是肯定的话，所有的一切就都会被引用计算；答案如果是否定的话，则该对象就失去了确定性；如果答案是“错误”的话，就违反了System.Object是对象层次结构的根这个基本前提。

5.7.5.3 接口

到目前为止，您应该能够感受到这个问题有多复杂了吧？如果一个确定性对象实现了接口，那么作为接口类型的引用也会被引用计算吗？如果答案是肯定的话，您就会引用计算实现接口的所有对象(注意System.Int32就实现了接口)；如果答案是否定的话，该对象又会失去确定性；如果答案是“错误”的话，确定性对象就不能实现接口了；如果答案是“它依赖于接口是否被标注上确定性”的话，就会产生另外一个分歧：接口并没有“义务”来表示出对象生命期。如果有人实现了一个采用ICollection接口的API，而您用来实现它的对象需要确定性结束，但是该接口又并不能确定性结束的话，您该怎么办？您当然就会“一筹莫展”了。这种情况下，就需要定义两个接口—— 一个具有确定性，一个不具有确定性——并且每个对象实现两次。信不信由您，这种自动产生两个版本方法的方式也考虑过。但是这种思路后来陷入巨大无比的复杂性之中，最后只好放弃。

5.7.6 Dispose设计模式

那么，我们应该如何解决上述问题呢？目前，确定性结束看起来是这样操作的：

- GC跟踪被引用的对象；
- GC在扫描对象以便确定该对象不再被引用时，扫描线程的优先级总是低；
- 第二个低优先级的线程负责清除，这个线程就是该对象的Finalize方法；

这种思路的确解决了循环引用的问题，但它又产生了另外的问题。比如，不能保证运行环境总能调用Finalize方法。这就是“确定性结束”这个术语中为什么强调“确定性”的意思。因此问题就转化为：“怎样才能进行必要的清除并且知道每次都调用了清除代码呢？”

在释放稀缺资源的情形下，Microsoft提出了基于Dispose设计模式的解决方案。这种

设计模式推荐对象公开一个公共方法，通常可以叫做“Cleanup”或“Dispose”，然后在用户结束使用该对象时来调用这个方法。这样，就由类的设计者负责在该方法中进行必要的清除操作。实际上，在.NET 框架类库中有很多类都为此实现了 Dispose 方法。比如，在有关 System.Winforms.TrayIcon 类的文档中就有这样的话：“当您结束使用 TrayIcon 时，请调用 Dispose”。

5.8 继　　承

在第 1 章中我们已经提到，当一个类构建于另一个类之上 (数据或者行为)时就要用到继承，并且遵循可替代性规则，也就是说，派生类可以替代基类。比如说您想构造一个数据库类的层次结构，假设您想编写一个类来处理 Microsoft SQL Server 数据库和 Oracle 数据库，由于这两个数据库在某些方面有区别，您想为每个数据库分别编写一个类。但是，这两个数据库又共享很多功能，您希望将这些公共的功能放进一个基类，然后从基类中派生出另外两个类，在需要的时候再覆盖或修改所继承的基类行为。

要从一个类中继承另一个类，您可以使用如下的语法：

class *<derivedClass> : <baseClass>*

下面是该数据库示例程序的代码：

```
using System;

class Database
{
    public Database()
    {
        CommonField = 42;
    }

    public int CommonField;

    public void CommonMethod()
    {
        Console.WriteLine("Database.Common Method");
    }
}

class SQLServer : Database
{
    public void SomeMethodSpecificToSQLServer()
    {
```

```
        Console.WriteLine("SQLServer.SomeMethodSpecificToSQLServer");
    }
}

class Oracle : Database
{
    public void SomeMethodSpecificToOracle()
    {
        Console.WriteLine("Oracle.SomeMethodSpecificToOracle");
    }
}

class InheritanceApp
{
    public static void Main()
    {
        SQLServer sqlserver = new SQLServer();

        sqlserver.SomeMethodSpecificToSQLServer();
        sqlserver.CommonMethod();
        Console.WriteLine("Inherited common field = {0}",
                          sqlserver.CommonField);
    }
}
```

编译并执行这个程序就会得到如下的输出：

```
SQLServer.SomeMethodSpecificToSQLServer
Database.Common Method
Inherited common field =42
```

注意 Database.CommonMethod 方法和 Database.CommonField 方法现在都是 SQLServer 类定义的一部分。因为 SQLServer 类和 Oracle 类都是从 Database 基类中派生而来的，因此它们都继承了基类中定义为“public”、“protected”或“internal”的几乎所有成员，惟一的例外是构造函数，因为它不能被继承。每个类都必须实现它自己的与其基类无关的构造函数。

第 6 章中将介绍方法覆盖。不过，为了本节的完整性，此处先简单介绍一下。方法覆盖就是允许您从基类中继承一个方法，然后改变这个方法的具体实现。抽象类与方法覆盖联系很紧密——这也将在第 6 章中介绍。

5.8.1 多接口

因为在很多新闻组和邮件列表中，对有关多继承的主题(比如：C#不支持通过派生的多继承)很有争议，因此我们来澄清一下。其实，通过实现多接口就可以将多个程序实体的行为特征聚集在一起。接口以及如何使用接口将在第9章中介绍。现在可以先将C#接口看作COM接口。

因此，下面这个程序就是非法的：

```
class Foo
{
}

class Bar
{
}

class MITest : Foo, Bar
{
    public static void Main ()
    {
    }
}
```

在这个示例中将出现的错误与您如何实现接口有关。您选定要实现的接口要列在该类的基类后面。因此，在这个例子中，C#编译器就认为Bar应当是一个接口类型，这也是C#编译器带给您下面这条错误信息的原因：

```
'Bar':type in interface list is not an interface
```

下面这个更实际的例子完全合法，因为MyFancyGrid是从Control中派生出来的，并且实现了ISerializable和IdataBound接口：

```
class Control
{
}

interface ISerializable
{
}

interface IDataBound
{
```

```
}

class MyFancyGrid : Control, ISerializable, IDataBound
{
}
```

这里的关键在于：在C#中实现所谓的多继承的惟一方式就是使用接口。

5.8.2 封装类

如果您希望确保某个类永远也不用作基类，在定义该类时就应该使用限定符“sealed”。定义封装类的惟一限制就是抽象类不能被定义为封装类，因为抽象类“天生”就是要被当作基类使用的。另外，尽管使用封装类的目的是阻止派生意图，但其实将类定义为封装类之后还会允许在运行的时候进行一定的优化。特别地，由于编译器保证了封装类永远也不会有任何派生类存在，因此就可以将封装类实例中的虚函数成员调用转变为非虚调用。下面是一个封装类的示例：

```
using System;
sealed class MyPoint
{
    public MyPoint(int x,int y)
    {
        this.x =x;
        this.y =y;
    }

    private int X;
    public int x
    {
        get
        {
            return this.X;
        }
        set
        {
            this.X =value;
        }
    }

    private int Y;
    public int y
    {
```

```
            get
            {
                return this.Y;
            }
            set
            {
                this.Y = value;
            }
        }
    }

    class SealedApp
    {
        public static void Main()
        {
            MyPoint pt = new MyPoint(6,16);
            Console.WriteLine("x = {0}, y = {1}", pt.x, pt.y);
        }
    }
```

注意内部类成员 X 和 Y 使用了访问限定符“private”，如果使用限定符“protected”就会使编译器发出警告信息，因为受保护的成员对于派生类而言是可见的，但现在我们已经知道，封装类不会有任何派生类。

5.9 本章小结

类的概念和类与对象的关系是基于对象编程思路的基础。构建于从 C++传递下来的继承之上的 C#的面向对象特征在.NET 框架中得到了进一步的加强。在类似公共语言运行环境(CLR)这样的受控系统中，资源管理是开发人员需要特别关注的地方。CLR 通过基于确定性结束的垃圾收集器来努力将程序员们从引用计算这样的“苦差事”中解放出来。C#中的继承处理也与 C++中不同。尽管 C#只支持单一继承，但开发人员可以通过实现多接口来获得一些多继承的好处。

第6章 方　　法

我们在第1章中已经学到，类就是封装的数据和加工数据的方法。换句话说，方法为类赋予了行为特征，而我们根据自己的意图使用希望类执行的操作来为方法命名。到目前为止，我们还没有涉及到在C#中有关定义和调用方法的更具体的问题。这也正是本章的目的——您将会学习“ref”和“out”这两个方法参数关键字，以及您该如何使用它们来定义方法，以便能返回多个值给调用程序。您还会学习如何定义重载方法，以便多个同名的方法根据传递给它们的不同类型和(或)参数值的个数而具有不同的功能。然后您将学到如何处理直到运行的时候才能知道方法参数值的确切个数的情形。最后，我们将以对虚拟方法的讨论(虚拟方法构建于第5章所讨论的继承之上) 以及如何定义静态方法作为本章的结束。

6.1 方法参数“ref”和“out”

当您在C#中通过使用方法来试图获取一些信息时，通常只能得到一个返回值。因此，初看起来就好像每个方法调用只能返回一个值。显然，在很多时候如果为每一个必要的数据都调用一个方法会相当麻烦。比如，假设您有一个Color类，用来代表一种给定的颜色，该颜色由三个值确定，这些值使用描述颜色的标准RGB(red-green-blue，红-绿-蓝)模式来给出。如果只使用一个返回值，您就不得不书写如下的代码来得到这三个值：

```
// Assuming color is an instance of a Color class.
int red = color.GetRed();
int green = color.GetGreen();
int blue = color.GetBlue();
```

但是我们希望的却是类似于这样的代码：

```
int red;
int green;
int blue;
color.GetRGB(red,green,blue);
```

不过，这里有个问题。当调用color.GetRGB方法时，参数值red、green和blue的值会被复制到该方法的局部堆栈中，该方法对这些变量的任何修改都不会传递给调用程序的变量。

在C++中，这个问题是这样解决的：调用程序将指针或引用传递给变量，这样，被调用的方法就能加工调用程序的数据了。在C#中的解决方案与此类似。事实上，C#提供了两种相似的解决方案。第一种包含关键字“ref”，这个关键字告诉C#编译器被传递的参数值指向与调用代码中变量相同的内存。这样，如果被调用的方法修改了这些值然后返回的话，调用代码的变量也就被修改了。下面的程序说明了如何在Color类中使用关键字“ref”：

```
using System;

class Color
{
   public Color()
   {
      this.red = 255;
      this.green = 0;
      this.blue = 125;
   }

   protected int red;
   protected int green;
   protected int blue;

   public void GetColors(ref int red, ref int green, ref int blue)
   {
      red = this.red;
      green = this.green;
      blue = this.blue;
   }
}

class RefTest1App
{
   public static void Main()
   {
      Color color = new Color();
      int red;
      int green;
      int blue;
      color.GetColors(ref red, ref green, ref blue);
      Console.WriteLine("red = {0}, green = {1}, blue = {2}",
red, green, blue);
   }
}
```

使用关键字“ref”有一个缺点，实际上正是由于这个限制，以上的代码不会通过编译。

当您使用关键字“ref”时，您必须在调用方法之前初始化被传递的参数值。因此，要想让上面的代码能够正常运行，必须做出如下修改：

```
using System;

class Color
{
    public Color()
    {
        this.red = 255;
        this.green = 0;
        this.blue = 125;
    }

    protected int red;
    protected int green;
    protected int blue;

    public void GetColors(ref int red, ref int green, ref int blue)
    {
        red = this.red;
        green = this.green;
        blue = this.blue;
    }
}

class RefTest2App
{
    public static void Main()
    {
        Color color = new Color();
        int red = 0;
        int green = 0;
        int blue = 0;
        color.GetColors(ref red, ref green, ref blue);
        Console.WriteLine("red = {0}, green = {1}, blue = {2}",
red, green, blue);
    }
}
```

在这个例子中，初始化这些就要被重写的变量看起来好像是白费功夫。因此，C#提供了另外一种途径来传递这种参数值——调用代码需要看到它们的值的变化：关键字“out”。下面是同样的 Color 类示例程序，只不过使用的是关键字“out”：

```
using System;

class Color
{
    public Color()
    {
        this.red = 255;
        this.green = 0;
        this.blue = 125;
    }

    protected int red;
    protected int green;
    protected int blue;

    public void GetColors(out int red, out int green, out int blue)
    {
        red = this.red;
        green = this.green;
        blue = this.blue;
    }
}

class OutTest1App
{
    public static void Main()
    {
        Color color = new Color();
        int red;
        int green;
        int blue;
        color.GetColors(out red, out green, out blue);
        Console.WriteLine("red = {0}, green = {1}, blue = {2}",
                  red, green, blue);
    }
}
```

关键字“ref”和“out”之间的惟一区别就是关键字“out”不要求调用代码初始化要传递的参数值。那么，关键字“ref”什么时候使用呢？当您需要确保调用方法已经初始化参数值的时候，您就应该使用关键字“ref”。在上面的例子中，能够使用“out”是因为被调用的方法并不依赖于被传递的变量的值。但如果被调用的方法要使用参数值的值呢？看看下面这段代码：

```
using System;
```

```
class Window
{
    public Window(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    protected int x;
    protected int y;

    public void Move(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void ChangePos(ref int x, ref int y)
    {
        this.x += x;;
        this.y += y;

        x = this.x;
        y = this.y;
    }
}

class OutTest2App

{
    public static void Main()
    {
        Window wnd = new Window(5, 5);

        int x = 5;
        int y = 5;

        wnd.ChangePos(ref x, ref y);
        Console.WriteLine("{0}, {1}", x, y);

        x = -1;
        y = -1;
        wnd.ChangePos(ref x, ref y);
        Console.WriteLine("{0}, {1}", x, y);
```

```
        }
    }
```

从中可以看出，被调用的方法 Window.ChangePos 根据被传递的值来进行操作。这种情况下，关键字“ref”强制调用代码要初始化这些值，以便被调用的方法能够正确地工作。

6.2 方法重载

方法重载允许 C#程序员可以多次使用同名的方法而每次传递的参数值不同。这一点在两种情形下特别有用：第一种情形是您希望公开一个方法名，该方法的外部表现根据传递的参数值的类型不同而稍微有些不同。比如，假设您有一个日志类，允许您的应用程序将诊断信息写入磁盘。要想把这个类变得更灵活一些，您可能会有几个不同形式的 Write 方法用于指定要写入的信息。除了接受要写入的字符串之外，该方法可能还需要接受一个字符串源的标识符。如果没有重载方法，您就不得不为每种情形分别实现一个方法，如 WriteString 和 WriteFromResourceId。不过，通过方法重载，您就可以实现以下的方法——都叫做“WriteEntry” ——每个方法只是参数类型不同而已：

```
using System;

class Log
{
    public Log(string fileName)
    {
        // Open fileName and seek to end.
    }

    public void WriteEntry(string entry)
    {
        Console.WriteLine(entry);
    }

    public void WriteEntry(int resourceId)
    {
        Console.WriteLine
       ("Retrieve string using resource id and write to log");
    }
}

class Overloading1App
{
```

```
    public static void Main()
    {
        Log log = new Log("My File");
        log.WriteEntry("Entry one");
        log.WriteEntry(42);
    }
}
```

第二种方法重载有用的情形是当您使用构造函数的时候，构造函数是实例化对象时要调用的基本方法。假设您希望创建一个类，这个类可以用多种方式来构造——比如，采用文件句柄(int)或者文件名(string)来打开一个文件。因为 C#规则规定了类的构造函数必须与类同名，因此您不能简单地为每一种不同的变量类型创建不同名称的方法。此时，您应该重载构造函数：

```
using System;

class File
{
}

class CommaDelimitedFile
{
    public CommaDelimitedFile(String fileName)
    {
        Console.WriteLine("Constructed with a file name");
    }

    public CommaDelimitedFile(File file)
    {
        Console.WriteLine("Constructed with a file object");
    }
}

class Overloading2App
{
    public static void Main()
    {
        File file = new File();
        CommaDelimitedFile file2 = new CommaDelimitedFile(file);
        CommaDelimitedFile file3 =
new CommaDelimitedFile("Some file name");
    }
}
```

关于方法重载要记住的重要一点是：每个方法的参数值列表必须有所不同。因此，下面的代码不会通过编译，因为 Overloading3App.Foo 方法的两个版本之间只有返回类型不同而已：

```
using System;

class Overloading3App
{
    void Foo(double input)
    {
        Console.WriteLine("Overloading3App.Foo(double)");
    }

    // ERROR: Only differs by return value. Won't compile.
    double Foo(double input)
    {
        Console.WriteLine("Overloading3App.Foo(double)
                          (second version)");
    }

    public static void Main()
    {
        Overloading3App app = new Overloading3App();

        double i = 5;
        app.Foo(i);
    }
}
```

6.3 可变的方法参数

有时候，传递给方法的参数值的个数要到运行的时候才能确定下来。比如，如果您希望一个类根据一系列 x 坐标和 y 坐标来画线，这个类公开了一个方法，这个方法采用代表 x 和 y 的值的对象 Point 作为单个参数值。然后这个方法将每个 Point 对象都存储在一个链表或数组成员中，直到调用代码希望打印出整个的点序列。不过，这种设计方案相当笨拙。首先，它要求用户在画这条线的每个点时都执行不必要的方法调用——如果这条线很长，这样做就更乏味——然后再调用另一个方法来画线。其次，它要求类将这些点以一定的方式存储起来，而需要这些点的惟一理由就是要满足这一个方法 DrawLine 的需要。可变参数值解决起这类问题来得心应手。

使用关键字“params”并在方法的参数值列表中指定一个数组就可以指定可变数目的方法参数。下面这个例子就是C#中的Draw类，它允许用户只调用一次带有不定数量的Point对象的方法，就可以打印出每个点来：

```
using System;

class Point
{
   public Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   public int x;
   public int y;
}

class Chart
{
   public void DrawLine(params Point[] p)
   {
      Console.WriteLine("\nThis method would print a line " +
                        "along the following points:");
      for (int i = 0; i < p.GetLength(0); i++)

      {
         Console.WriteLine("{0}, {1}", p[i].x, p[i].y);
      }
   }
}

class VarArgsApp
{
   public static void Main()
   {
      Point p1 = new Point(5,10);
      Point p2 = new Point(5, 15);
      Point p3 = new Point(5, 20);

      Chart chart = new Chart();
      chart.DrawLine(p1, p2, p3);
   }
}
```

方法 DrawLine 告诉 C#编译器它可以使用不定数量的 Point 对象。运行的时候，该方法使用一个简单的 for 循环就访问了传递给它的所有 Point 对象，并将它们逐个打印出来。

注意在真正的应用程序中，使用属性来访问 Point 对象的 x 和 y 成员要比将这些成员公开好得多。此外，在 DrawLine 方法中 for 循环处使用 foreach 语句也更好一些。不过，为了本书的连续性，现在还不能提出这些语言特性。在第 7 章中将介绍属性，在第 11 章中将介绍 foreach 语句。

6.4 虚拟方法

在第 5 章中我们已经看到，可以从一个类中派生出另外一个类，这样类就可以继承并在已存在的类的功能之上构建新的功能。因为那时我们还没有讨论到方法，因此只仅仅接触到了字段和方法的继承。也就是说，我们并没有研究在派生类中修改基类行为的方法。这是通过使用虚拟方法来实现的，这也正是本节的主题所在。

6.4.1 方法覆盖

首先我们来看看如何在一个继承的方法中覆盖基类的功能。还是以代表雇员的基类开始，为了使这个示例程序尽可能简单，我们只给这个基类一个方法 CalculatePay，而方法内部除了让我们知道被调用的方法的名称外，别的什么也不做。这将有助于我们在稍后确定继承树中的哪个方法被调用：

```
class Employee
{
    public void CalculatePay()
    {
        Console.WriteLine("Employee.CalculatePay()");
    }
}
```

现在，假设您想要从 Employee 类中派生一个类，并希望覆盖掉 CalculatePay 方法，以便针对派生类进行一些具体的操作。要想达到这样的目的，您需要在派生类的方法定义中使用关键字“new”。下面的代码显示了其简单性：

```
using System;

class Employee
{
    public void CalculatePay()
```

```
        {
            Console.WriteLine("Employee.CalculatePay()");
        }
    }

    class SalariedEmployee : Employee
    {
        // The new keyword enables you to override
        // base class' implementation.
        new public void CalculatePay()
        {
            Console.WriteLine("SalariedEmployee.CalculatePay()");
        }
    }

    class Poly1App
    {
        public static void Main()
        {
            Poly1App poly1 = new Poly1App();

            Employee baseE = new Employee();
            baseE.CalculatePay();

            SalariedEmployee s = new SalariedEmployee();
            s.CalculatePay();
        }
    }
```

编译并执行这个程序将产生如下输出：

```
c:\>Poly1App
Employee.CalculatePay()
Salaried.CalculatePay()
```

6.4.2 多态性

引用派生对象的时候，利用关键字“New”便可以实现方法的覆盖；那么，如果引用的是基类，如何让编译器调用派生类的方法呢？这就是多态性要解决的问题。多态性允许在类的层次结构中多次定义同一个方法，并使运行环境根据要使用的对象来调用该方法的适当版本。

仍以前面的雇员程序为例来说明。应用程序 Poly1App 能够正常运行，因为我们有两个

对象：一个 Employee 对象和一个 SalariedEmployee 对象。在更接近实际的应用程序中，我们可能会从一个数据库中读出所有的雇员记录并形成一个数组。尽管这些雇员中有些是合同工，有些是正式工，但我们仍然需要将它们都以相同的类型——基类类型 Employee——存放在数组中。这样，在遍历这个数组、获取对象并调用每个对象的 CalculatePay 方法时，我们希望编译器能够正确调用每个对象对 CalculatePay 方法的实现。

在下面的例子中添加了一个新类——ContractEmployee。主要的应用程序类现在包含一个 Employee 类型的对象数组和两个方法——LoadEmployees 将 Employee 对象装载到数组中，DoPayroll 遍历这个数组，调用每个对象的 CalculatePay 方法。

```
using System;

class Employee
{
    public Employee(string name)
    {
        this.Name = name;
    }

    protected string Name;

    public string name

    {
        get
        {
            return this.Name;
        }
    }

    public void CalculatePay()
    {
        Console.WriteLine("Employee.CalculatePay called for {0}",
                          name);
    }
}

class ContractEmployee : Employee
{
    public ContractEmployee(string name)
    : base(name)
    {
    }
```

```
        public new void CalculatePay()
        {
            Console.WriteLine("ContractEmployee.CalculatePay  called  for
{0}",
                              name);
        }
    }

    class SalariedEmployee : Employee
    {
        public SalariedEmployee (string name)
        : base(name)
        {
        }

        public new void CalculatePay()
        {
            Console.WriteLine("SalariedEmployee.CalculatePay  called  for
{0}",
                              name);
        }
    }

    class Poly2App
    {
        protected Employee[] employees;

        public void LoadEmployees()
        {
            // Simulating loading from a database.
            employees = new Employee[2];
            employees[0] = new ContractEmployee("Kate Dresen");
            employees[1] = new SalariedEmployee("Megan Sherman");
        }

        public void DoPayroll()
        {
            foreach(Employee emp in employees)
            {
                emp.CalculatePay();
            }
        }
```

```
    public static void Main()
    {
        Poly2App poly2 = new Poly2App();
        poly2.LoadEmployees();
        poly2.DoPayroll();
    }
}
```

不过，运行这个程序会产生如下输出结果：

```
c:\>Poly2App
Employee.CalculatePay called for Kate Dresen
Employee.CalculatePay called for Megan Sherman
```

显然，这并不是我们想要的结果——每个对象都调用了基类对 CalculatePay 方法的实现。这里所发生的就是所谓“早期捆绑”（early binding）现象的一个例子。当编译代码的时候，C#编译器一看到对 emp.CalculatePay 的调用，就确定了调用发生时在内存中它应跳转的地址。这种情况下，采用的当然就是 Employee.CalculatPay 方法的内存地址了。

看看下面从 Poly2App 应用程序产生的 MSIL 代码，特别注意 IL_0014 行上显式调用了 Employee.CalculatePay 方法：

```
.method public hidebysig instance void DoPayroll() il managed
{
  // Code size       34 (0x22)
  .maxstack  2
  .locals (class Employee V_0,
          class Employee[] V_1,
          int32 V_2,
          int32 V_3)

  IL_0000:  ldarg.0
  IL_0001:  ldfld     class Employee[] Poly2App::employees
  IL_0006:  stloc.1
  IL_0007:  ldloc.1
  IL_0008:  ldlen
  IL_0009:  conv.i4
  IL_000a:  stloc.2
  IL_000b:  ldc.i4.0
  IL_000c:  stloc.3
  IL_000d:  br.s      IL_001d
  IL_000f:  ldloc.1
  IL_0010:  ldloc.3
  IL_0011:  ldelem.ref
  IL_0012:  stloc.0
```

```
  IL_0013:  ldloc.0
  IL_0014:  call      instance void Employee::CalculatePay()
  IL_0019:  ldloc.3
  IL_001a:  ldc.i4.1
  IL_001b:  add
  IL_001c:  stloc.3
  IL_001d:  ldloc.3
  IL_001e:  ldloc.2
  IL_001f:  blt.s     IL_000f
  IL_0021:  ret
} // end of method Poly2App::DoPayroll
```

对 Employee.CalculatePay 方法的调用就是问题所在。我们希望的是发生“后期捆绑”（late binding），也就是说编译器直到运行的时候才选择要调用的方法。要强制编译器调用向上转换了的对象的方法的正确版本，可以使用两个新的关键字：“virtual”和“override”。关键字“virtual”必须用于基类方法，关键字“override”用于派生类对该方法的实现。下面更改过的代码这次就能正常工作了：

```
using System;

class Employee
{
    public Employee(string name)
    {
        this.Name = name;
    }

    protected string Name;
    public string name

    {
        get
        {
            return this.Name;
        }
    }

    virtual public void CalculatePay()
    {
        Console.WriteLine("Employee.CalculatePay called for {0}",
                          name);
    }
}
```

```
    class ContractEmployee : Employee
    {
        public ContractEmployee(string name)
        : base(name)
        {
        }
        override public void CalculatePay()
        {
            Console.WriteLine("ContractEmployee.CalculatePay  called  for
{0}",
                              name);
        }
    }

    class SalariedEmployee : Employee
    {
        public SalariedEmployee (string name)
        : base(name)
        {
        }
        override public void CalculatePay()
        {
            Console.WriteLine("SalariedEmployee.CalculatePay  called  for
{0}",
                              name);
        }
    }

    class Poly3App
    {
        protected Employee[] employees;
        public void LoadEmployees()
        {
            // Simulating loading from a database.
            employees = new Employee[2];

            employees[0] = new ContractEmployee("Kate Dresen");
            employees[1] = new SalariedEmployee("Megan Sherman");
        }

        public void DoPayroll()
        {
            foreach(Employee emp in employees)
            {
```

```
            emp.CalculatePay();
        }
    }

    public static void Main()
    {
        Poly3App poly3 = new Poly3App();
        poly3.LoadEmployees();
        poly3.DoPayroll();
    }
}
```

在运行这个程序之前，先看看产生的 IL 代码。这次要注意 IL_0014 行上使用了 MSIL 代码“callvirt”，它告诉编译器真正要调用的方法要到运行的时候才能知道，因为它取决于使用的是哪个派生对象。

```
.method public hidebysig instance void DoPayroll() il managed
{
  // Code size       34 (0x22)
  .maxstack  2
  .locals (class Employee V_0,
           class Employee[] V_1,
           int32 V_2,
           int32 V_3)
  IL_0000:  ldarg.0
  IL_0001:  ldfld     class Employee[] Poly3App::employees
  IL_0006:  stloc.1
  IL_0007:  ldloc.1
  IL_0008:  ldlen
  IL_0009:  conv.i4
  IL_000a:  stloc.2
  IL_000b:  ldc.i4.0
  IL_000c:  stloc.3
  IL_000d:  br.s      IL_001d
  IL_000f:  ldloc.1
  IL_0010:  ldloc.3
  IL_0011:  ldelem.ref
  IL_0012:  stloc.0
  IL_0013:  ldloc.0
  IL_0014:  callvirt   instance void Employee::CalculatePay()
  IL_0019:  ldloc.3
  IL_001a:  ldc.i4.1
  IL_001b:  add
  IL_001c:  stloc.3
```

```
    IL_001d:  ldloc.3
    IL_001e:  ldloc.2
    IL_001f:  blt.s     IL_000f
    IL_0021:  ret
  } // end of method Poly3App::DoPayroll
```

这时再运行这个程序就会产生下面这样的结果了：

```
c:\>Poly3App
ContractEmployee.CalculatePay called for Kate Dresen
SalariedEmployee.CalculatePay called for Megan Sherman
```

注意： 虚拟方法不能被声明为私有(private)，因为私有成员对于派生类来说是不可见的。

6.5 静态方法

静态方法就是作为一个整体存在于类中的方法，而不是存在于类的一个特定实例中。与别的静态成员相比，静态方法的关键好处就在于它们虽然与类的特定实例分离，但却并没有影响应用程序的全局空间，也不因为没有与类实例相连而有损面向对象的风格。作者曾编写过一个C#的数据库应用编程接口，在这个类层次结构中，有一个SQLServerDb类带有一些基本的NURD(new, update, read and delete)功能，它也公开了一个修复数据库的方法Repair。在这个 Repair 方法中，并不需要打开数据库。实际上，作者使用的 ODBC 函数SQLConfigDataSource 要求在对该数据库进行操作时它必须被关闭。不过，SQLServerDb的构造函数还是打开了由传递给它的数据库名称所指定的数据库。这种情况下最好使用静态方法，这样就可以将一个方法放在它所属的SQLServerDb类中，而不用借助于类的构造函数了。显然，对于客户来说，好处就是他们也无需对SQLServerDb类进行实例化。在下一个例子中，您可以看到一个静态方法RepairDatabase被从Main方法中调用。注意我们并没有创建一个SQLServerDb的实例：

```
using System;

class SQLServerDb
{
    // Bunch of other nonsalient members.
    public static void RepairDatabase()
    {
        Console.WriteLine("repairing database...");
    }
```

```
}

class StaticMethod1App
{
    public static void Main()
    {
        SQLServerDb.RepairDatabase();
    }
}
```

使用关键字“static”就可以将方法定义为静态方法了。用户使用 Class.Method 这样的语法来调用静态方法。即使用户引用了类的实例，仍然需要使用这种语法格式。为了说明这一点，下面的代码将通不过编译：

```
// This code will fail to compile.
using System;

class SQLServerDb
{
    // Bunch of other nonsalient members.
    public static void RepairDatabase()
    {
        Console.WriteLine("repairing database...");
    }
}

class StaticMethod2App
{
    public static void Main()
    {
        SQLServerDb db = new SQLServerDb();
        db.RepairDatabase();
    }
}
```

访问类成员

关于静态方法的最后一点就是它能访问哪些类成员。您也许已经想到，静态方法可以访问类的任何静态成员，但是不能访问实例成员。下面的例子将说明这一点：

```
using System;

class SQLServerDb
```

```
{
    static string progressString1 = "repairing database...";
    string progressString2 = "repairing database...";

    public static void RepairDatabase()
    {
        Console.WriteLine(progressString1); // This will work.
        Console.WriteLine(progressString2); // Fails compilation.
    }
}

class StaticMethod3App
{
    public static void Main()
    {
        SQLServerDb.RepairDatabase();
    }
}
```

6.6 本章小结

方法赋予类以行为特征并且按照我们的意愿执行操作。C#中的方法很有弹性，允许返回多个值、重载以及可变参数等。关键字“ref”和“out”允许方法返回一个以上的值给调用程序。重载允许多个同名的方法按照传递给它们的参数值类型和个数的不同而具有不同的功能。方法还可以带可变参数，关键字“params”允许直到运行的时候才知道参数值个数的方法存在。虚拟方法允许在继承的类中控制如何修改方法。最后，关键字“static”允许方法作为类的一部分存在，而不是作为对象的一部分存在。

第 7 章　属性、数组和索引器

到目前为止，我们已经了解了 C#支持的基本类型以及在类和应用程序中如何声明和使用它们。本章将打破每章只介绍一种语言要素的模式，而同时介绍属性、数组和索引器这三种语言要素，因为它们之间存在某些共同的地方。它们允许 C#类开发人员扩展基本的类/字段/方法结构，以便展示出更加自然的类成员接口。

7.1　属性——智能字段

设计类时的一个好的目标总是不仅仅隐藏类成员的实现，而且禁止任何对类字段成员的直接访问。通过“存取器方法”——其职责就是获取和设置字段的值，您就可以确保字段可以被正确处理，也就是说，根据您特定的问题域规则而执行必要的操作处理。

比如，假设您有一个地址类 Address 包含一个邮政编码字段 ZipCode 和一个城市字段 City。当客户修改字段 Address.ZipCode 时，您希望通过一个数据库来验证邮政编码是否有效，并且根据邮政编码自动设置 Address.City 字段的值。如果客户可以直接访问一个公共的 Address.ZipCode 成员的话，上面这两个任务就有点难了，因为直接更改成员变量并不需要方法。因此，除了可以直接访问 Address.ZipCode 字段之外，更好的解决方案是将 Address.ZipCode 和 Address.City 这两个字段定义为“protected”，然后提供存取器方法来获取和设置 Address.ZipCode 字段的值。这样，您就可以附加一些代码来执行需要的操作了。

这个邮政编码的程序在 C#中的示例代码如下。注意真正的 ZipCode 字段被定义为“protected”，因此客户不能直接访问它；而存取器方法 GetZipCode 和 SetZipCode 被定义为“public”：

```
class Address
    {
        protected string ZipCode;
        protected string City;
        public string GetZipCode()
        {
            return this.ZipCode;
        }
        public void SetZipCode(string ZipCode)
```

```
        {
            // Validate ZipCode against some datastore.
            this.ZipCode = ZipCode;
            // Update this.City based on validated ZIP code.
        }
    }
```

客户可以这样访问 Address.ZipCode 的值：

```
Address addr = new Address();
addr.SetZipCode("55555 ");
string zip =addr.GetZipCode();
```

7.1.1 定义和使用属性

使用存取器方法固然不错，这种技术也被不少面向对象语言(包括 C++和 Java)的程序员们所采用，不过，C#提供了一种更加理想的机制——属性，它具有与存取器方法相同的功能，在客户端却表现更佳。通过属性，程序员们不必借助于存取器方法就可以让客户能够像访问公共字段一样来访问类的字段。

C#属性由一个字段声明和用于修改该字段值的存取器方法组成。这些存取器方法叫做“getter”和“setter”方法。getter 方法用于获取字段的值，而 setter 方法用于修改字段的值。

下面是使用 C#字段重写的前面的代码：

```
class Address
    {
        protected string city;
        protected string zipCode;

        public string ZipCode
        {
            get
            {
                return zipCode;
            }
            set
            {
                // Validate value against some datastore.
                zipCode = value;
                // Update city based on validated zipCode.
            }
        }
    }
```

注意此处创建了一个叫做“Address.zipCode”的字段和一个叫做“Address.ZipCode”的属性。有些人可能一开始会混淆，以为 Address.ZipCode 也是字段，不明白为什么一个字段要定义两次。但 Address.ZipCode 不是字段，而是属性，它其实就是一种定义类成员存取器的通用方式，以便可以使用更加自然的“object.field”语法格式。如果在这个例子中省略掉 Address.zipCode 字段，在 setter 语句中将 zipCode = value 改为 ZipCode = value 的话，就会导致 setter 方法被无限调用。同样要注意 setter 并没有任何参数值，被传递的值被自动放在一个叫做“value”的变量中，在 setter 方法内可以访问 value 变量(很快您就会在 MSIL 中看到这个“奇迹”是怎样发生的)。

既然我们已经写出了 Address.ZipCode 属性，就来看一看客户代码需要做出的改动吧：

```
Address addr = new Address();
addr.ZipCode = "55555";
string zip = addr.ZipCode;
```

这样，客户访问字段就很自然了：无需猜想或到文档(即源代码)中查找以便决定这个字段是否是公共的，如果不是，存取器方法的名称又是什么。

7.1.2 编译器的工作原理

那么，编译器是如何允许我们以标准的 object.field 语法格式来调用一个方法的呢？同样，value 变量又是从哪里来的呢？要回答这些问题，我们需要看一看编译器产生的 MSIL 代码，首先看一看属性的 getter 方法。

在目前的示例程序中，我们定义了下面这个 getter 方法：

```
class Address
{
    protected string city;
    protected string zipCode;
    public string ZipCode
    {
        get
        {
            return zipCode;
        }
        ...
}
...
}
```

从这个方法的 MSIL 代码中可以看到，编译器创建了一个叫做“get_ZipCode”的存取

器方法：

```
.method public hidebysig specialname instance string
        get_ZipCode() cil managed
{
  // Code size       11 (0xb)
  .maxstack  1
  .locals ([0] string _Vb_t_$00000003$00000000)
  IL_0000:  ldarg.0
  IL_0001:  ldfld     string Address::zipCode
  IL_0006:  stloc.0
  IL_0007:  br.s      IL_0009
  IL_0009:  ldloc.0
  IL_000a:  ret
} // end of method Address::get_ZipCode
```

我们之所以能够识别出存取器方法的名称，是因为编译器在属性名前加上了“get_”前缀(为 getter 方法)和“set_”前缀(为 setter 方法)。这样，下面的代码就表示调用 get_ZipCode：

```
String str =addr.ZipCode; // this calls Address::get_ZipCode
```

您可能会尝试以下对存取器方法的显式调用：

```
String str =addr.get_ZipCode; // **ERROR-Won't compile
```

但是，这句代码不会通过编译，因为显式调用内部的 MSIL 方法是非法的。

我们的问题是：“编译器是怎样允许我们使用标准语法格式 object.field 并调用方法的呢？”答案就是编译器在解析C#属性语法的时候，实际上为我们产生了适当的 getter 和 setter 方法。这样，出现 Address.ZipCode 属性的时候，编译器就产生了包含 get_ZipCode 和 set_ZipCode 方法的 MSIL 代码。

下面看看产生的 setter 方法。在 Address 类中，您会看到如下代码：

```
public string ZipCode
{
    ...
    set
      {
          // Validate value against some datastore.
          zipCode = value;
         // Update city based on validated zipCode.
      }
   }
```

注意在这段代码中根本没有声明一个名为“value”的变量，但我们仍然可以使用这个变量来存储调用程序传递过来的值，并且设置受保护的成员字段 zipCode。当 C#编译器为

setter 方法产生 MSIL 代码时，它将插入这个变量作为 set_ZipCode 方法的一个参数值。

在产生的 MSIL 代码中，set_ZipCode 方法将一个字符串变量作为参数值：

```
.method public hidebysig specialname instance void
        set_ZipCode(string 'value') cil managed
{
  // Code size       8 (0x8)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  stfld     string Address::zipCode
  IL_0007:  ret
} // end of method Address::set_ZipCode
```

尽管在 C#源代码中看不到这个方法，但是在设置 ZipCode 属性的时候，如 addr.ZipCode("12345")，就会产生一段 MSIL 代码来调用 Address::set_ZipCode("12345")。但如果在 C#中直接调用 set_ZipCode 这个方法就会产生错误。

7.1.3 只读属性

在前面的例子中，Address.ZipCode 属性可以认为是可读/写的，因为定义了 getter 和 setter 这两个方法。当然，有时候您可能不希望客户代码设置给定字段的值，这时候您可以将这个字段设为只读字段，将 setter 方法省略掉即可。为了进一步说明只读字段，我们现在限制客户代码设置 Address.city 字段，只留下 Address.ZipCode 属性作为更改这个字段值的惟一途径：

```
class Address
    {
        protected string city;
        public string City
        {
            get
            {
                return city;
            }
        }

        protected string zipCode;
        public string ZipCode
        {
            get
            {
```

```
            return zipCode;
        }
        set
        {
            // Validate value against some datastore.
            zipCode = value;
            // Update city based on validated zipCode.
        }
    }
}
```

7.1.4 继承属性

与方法一样，属性也可以使用第6章中介绍过的“virtual”、“override”或“abstract”限定符。这些限定符允许派生类继承并覆盖属性，与它处理来自基类的任何其他成员一样。关键在于您只能在属性这个级别上指定这些限定符。也就是说，如果同时存在 getter 方法和 setter 方法的话，如果您覆盖其中一个，您必须也覆盖另外的一个。

7.1.5 属性的高级使用

到目前为止，我们谈论到属性的用途时有以下方面：

- 它们为客户代码提供了一定层次上的抽象。
- 通过 object.field 语法格式，它们提供了一种访问类成员的通用方式。
- 它们允许类在修改或访问一个特定的字段时，还可以进行其他的操作。

上面第三点给我们指出了使用属性的另一个“用武之地”：实现所谓“偷懒的初始化”(lazy initialization)。这是一种优化技术，能够在需要类成员的时候才对它们进行初始化。

如果类中包含了很少被引用的成员，而这些成员的初始化又会占用大量的时间和系统资源的话，比如数据必须从数据库或拥挤的网络读入的时候，这时使用“偷懒的初始化”就很有用了。因为您知道这些成员不常被引用而对它们进行初始化的代价又很大，所以可以在调用它们的 getter 方法时再对它们进行初始化。为了说明这一点，假设您有一个库存应用程序，销售人员会在他们的笔记本电脑中运行，以便存入顾客的订单，但他们很少会使用这个程序来检查库存水平。通过属性的使用，您就可以允许相关的类初始化时不必读入库存记录，如下面的代码所示。这样，如果销售人员的确需要访问某项货物的库存记录时，这时 getter 方法才访问远程数据库。

```
class Sku
    {
```

```
        protected double onHand;

        public string OnHand
        {
            get
            {
                // Read from central db and set onHand value.
                return onHand;
            }
        }
    }
```

本章到目前为止，您可能已经看到，属性通过为字段提供存取器方法，从而为客户代码提供了更通用和自然的接口。正因为此，属性有时候也叫做“智能字段”。现在，我们更进一步来看一看在 C#中是如何定义和使用数组的，以及属性如何以“索引器”的形式与数组共用。

7.2 数　　组

到目前为止，本书中的大多数示例程序展示的都是如何定义有限个数、且预先能确定个数的变量。但是在实际的应用程序中，有时候您可能会直到运行的时候才知道所需对象的确切数目。比如，您开发了一个编辑器，希望跟踪添加到一个对话框里的控件，但编辑器将显示出来的控件的确切数目要到运行的时候才知道。这时，您就可以使用一个数组来存储并跟踪一组动态分配的对象——这个例子中即编辑器中的控件。

在 C#中，数组对象共同的基类是 System.Array。因此，虽然在 C#中定义数组的语法格式与 C++或 Java 中类似，但定义数组实际上是实例化了一个.NET 类，也就是说，声明的每一个数组都从 System.Array 中继承了相同的成员。本节我们将介绍如何定义和实例化数组、如何使用不同的数组类型、如何遍历数组的每一个元素以及 System.Array 类的一些经常使用的属性和方法。

7.2.1 声明数组

在 C#中声明一个数组时，只需将空的方括号放置在变量的类型和名称之间，如：

```
int[] numbers;
```

注意这种语法格式与 C++中稍微有所不同，在 C++中括号是在变量名之后。因为数组是基于类的，所以声明类的很多规则也可以应用于声明数组。比如，当您声明一个数组时，

实际上并没有创建那个数组。与使用类一样，在分配数组的元素之前，您必须实例化该数组。在下面的例子中，在声明数组的同时实例化了数组：

```
// Declares and instantiates a single dimensional array of 6 integers
int [] numbers = new int [6];
```

不过，在将数组声明为类的一个成员时，您必须将声明数组与实例化数组分成两个不同的步骤，因为直到运行的时候您才能实例化对象：

```
class YourClass
{
   ...
int[] numbers;
...

   void SomeInitMethod()
{
   ...
      numbers = new int[6];
      ...
   }
}
```

7.2.2 一维数组

下面这个例子将一个一维数组声明为一个类成员，在构造函数中实例化并填充该数组，然后使用了一个 for 循环语句来遍历这个数组，打印出每一个元素：

```
using System;

class SingleDimArrayApp
{
   protected int[] numbers;

   SingleDimArrayApp()
   {
      numbers = new int[6];
      for (int i = 0; i < 6; i++)
      {
         numbers[i] = i * i;
      }
   }
```

```
    protected void PrintArray()
    {
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine("numbers[{0}]={1}", i, numbers[i]);
        }
    }

    public static void Main()
    {
        SingleDimArrayApp app = new SingleDimArrayApp();
        app.PrintArray();
    }
}
```

编译并运行这个程序将产生如下输出结果：

```
numbers [0]=0
numbers [1]=1
numbers [2]=4
numbers [3]=9
numbers [4]=16
numbers [5]=25
```

在这个例子中，SingleDimArray.PrintArray 方法使用了 System.Array.Length 属性来确定该数组中的元素个数。由于这里只是一个一维数组，看起来不是很明显，实际上 Length 属性返回的是数组所有维的所有元素的个数。因此，如果是一个 5×4 的两维数组，Length 属性将返回 20。下一节我们将学习多维数组以及如何确定一个特定数组维的上界。

7.2.3 多维数组

C#除了一维数组以外，还支持多维数组，多维数组的每一维由一个逗号分隔。下面的代码声明了一个 double 类型的三维数组：

```
Double[ , , ] numbers;
```

要确定一个已声明 C#数组的维数，只需将逗号的个数加 1 即可。

下面的例子中使用了一个两维数组的销售数字来代表今年当月的销售总量与去年同期的销售总量，要特别注意实例化该数组所用的语法格式(在 MultiDimArrayApp 构造函数中)：

```
using System;

class MultiDimArrayApp
{
```

```
        protected int currentMonth;
        protected double[,] sales;

        MultiDimArrayApp()

        {
            currentMonth=10;

            sales = new double[2, currentMonth];
            for (int i = 0; i < sales.GetLength(0); i++)
            {
                for (int j=0; j < 10; j++)
                {
                    sales[i,j] = (i * 100) + j;
                }
            }
        }

        protected void PrintSales()
        {
            for (int i = 0; i < sales.GetLength(0); i++)
            {
                for (int j=0; j < sales.GetLength(1); j++)
                {
                    Console.WriteLine("[{0}][{1}]={2}", i, j, sales[i,j]);
                }
            }
        }

        public static void Main()
        {
            MultiDimArrayApp app = new MultiDimArrayApp();
            app.PrintSales();
        }
    }
```

运行这个 MultiDimArrayApp 程序将产生如下输出结果：

```
[0][0]=0
[0][1]=1
[0][2]=2
[0][3]=3
[0][4]=4
[0][5]=5
[0][6]=6
```

```
[0][7]=7
[0][8]=8
[0][9]=9
[1][0]=100
[1][1]=101
[1][2]=102
[1][3]=103
[1][4]=104
[1][5]=105
[1][6]=106
[1][7]=107
[1][8]=108
[1][9]=109
```

在前面的一维数组示例程序中曾说过，Length 属性将返回数组中所有元素的个数，因此在这个例子中，Length 的返回值就是 20。在 MultiDimArray.PrintSales 方法中使用了 Array.GetLength 方法来确定数组每一维的长度(或者叫上界)，然后在 PrintSales 方法中就可以使用每一个特定的值了。

7.2.4 查询秩

既然我们已经看到动态遍历一维数组或多维数组是多么简单，您可能就会问：如何从程序中确定数组的确切维数呢？数组的维数就叫做该数组的“秩”(rank)，而秩可以通过 Array.Rank 属性来获得。请看下面的示例程序：

```
using System;

class RankArrayApp
{
    int[] singleD;
    int[,] doubleD;
    int[,,] tripleD;

    protected RankArrayApp()
    {
        singleD = new int[6];
        doubleD = new int[6,7];
        tripleD = new int[6,7,8];
    }

    protected void PrintRanks()
    {
```

```
        Console.WriteLine("singleD Rank = {0}", singleD.Rank);
        Console.WriteLine("doubleD Rank = {0}", doubleD.Rank);
        Console.WriteLine("tripleD Rank = {0}", tripleD.Rank);
    }

    public static void Main()
    {
        RankArrayApp app = new RankArrayApp();
        app.PrintRanks();
    }
}
```

正如您可以想到的，RankArrayApp 程序将产生如下输出结果：

```
singleD Rank =1
doubleD Rank =2
tripleD Rank =3
```

7.2.5 锯齿状数组

关于数组的最后一点就是所谓的“锯齿状数组”(jagged array)。锯齿状数组其实就是一个数组的数组。下面的代码定义了一个包含整数数组的数组：

```
int[][] jaggedArray;
```

如果您要开发一个编辑器，可能就会用到锯齿状数组。在这个编辑器中，您希望在一个数组中存储代表用户创建的控件的每一个对象。假设您已经有一个按钮和组合框的数组，您可能希望将三个按钮和两个组合框都分别存储到这个数组中。声明一个锯齿状数组就能够让您拥有一个这些数组的“父”数组，这样在程序上您就可以在需要的时候轻松遍历这些控件，如下所示：

```
using System;

class Control
{
    virtual public void SayHi()
    {
        Console.WriteLine("base control class");
    }
}

class Button : Control
{
```

```
    override public void SayHi()
    {
        Console.WriteLine("button control");
    }
}

class Combo : Control
{
    override public void SayHi()
    {
        Console.WriteLine("combobox control");
    }
}

class JaggedArrayApp
{
    public static void Main()
    {
    Control[][] controls;
    controls = new Control[2][];

    controls[0] = new Control[3];
    for (int i = 0; i < controls[0].Length; i++)
    {
        controls[0][i] = new Button();
    }

    controls[1] = new Control[2];
    for (int i = 0; i < controls[1].Length; i++)
    {
        controls[1][i] = new Combo();
    }

    for (int i = 0; i < controls.Length;i++)
    {
        for (int j=0;j< controls[i].Length;j++)
        {
            Control control = controls[i][j];
            control.SayHi();
        }
    }

    string str = Console.ReadLine();
    }
```

```
}
```

上面的示例程序中定义了一个基类(Control)和两个派生类(Button 和 Combo)，并且声明了一个锯齿状数组作为包含 Control 对象数组的数组。这样就可以将特定的类型存放在特定的数组中了。通过“神奇”的多态性，在从数组中抽取对象的时候(通过向上转换的对象)，就会得到我们希望得到的结果。

7.3 使用索引器将对象当作数组对待

在“数组”一节里，我们学习了如何声明和实例化数组、如何使用不同的数组类型，以及如何遍历数组中的元素，同时还了解到如何利用 System.Arraly 类的一些常用的属性和方法。下面我们继续学习有关数组的知识，C#特有的一个属性——索引器，看看它是如何允许我们在程序上如对待数组一样对待对象。

那么我们为什么会希望把对象当作数组对待呢？与程序语言的大多数属性相似，索引器的好处就在于能够让应用程序的书写更加自然一些。在本章的第一节里，我们已经看到 C#属性是如何使我们能够利用标准的 class.field 语法格式来引用类的字段，最终导致产生了 getter 和 setter 方法。这种抽象将书写客户代码的程序员解放了出来，他们无需确定字段的 getter/setter 方法是否存在，无需知道这些方法的确切格式。同样地，索引器也允许类的客户代码能够在对象中进行索引，就好像对象是一个数组一样。

考虑下面这一个例子。您有了一个列表框类需要公布一些成员，比便使用这个类的用户可以插入字符串。如果您熟悉 Win32 SDK 的话，您就知道为了将一个字符串插入到列表框窗口，需要发送一个 LB_ADDSTRING 或 LB_INSERTSTRING 消息。这种机制在二十世纪八十年代后期出现以后，我们就认为自己真的已经是面向对象的程序员了。总之，我们不是已经给对象发送消息了吗？与那些好笑的、面向对象分析和设计的书籍告诉我们的一样。其实，在诸如 C++和 Object Pascal 这样的面向对象和基于对象的语言扩散的同时，我们已经知道可以使用对象来为这样的任务创建更加自然的程序接口。使用 C++和 MFC(Microsoft Foundation Classes)，我们就有了整个的一个类的格子，使得我们可以将窗口(比如列表框)当作对象对待，这样的对象公开了一些成员函数，这些成员函数基本上是为对象与潜在的 Microsoft Windows 控件发送和接收消息提供了一层薄薄的包装。在 CListBox 类(也就是 Windows 列表框控件的 MFC 包装)这种情况下，我们就有了 AddString 和 InsertString 这样的成员函数来完成任务，而先前这些任务是由发送 LB_ADDSTRING 和 LB_INSERTSTRING 这样的消息来完成的。

不过，为了开发出最好和最自然的语言，C#设计小组看到这些问题并开始思索：“为什么不能将一个其实就是数组的对象当作数组对待呢？”比如列表框不就是一个带有额外

的显示和排序功能的字符串数组吗？正是从这个想法诞生了索引器这一概念。

7.3.1 定义索引器

因为属性有时候被叫做“智能字段”，而索引器被称为“智能数组”，因此属性和索引器共享同样的语法格式也是合理的。实际上，定义索引器与定义属性极其相似，只有两点不同：第一，索引器有索引参数值；第二，由于类本身被当作数组使用，就将关键字“this”当作索引器的名称。我们很快会看到一个更完整的示例程序，先看看下面这个索引器例子：

```
class MyClass
{
    public object this [int idx]
    {
        get
        {
            // Return desired data.
        }
        set
        {
            // Set desired data.
        }
    }
    ...
}
```

此处还没有展示一个完整的例子来说明索引器的语法格式，因为定义数据以及如何得到或设置这些数据的内部实现其实与索引器无关。要记住不管在内部如何存储数据(存储为数组、集合等等)，索引器都只不过是一种途径，以便实例化类的程序员可以这样来编写代码：

```
MyClass cls = new MyClass();
cls [0] = someObject;
Console.WriteLine("{0}",cls[0]);
```

至于在索引器内部执行什么操作，那就是您自己的事儿了，只要使用类的客户在访问被当作数组的对象时能得到希望的结果即可。

7.3.2 索引器示例程序

下面看看在哪些场合可以使用索引器。还是以前面用过的列表框为例，从概念上来说，列表框就是一个列表，或者说是要显示的一个字符串数组。在下面的例子中，声明了一个

叫做“MyListBox”的类，它包含了一个索引器通过 ArrayList 对象来设置和获取字符串(ArrayList 类就是一个用于存储对象集合的.NET 框架类)。

```
using System;
using System.Collections;

class MyListBox

{
   protected ArrayList data = new ArrayList();

   public object this[int idx]
   {
      get
      {
         if (idx > -1 && idx < data.Count)
         {
            return (data[idx]);
         }
         else
         {
            // Possibly throw an exception here.
            return null;
         }
      }
      set
      {
         if (idx > -1 && idx < data.Count)
         {
            data[idx] = value;
         }
         else if (idx == data.Count)
         {
            data.Add(value);
         }
         else
         {
            // Possibly throw an exception here.
         }
      }
   }
}

class IndexersApp
```

```
{
    public static void Main()
    {
        MyListBox lbx = new MyListBox();
        lbx[0] = "foo";
        lbx[1] = "bar";
        lbx[2] = "baz";
        Console.WriteLine("{0} {1} {2}",
                          lbx[0], lbx[1], lbx[2]);
    }
}
```

注意在这个例子中我们检查了数据索引时可能会发生的越界错误。这种技术并不一定要与索引器一起使用，我们已经说过，索引器只需做到使用类的客户使用这个被当作数组的对象时感觉一切正常即可，而与数据内部如何表示无关。不过，在学习一种新的语言属性的时候，了解这种属性的实际用法总比只看它的语法格式有用。因此，在索引器的 getter 和 setter 方法中，我们校验了存储在类的 ArrayList 成员中的数据传递过来的索引值。作者个人可能会选择在传递过来的索引值无法解析时抛出异常，但这也仅仅是个人习惯——各人的错误处理方式也许会不同。关键在于如果传递的是非法索引时，您需要对客户代码指出这个错误。

7.3.3 设计规则

索引器是 C#设计小组提供的另一个很灵巧、也很强大的语言属性，有助于我们在开发过程中提高工作效率。但是，与任何语言的任何属性一样，索引器也要用得恰当。它们只能用在将对象当作数组会更自然的场合。让我们看一看开发票的那个例子。开发票的应用程序有一个 Invoice 类，它定义了一个 InvoiceDetail 对象的成员数组。这种情况下，用户要访问这些细节栏，采用如下的语法会更自然一些：

```
InvoiceDetail detail = invoice [2 ];  // Retrieves the 3rd detail line
```

不过，如果更进一步试图将所有的 InvoiceDetail 成员都转换为数组以便通过索引器来访问的话，就会适得其反了。这里您可以看到，下面第一行代码比第二行代码更容易理解得多：

```
TermCode terms = invoice.Terms; // Property accessor to Terms member.
TermCode terms = invoice [3 ];  // a solution in search of a problem.
```

这种情况下，您可以做某事但并不意味着您必须做某事，而要具体情况具体分析。或者，更具体一点地讲，要想想实现的任何新属性会如何影响使用您的类的客户代码，在您

决定要实现这些属性时，要保证客户代码能够更轻松地使用您的类。

7.4 本章小结

C#属性由字段声明和存取器方法组成。属性允许对类的字段进行智能访问，这样使用类书写客户代码的程序员就无需确定是否应该、以及如何为该字段创建存取器方法了。C#中的数组声明时只需将空的方括号放置在变量类型和变量名之间即可，这一点与 C++中使用的语法格式稍微有所不同。C#数组可以是一维数组、多维数组或者锯齿状数组。通过索引器的使用，C#中的对象可以被当作数组对待。索引器允许程序员可以更轻松地处理和跟踪同样类型的多个对象。

第8章 特　　性

大多数的编程语言都是按照设计者头脑当中设想的功能来设计的。例如，当您开始设计一个编译器的时候，您将会考虑用新语言编写的程序将会具有怎样的结构、代码之间如何调用、功能是如何封装的，以及许多其他力图提高这种语言的编程效率的问题。编译器设计人员最初得到的东西多数情况下都是静态的。例如，在 C#里面，您通过在一个类名之前加上 class 关键词来定义一个类。然后您在类名后加一个冒号，跟上一个基类名，以此来标明这个类的源。这是一种设计思路的例子，语言设计员一旦决定这种设计思路，就不会再改变了。

现在，那些编写编译器的人员对他们现在所做很擅长。但是，即使是他们也不能预期我们行业的未来发展，以及这些发展是怎样来改变程序员用某种语言表示其类型的方式的。例如，您怎样来创建 C++中的一个类和这个类的 URL 记录之间的关系？还有您怎样把 C++类中的一个具体的成员和您公司的 B-to-B 的方案的 XML 信息组关联起来？因为在早于 Internet 和诸如 XML 等协议出现好多年之前，C++就被设计出来了，所以要实现这些任务中的任何一个都不是件容易的事情。

直到现在，对这些问题人们正在考虑这样一些方案，即在一个独立的文件中 (比如 DEF，IDL 等等)存储一些额外的信息，然后再把这些文件和那些类或者成员松散地关联起来。由于编译器不知道这个独立文件或类与文件之间由代码生成了何种关系，因此这种方法经常称为“分离解决方案”。主要的问题在于这个类不再是“自我描述性”的类了，也就是说，用户不能再自己来查看类的定义并了解这个类的所有细节了。自我描述性组件的一个优点就在于：编译器和运行环境可以保证组件的关联关系。另外，对于一个自我描述性组件，由于开发者可以看到所有和这个类相关的信息，所以，自我描述性组件更易于维护。

全世界的编译器发展都是按照这种方式演化了好几十年。语言设计者试图您对语言都有哪些要求，然后他们根据这些要求来设计编译器，并且无论好坏，您都要在下一个编译器出现之前一直使用它。也就是说，直到现在这种情况还是如此。C#提供了一种不同的模式，它起源于对一种叫做特性 (attributes)的引入。

8.1 特性的介绍

特性给您带来了前所未有的新东西。它们为您提供了一种通用的、把信息 (如注解) 与

您定义的C#类型关联起来的方式。您可以用特性来定义设计时的信息（如文件信息等）、运行时的信息（如数据库字段的列名等），甚至也可以定义运行时的行为特征(如一个预设的成员是否“能被处理”——即具有参与事务处理的能力）。特性的用法和功能简直就是举不胜举。因为您可以根据您所喜欢的任何信息创建特性，所以有一套标准机制来定义特性本身，并在运行时根据成员的附加特性对成员和类型进行查询。

一个例子可以更好地说清楚怎样使用这个具有强大功能的特征。假如您有一个在Windows注册表里存储信息的程序。那么您要解决的一个设计难题是在何处存储注册表的键值信息。在大多数的开发环境中，一种典型的方式就是把该信息存储在一个资源文件里，或者存在常量里，或者是把该信息牢固地编写到Registry API的调用中。但是，这种方式会把原本是一个整体的类与该类的定义部分分开来存储。如果使用特性，我们就可以把这部分信息附在类成员上，这样的话，我们就有一个完全自我描述的组件了。下面通过一个例子来看看这些是如何实现的，假定我们已经在其他地方定义了一个叫做RegistryKey的特性。

```
class MyClass
{
    [RegistryKey(HKEY_CURRENT_USER, "foo")]
    public int Foo;
}
```

要给一个C#类型或者成员附加一个已定义的特性，您只要在目标类型或成员的前面加上特性数据，并用括号把它们括起来即可。例如在上面的代码中我们给MyClass.Foo字段附加了一个名为RegistryKey的特性。在运行时我们就可以立刻看到——对于注册表键值我们要做的就是查询这个字段，然后用这个值把日期保存到注册表里。

8.2 特性的定义

在前面的例子中，您会注意到给一个类型或成员附加特性的句法看起来有点像类的实例化。这是因为一个特性确切地说就是一个从System Attribute基类派生的类。现在让我们看一看RegistryKey特性的结构。

```
public enum RegistryHives
{
    HKEY_CLASSES_ROOT,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
    HKEY_USERS,
    HKEY_CURRENT_CONFIG
```

```
}

public class RegistryKeyAttribute : Attribute
{
    public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
    {
        this.Hive = Hive;
        this.ValueName = ValueName;
    }

    protected RegistryHives hive;
    public RegistryHives Hive
    {
        get { return hive; }
        set { hive = value; }
    }

    protected String valueName;
    public String ValueName
    {
        get { return valueName; }
        set { valueName = value; }
    }
}
```

我们在这里加了一个 enum 来声明一个不同的 Registry 类型，它有一个特性类的构造函数（带有一个 Registry 类型参数和一个值名参数），两个属性（Registry 配置单元和值名）。您可以有很多种方法来定义特性，现在，我们知道了怎样定义和附加特性，那么让我们接着学习怎样在运行时查询特性。这样我们就可以有一个完整的例子来开始我们的工作了。一旦完成了这部分，我们就可以继续处理一些定义和附加特性的更高级的问题。

注意： 注意在这个例子中特性类名后附加了 Attribute。但是，当在以后把这个特性附加给一个类型或成员时，不需要再包括这个 Attribute 后缀了。这是 C# 语言设计员们提供的一种快捷方式。当编译器看到一个特性附加给一个类型或成员时，它就会搜索带有指定名字的 System.Attribute 的派生类。如果没有找到这样的类，编译器就会先给这个特性加上 Attribute，然后再搜索这个特性。因此，通常的做法是：定义特性类名时以 Attribute 结尾，之后常常省略 Attribute 这部分。

8.3 对特性进行查询

我们已经知道怎样来通过 System.Attribute 派生来定义一个特性，以及怎样把它附加给一个类型或成员。那么在代码中我们怎样使用特性呢？换句话说，我们怎么样才能根据所附加的特性（和它的参数）来查询一个类型或者一个成员呢？

要查询一个类型或者成员的附加特性，我们需要用到反射(reflection)。反射是一个高级的主题，它将在第 16 章中详细讨论，所以在这里不做赘述，此处暂且只讨论一下在运行时要获得特性信息需要哪些东西。如果您想知道更多有关反射的信息，请参考第 16 章。

反射就是这样一种特征：它允许您可以在运行时动态地决定应用程序的类型特点。例如，您可以使用 .NET 的框架反射 API 来通过对元数据的遍历获得一个完整的组件，并且生成这个组件的所有类、类型和方法的清单。下面让我们根据一些特性的例子来看看怎样通过使用反射来查询特性。

8.3.1 类的特性

怎样获得一个特性取决于您要查询的成员类型。假如您要定义一个特性，这个特性被用来定义创建对象的远程服务器。不使用特性的话，您就要把这个信息存储在一个常量里或者是应用程序的资源文件里。使用特性的话，您只需简单地(像下面这样)为这个类添加一个远程服务器的注释即可。

```
using System;

public enum RemoteServers
{
    JEANVALJEAN,
    JAVERT,
    COSETTE
}

public class RemoteObjectAttribute : Attribute
{
    public RemoteObjectAttribute(RemoteServers Server)
    {
        this.server = Server;
    }

    protected RemoteServers server;
```

```
    public string Server
    {
        get
        {
            return RemoteServers.GetName(typeof(RemoteServers),
                                         this.server);
        }
    }
}

[RemoteObject(RemoteServers.COSETTE)]
class MyRemotableClass
{
}
```

要确定创建对象的远程服务器，可以使用如下代码：

```
class ClassAttrApp
{
    public static void Main()
    {
        Type type = typeof(MyRemotableClass);
        foreach (Attribute attr in type.GetCustomAttributes())
        {
            RemoteObjectAttribute remoteAttr =
                attr as RemoteObjectAttribute;
            if (null != remoteAttr)
            {
                Console.WriteLine("Create this object on {0}.",
                                  remoteAttr.Server);
            }
        }
    }
}
```

正如您所期望的，应用程序的输出如下所示：

```
Create this object on COSETTE.
```

上面的例子很有代表性，我们可以通过它来测试一下反射是怎样工作的，以及运行时是怎样返回特性的值的。

您会注意到在主程序的第一行使用了 ***typeof*** 操作符。

```
Type type =typeof(MyRemotableClass);
```

这个操作符返回和这个作为它惟一的参数值的类型相关的 System Type 对象。一旦您

得到这个对象，您就可以对它进行查询。对于代码的下一行有两部分需要解释。

```
foreach (Attribute attr in type.GetCustomAttributes(true))
```

第一部分是对 ***Type.GetcustonAttribute*** 方法的调用。这个方法返回一个 ***Attribute*** 类型的数组，这个数组包含所有的附加给名为 ***MyRemotableClass*** 的类的特性。第二部分是 ***foreach*** 语句，***foreach*** 语句遍历返回的数组，依次给 ***Attribute*** 类型的变量 (*attr*)赋值。

下面的这个语句使用了 ***as*** 操作符，它用来把 *attr* 变量转换成 ***RemoteObjectAttribte*** 类型。

```
RemoteObjectAttribute remoteAttr =
attr as RemoteObjectAttribute;
```

接下来，我们看看 as 转换是否产生了空值。如果没有产生空值——这就意味着 remoteAttr 变量把一个有效的特性附加给了 MyRemotableClass 类型——我们就可以调用 RemoteObjectAttribute 的一个属性来打印这个远程服务器的名字了。

```
if (null != remoteAttr)
{
   Console.WriteLine("Create this object on {0}",
                     remoteAttr.Server);
}
```

8.3.2 方法的特性

现在我们已经看到怎样使用类的特性了，下面让我们接着学习方法的特性。之所以将这部分单列出来进行讨论，是因为对方法的特性查询的反射代码不同于对类的特性查询的反射代码。在这个例子里，我们将使用一个特性用来定义一种可进行事务处理的方法。

```
using System;
using System.Reflection;

public class TransactionableAttribute : Attribute
{
   public TransactionableAttribute()
   {
   }
}

class TestClass
{
   [Transactionable]
   public void Foo()
   {}
```

```
    public void Bar()
    {}

    [Transactionable]
    public void Baz()
    {}
}

class MethodAttrApp
{
    public static void Main()
    {
        Type type = Type.GetType("TestClass");
        foreach(MethodInfo method in type.GetMethods())
        {
            foreach (Attribute attr in
                    method.GetCustomAttributes()) .
            {
                if (attr is TransactionableAttribute)
                {
                    Console.WriteLine("{0} is transactionable.",
                                      method.Name);
                }
            }
        }
    }
}
```

代码的输出部分如下：

```
Foo is transactionable.
Baz is transactionable.
```

在这个特殊的例子里，仅仅凭借 TransactionableAttribute 就可以让代码知道具有这种特性的方法可以参与事务处理。这也是为什么在这里只有一个简单的、不带参数的构造函数，而没有其他成员。然后 TestClass 定义了三种方法：Foo 方法、Bar 方法和 Baz 方法；其中 Foo 方法和 Baz 方法被定义为具有处理事务能力的方法。请注意当您附加一个带构造函数的特性时，如果这个构造函数不带参数，您就不需要把左括号和右括号包括进去了。

现在让我们来看看这个程序当中值得关注的部分，看看怎样通过方法的特性来查询类的方法。我们开始先用静态的 Type 方法 GetType 来获得 TestClass 类的 System.Type 对象。

```
Type type =Type.GetType("TestClass ");
```

然后我们使用 Type.GetMethods 方法来得到一个 MethodInfo 对象数组。每一个这样的对象都包括 TestClass 类的一个方法信息。我们用 foreach 语句来循环处理我们的每一个方法。

```
foreach(MethodInfo method in type.GetMethods())
```

现在我们有了一个 MethodInfo 对象，我们就可以使用 MethodInfo.GetCustom Attributes 方法来得到所有的用户创建的方法特性。我们还是使用 foreach 语句来循环处理返回的对象数组。

```
foreach (Attribute attr in method.GetCustomAttributes(true))
```

在代码的这个地方，我们的方法有了特性。现在，通过使用 is 操作符，我们来判断一个特性是不是一个 TransactionableAttribute 特性。如果它是，就打印出这个方法的名字。

```
if (attr is TransactionableAttribute)
{
Console.WriteLine("{0} is transactionable.",
                  method.Name);
}
```

8.3.3 字段的特性

在最后的这个例子里，我们要根据成员的附加特性来进行成员的查询。我们会看到怎样来进行一个类的字段查询。假如您有一个包含一些字段的类，您想把这些字段的值存储在注册表中。要想这样，您可以利用构造函数来定义一个特性，并且使该构造函数取两个参数，一下为枚举类型，代表正确的注册表配置单元；一个为字符串类型，代表注册表的值名。下面您就可以根据字段的注册键值在运行时进行字段的查询了。

```
using System;
using System.Reflection;

public enum RegistryHives
{
    HKEY_CLASSES_ROOT,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
    HKEY_USERS,
    HKEY_CURRENT_CONFIG
}

public class RegistryKeyAttribute : Attribute
```

```
{
    public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
    {
        this.Hive = Hive;
        this.ValueName = ValueName;
    }

    protected RegistryHives hive;
    public RegistryHives Hive
    {
        get { return hive; }
        set { hive = value; }
    }

    protected String valueName;
    public String ValueName
    {
        get { return valueName; }
        set { valueName = value; }
    }
}

class TestClass

{
    [RegistryKey(RegistryHives.HKEY_CURRENT_USER, "Foo")]
    public int Foo;

    public int Bar;
}

class FieldAttrApp
{
    public static void Main()
    {
        Type type = Type.GetType("TestClass");
        foreach(FieldInfo field in type.GetFields())
        {
            foreach (Attribute attr in field.GetCustomAttributes())
            {
                RegistryKeyAttribute registryKeyAttr =
attr as RegistryKeyAttribute;
                if (null != registryKeyAttr)
                {
```

```
                    Console.WriteLine
                        ("{0} will be saved in {1}\\\\{2}",
                         field.Name,
                         registryKeyAttr.Hive,
                         registryKeyAttr.ValueName);
                }
            }
        }
    }
}
```

我们不需要对这段代码的所有部分都再仔仔细细地讲解了，因为它有一部分是和前面的例子重复的。但是，有两个细节很重要。第一个是您要注意到，就像 MethodInfo 对象被定义用来获得类型对象的方法信息一样，FieldInfo 也被用来提供同样的功能，即从类型对象那里获得字段信息。像前面的例子一样，我们从获得与我们的测试类相关的类型对象开始，然后逐个访问 FieldInfo 数组，对每一个 FieldInfo 对象，我们要查询它的那些特性直到我们找到要找的特性：RegistryKeyAttribute。如果我们找到了该特性，我们就打印出该字段的名字，并从这个特性中检索出 Hive 字段和 ValueName 字段信息。

8.4 特性的参数

在上面的例子中，通过构造函数我们讨论了附加特性的使用。现在我们要来看看在前面没有谈到的特性的构造函数的一些方面。

8.4.1 定位参数和命名参数

在前一节的 FieldAttrApp 例子中，您看到了一个名为 RegistryKeyAttribute 的特性。它的构造函数形式如下：

public RegistryKeyAttribute(RegistryHives *Hive*, String *ValueName*)

在这个构造函数声明之后，通过如下这种形式特性就附加给了一个字段：

```
[RegistryKey(RegistryHives.HKEY_CURRENT_USER,"Foo ")]
public int Foo;
```

到此为止，这些都很容易理解。这个构造函数有两个参数，这两个参数都是在把一个特性附加给一个字段时用到的。不过，我们可以让这种编程更简单。如果这个参数大多数时候都不变，那为什么每次都要让使用这个类的用户再费劲地输入这些参数呢？我们可以使用定位参数(positional parameter)和命名参数来给这些参数设置默认值。

定位参数是用在构造函数中的参数。在每次使用特性时它们是必需的参数，并且要必须指明这些参数。在上面的 RegistryKeyAttribute 例子中，Hive 和 ValueName 都是定位参数。命名参数在特性的构造函数中实际上并没有定义，更确切地说，它们是非静态的字段和属性。因此，在一个特性被实例化时，命名参数让客户端能够设置这个特性的字段和属性，而不必让您为客户端要设置的每一种字段和属性的可能的组合而创建构造函数。

每一个公共的构造函数都可以定义一系列的定位参数，就像所有类型的类一样。但是，对于特性来说，一旦它的定位参数被确定，用户就可以使用 FieldOrPropertyName=Value 来对某个字段或属性进行引用。下面我们通过对特性 RegistryKeyAttribute 的修改来解释一下这种情况。在这个例子里，我们取 RegistryKeyAttribute.ValueName 作为惟一的定位参数，而 RegistryKeyAttribute.Hive 就成了可选的命名参数。接下来，问题就是“您怎样才能把一些参数定义为命名参数？”因为只有定位参数——即必需性的——参数才包括在构造函数的定义中，因此我们只需简单地把这个可选参数从构造函数的定义中删除即可。然后用户就可以引用如下部分作为命名参数：非只读、静态或常量的任何字段，或包括设置存取器方法或非静态的 setter 的任何属性。因此，为了使 RegistryKey Attribute.Hive 成为一个命名参数，我们要把它从构造函数的定义中删除，因为它作为一个公共的读/写属性已经存在了。

```
public RegistryKeyAttribute(String ValueName)
```

用户现在就可以用下面的任一种方法来附加特性了：

```
[RegistryKey("Foo ")]
[RegistryKey("Foo",Hive =RegistryHives.HKEY_LOCAL_MACHINE)]
```

采用这种方式具有很好的灵活性，您既可以使用字段的默认值，同时，又可以让用户能够在需要的时候用其他值覆盖原来的默认值。但是要记住：如果用户没有设置 RegistryKeyAttribute.Hive 字段的值，我们怎样来默认它？您也许会想到，“哦，我们来检查一下看它是不是在构造函数中设置了。”但是，问题是 RegistryKeyAttribute.Hive 是一个 enum 型的，它的底层数据类型是 int 型——它是一个数值。这就意味着在定义时编译器已经把它初始化为 0 了！如果我们测试一下构造函数中的 RegistryKeyAttribute.Hive 的值就会发现它等于 0，我们不知道，是由调用程序通过命名参数设置的那个值，还是由编译器在编译时，因为它是一个数值型才给它初始化设置了该值。不幸的是，现在能解决这个问题的惟一的途径是改变这段代码，让它的值为 0 时无效。这可以通过如下改变 RegistryHives enum 的方式实现：

```
public enum RegistryHives
{
    HKEY_CLASSES_ROOT = 1,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
```

```
        HKEY_USERS,
        HKEY_CURRENT_CONFIG
    }
```

现在我们知道，使 RegistryKeyAttribute.Hive 为 0 的惟一途径是编译器把它初始化为 0，并且用户没有通过一个命名参数来覆盖它的初值。我们可以用类似下面的代码来将其初始化：

```
public RegistryKeyAttribute(String ValueName)
    {
        if (this.Hive == 0)
            this.Hive = RegistryHives.HKEY_CURRENT_USER;

        this.ValueName = ValueName;
    }
```

8.4.2 使用命名参数时的常见错误

当您使用命名参数时，您必须首先要指定定位参数。之后，由于命名参数是放在字段名或属性之前的，因此命名参数之间没有先后次序。下面的例子将导致一个编译错误：

```
//This is an error because positional parameters can't follow
//named parameters.
[RegistryKey(Hive=RegistryHives.HKEY_LOCAL_MACHINE,"Foo ")]
```

另外，您不能命名定位参数。编译器在编译特性的使用时，它会试着先去解析那些命名参数。然后再试着根据方法特性去解析剩下的——定位参数值。本段下面的代码无法通过编译，因为在要解析的参数当中必须包含至少一个定位参数，如果全部都是命名参数，就会出现提示“No overload for method'Registrykey Attribute'takes'O'arguments”(Registrykey Attribute 方法没有参数值，无法进行重载)。

```
[RegistryKey(ValueName="Foo",Hive=RegistryHives.HKEY_LOCAL_MACHINE)]
```

最后，命名参数可以是任何公共的、可存取的字段或属性——包括 setter 方法——只要不是静态的或常量即可。

8.4.3 有效的特性参数类型

特性类的定位参数和命名参数的类型仅限于特性参数类型，这些包括：

- bool, byte, char, double, float, int, long, short, string
- System.Type

- object
- enum 类型，前提是它或任何有它嵌套在里面的类型必须是公共的可存取类型——就像在那个使用 RegistryHives 枚举的例子中一样。
- 由上述的任何类型组成的一维数组。

因为有效的参数类型仅局限于上述列出来的类型，因此您不能把一个像类那样的数据结构传递给特性构造函数作为参数。这种限制很有意义，因为特性是在程序设计时附加上的，此时您并没有这个类（对象）的实例化的实例。使用上面列出来的这些有效类型，您就可以在程序设计时把它们的值固定下来，这就是为什么能使用它们的原因。

8.5 AttributeUsage 特性

除了可以定制自己的特性来注释常用的 C#类型外，您还可以用 AttributeUsage 特性来定义您想怎样使用这些特性。AttributeUsage 特性采用如下的调用惯例：

[AttributeUsage(
validon,
AllowMultiple = *allowmultiple,*
Inherited = *inherited*
)]

您可以非常容易地区别出哪些是定位参数，哪些是命名参数。强烈建议您在记录您的特性时采用这种格式，以便不必通过查看特性类的源代码，您的用户即可以找到哪些公共的读/写字段和属性可以用作命名特性。

8.5.1 定义一个特性目标

现在，让我们再来看看前一节的 AttributeUsage 特性，您会注意到 validon 参数是一个定位的(必需的)参数。这个参数使您能够指定您的特性都能附加给哪些类型。确切地说，AttributeUsage 特性中的 validon 参数是 AttributeTargets 类型的，它事实上是按如下方式定义的一种枚举：

```
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,
    All = Assembly | Module | Class | Struct | Enum | Constructor |
          Method | Property | Field | Event | Interface | Parameter |
          Delegate,

    ClassMembers  =  Class | Struct | Enum | Constructor | Method |
                     Property | Field | Event | Delegate | Interface,
}
```

注意当使用 AttributeUsage 特性时您可以指定 AttributeTargets.All，这样的话，这个特性就可以附加给任何在 AttributeTargets 枚举中列出的类型了。这是在您不指定 AttributeUsage 特性时的默认方式。假如 AttributeTargets.All 是默认值，您可能会觉得奇怪：为什么您还要使用 validon 值。其原因就是命名参数可以在这个特性上使用，您也许要改变它们其中的一个。但是您要记住：如果您使用了一个命名参数，您必须把它放在所有的定位参数的后面。这可以让您很方便地指定您想使用的特性采用 AttributeTargets.All 默认值，同时您仍然可以设置它们的命名参数。

那么，在什么时候并且为什么您指定这个 validon (AttributeTargets) 参数？在任何您想完全控制怎样使用一个特性时，您都可以使用这个参数。在上面的这些例子中，我们创建了一个只有类才能应用的 RemoteObjectAttribute 特性，还创建了一个只能用在方法上的 TransactionableAttribute 特性，以及一个只对字段有作用的 RegistryKeyAttribute 特性。如果我们想让这些特性只注释在设计时它们所注释的那些类型，我们可以这样来定义它们（此处为简明起见略去了特性的主体）：

```
[AttributeUsage(AttributeTargets.Class)]
public class RemoteObjectAttribute : Attribute]
{
    ...
}

[AttributeUsage(AttributeTargets.Method)]
public class TransactionableAttribute : Attribute
{
    ...
}
```

```
[AttributeUsage(AttributeTargets.Field)]
public class RegistryKeyAttribute : Attribute
{
    ...
}
```

最后关于 AttributeTargets 枚举有一点要提一下：您可以使用“|”操作符来组合成员。如果您有一个既要应用到字段和又要应用到属性的特性，您就可以按如下格式来附加这个 AttributeUsage 特性：

```
[AttributeUsage(AttributeTargets.Field |AttributeTargets.Property)]
```

8.5.2 单次特性和多次特性

您可以用 AttributeUsage 来把特性定义为单次或者是多次的特性。使用哪种形式取决于一个单独的特性在一段单独的字段里的使用次数。在默认形式下，所有的特性都是单次的，这意味着在编译如下代码时将导致一个编译错误：

```
using System;
using System.Reflection;

public class SingleUseAttribute : Attribute
{
    public SingleUseAttribute(String str)
    {
    }
}

// ERROR: This results in a "duplicate attribute" compiler error.
[SingleUse("abc")]
[SingleUse("def")]
class MyClass
{
}

class SingleUseApp
{
    public static void Main()
    {
    }
}
```

为了解决这个问题，您需要在 AttributeUsage 那一行指定您要把这个特性多次附加给某

些类型。具体方法如下：

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All, AllowMultiple=true)]
public class SingleUseAttribute : Attribute
{
    public SingleUseAttribute(String str)
    {
    }
}

[SingleUse("abc")]
[SingleUse("def")]
class MyClass
{

}

class MultiUseApp
{
    public static void Main()
    {
    }
}
```

您可能用到这种方法的一个实际例子就是用在“特性的定义”一节中所讨论到的RegistryKeyAttribute 特性上。因为我们知道一个字段可能会存储在注册表的多个地方，您可能需要(就像这里的代码一样)通过 AllowMultiple 命名参数把 AttributeUsage 特性附加给它们。

8.5.3 指定继承特性的规则

AttributeUsageAttribute 特性的最后一个参数是它的 inherited 标记，即用来标识这个特性是否可以被继承。它的默认值是 false。但是，如果 inherited 标记被设置成 true，那么它还需要看 AllowMultiple 标记的值。如果 inherited 标记是 true 而 AllowMultiple 标记是 false，这个特性就不再有继承特性了。不过，如果 inherited 标记是 true 并且 AllowMultiple 标记也是 true，这个特性就可以累加在这个成员上。

8.6 特性标识符

请看一下下面的代码，并试着标出这个特性是否注释了其中的返回值或方法：

```
class MyClass
{
    [HRESULT]
    public long Foo();
}
```

如果您具有 COM 编程经验的话，您就会知道 HRESULT 是所有除了名为 AddRef 或 Release 的方法的基本返回类型。不过，我们可以很容易就明白，如果特性的名字既可以用于返回值又可以用于方法名的话，编译器就不可能知道您的用意到底是什么。下面是编译器不能根据上下文掌握您的真正用意的其他一些情况：

- 方法和返回类型
- 事件、字段和属性
- 代表和返回类型
- 属性、存取器、getter 方法的返回值和 setter 方法的值参数

在这些情况下，编译器根据它所认为的最可能原则来进行判断。为了不让这种判断出现，您可以使用下面所列的特性标识符：

- assembly
- module
- type
- method
- property
- event
- field
- param
- return

要使用特性标识符，只需在这个特性名字前面加上所需的标识符和一个冒号即可。在 MyClass 例子中，如果您想保证编译器能够判断出 HRESULT 是用来注释返回值而不是其中的方法，您可以这样指定标识符：

```
class MyClass
{
    [return:HRESULT]
```

```
        public long Foo();
    }
```

8.7 本章小结

C#特性提供了这样一种机制，就是在设计时可以用信息来对类型和成员进行注释，并且可以在运行时通过反射(reflection)来获得这些信息。这使您可以真正创建自我包含的、自我描述的组件，而不必借助于向一些资源文件和常量中填塞一些必需的琐碎信息。这样有利于编写出更简单、更易于维护、移植性更强的组件。

第9章 接　　口

理解接口的关键是把接口与类放在一起比较。类是具有属性和在这些属性上操作的方法的集合。尽管类确实展示了一些行为特征(方法)，然而类更是一种与行为相对的结构，而这正是为什么出现接口的原因。接口使您可以定义行为特征或能力，而且可以直接向对象应用这些行为，而无需考虑类的层次。例如，假设您有一个分布式应用程序，其中有些实体可以被串行化。这些实体包括 Customer、Supplier 和 Invoice 类。而有些其他类，例如 MaintenanceView 和 Document，无法被定义成可串行化。怎样才能使自己选择的类成为可串行化的呢？一个显而易见的方法就是，创建一个基础类，可以给这个类起一个诸如 Serializable 之类的名字。然而，这种方法有一个主要的缺陷。单继承路径无法正常工作，因为我们不希望类的所有行为都被共享。C#并不支持多继承，因此无法使一个给定的类可以选择性地从多个类中派生。所以真正的答案就是接口。接口使您能够定义一组语义上相关的方法和属性，所选择的类可以实现这些方法和属性，而无需考虑类的层次关系。

从概念的角度来看，接口是两个不同代码段之间的约定。也就是说，一旦定义了一个接口，而且定义了一个实现这个接口的类，类的客户端就可以得到保证——这个类已经实现了接口中所定义的所有方法。让我们看几个例子，您很快就可以明白这一点了。

在本章中，我们将考虑接口为什么是 C#以及一般的基于组件的编程中如此重要的一个部分。然后我们将看一看如何在 C#应用程序中声明和实现接口。最后，我们将深入学习更多特定细节，这些细节是关于使用接口以及克服多继承和名字冲突所带来的问题的。

注意　当您定义了一个接口，并在接口的定义中指定了使用这个接口的类时，我们就称这个类“实现了该接口”或是“从接口继承”。这两种说法都可以使用，而且您将会见到这两种说法在其他文章中交替使用。从个人角度来说，相对于“从其他类继承”的说法，作者认为“实现”是一个语义上更为正确的术语——接口被定义了行为，并且类被定义成实现该行为——但是这两种术语都是正确的。

9.1　接口的应用

为了理解接口的用途所在，让我们先看一看 Microsoft Windows 开发中一个传统的编

程问题，其中没有使用接口，但是里面有两段完全不同的代码需要以一种普通的方式进行通信。假如您为 Microsoft 工作，而且您是控制面板小组(Control Panel Team)的一个出色的程序员。您需要为控制面板中所包含的所有客户端程序提供一种通用的手段，利用这种手段可以在控制面板中显示这些客户端程序的图标，而且最终用户可以执行这些程序。请记住，在引入 COM 之前就要设计出这种功能。您怎样才能为所有未来的应用程序都提供一种通用手段，使之可以利用这种手段集成到控制面板中呢？多年以来，所考虑的解决方案已经成为 Windows 开发中的一个标准部分了。

作为控制面板开发小组中的一位出色的程序员，您要设计一个(或多个)需要客户端应用程序来实现的功能，并为这些功能编写文档，此外还要设计一些规则。在控制面板程序的情况中，Microsoft 决定编写一个您所需要的控制面板程序，来创建一个动态链接库(DLL)，该动态链接库实现并导出了一个名为 CPIApplet 的函数。您还需要用扩展名 .CPL 来扩展这个 DLL 的名字，并把该 DLL 放到 Windows System32 文件夹中(在 Windows ME 或者是 Windows98 中，应该放在文件夹 Windows\System32 中，而在 Windows 2000 中，应该放在文件夹 WINNT\System32 中)。一旦控制面板加载，就会(利用函数 LoadLibrary)把 System32 文件夹中所有扩展名为.CPL 的文件加载进来，然后使用函数 GetProcAddress 来加载 CPIApplet 函数，从而检验您是否遵循了这些规则，并且检验是否可以与控制面板进行通信。

正如前面所提到过的，如果要处理这种情况，即有一些代码需要与一些现在未知的代码以一种通用的方法进行通信，这已经成为 Windows 中一种标准的编程模型。然而，这并不是最好的解决方法，而且这种方法自身也有明显的缺陷。这种技术最大的缺点就是，它强迫客户端——在这个例子中是控制面板代码——中包含了大量有效性确认代码。例如，控制面板不能仅仅只假设文件夹中的任何.cpl 文件都是 Windows DLL 文件。此外，控制面板还需要证实在那个 DLL 中有正确的函数，并且要证实那些功能能够完成文件所指定的工作。我们就是要在这里使用接口。接口使您能够在完全不同的代码之间创建相同的约定安排，但是是以一种更为面向对象和灵活的方式来完成的。此外，因为接口是 C#语句的一部分，所以编译器确保了当一个类被定义为实现一个给定接口时，这个类可以完成其应该完成的工作。

在 C#中，接口是“头等”概念，其中声明了一个引用类型，这个引用类型中只包含方法声明。“头等概念”到底是什么含义？该术语的意思是，当前所讨论的这个特性是该语言中一种内置的、集成的部分。换句话说，这不是在该语言被设计出来以后才加入进来的特性。现在让我们来深入地了解一些细节——接口是什么以及如何声明接口。

注意 C++开发人员请注意，一个接口基本上就是一个抽象类，这个抽象类中除了声明 C#类的其他成员类型——例如属性、事件和索引器以外，只声明了纯虚拟方法。

9.2 声明接口

接口中可以包含方法、属性、索引器和事件——其中任何一种都不是在接口自身中来实现的。让我们看一个例子，这样就可以懂得如何使用这个特性。假设您在为您的公司设计一个编辑器，在该编辑器上面要处理不同的 Windows 控件。您正在编写编辑器和例程，这些例程被用来对用户放在编辑器表单中的控件进行有效性检查。开发小组的其他人员编写表单上应该存放的控件。您几乎一定需要提供一些表单级的有效性。在适当的时候——例如当用户显式地告诉表单来对所有的控件或者是在表单处理期间进行有效性检查——表单就会迭代其附加控件并且对每个控件进行有效性检查，或者更为适当地，通知控件对自身进行检查。

您如何才能向控件提供这种有效性确认能力？正是在这里接口有着重要的作用。下面有一个简单的接口例子，其中包含了一个简单的名为 Vaildate 的方法：

```
interface IValidate
{
    bool Validate();
}
```

现在您可以证明这样的事实——如果控件实现了 Ivalidate 接口，控件就可以生效了。

让我们仔细考虑一下上述代码片断的几个方面。第一，您并不需要在接口方法上指定一个访问限定符，例如 public。事实上，预先考虑访问限定符的方法声明会导致产生一个编译时的错误。这是因为按照定义，所有的接口方法都是 public 类型的(C++程序员可能也注意到，因为按照定义，接口是抽象类，所以不需要显式地利用 appending=0 把方法声明为 pure virtual)。

除了方法以外，接口还可以定义属性、索引器和事件，如下所示：

```
interface IExampleInterface
{
    // Example property declaration.
    int testProperty { get; }

    // Example event declaration.
    event testEvent Changed;

    // Example indexer declaration.
    string this[int index] { get; set; }
}
```

9.3 实现接口

因为接口定义了一个约定，任何实现一个接口的类都必须定义那个接口中的每一项条目，否则就无法编译代码。要想使用前面部分中的 IValidate 例子，一个客户端类只需要实现接口的方法就可以了。在下面的例子中，有一个名为 FancyControl 的基础类和一个名为 IValidate 的接口。还有一个类 MyControl，这个类从 FancyControl 类派生出来，并实现了 IValidate。请注意语法以及 MyControl 对象是如何被转换(cast)到 IValidate 接口来引用其成员的。

```
using System;

public class FancyControl
{
   protected string Data;
      public string data
      {
         get

      {
         return this.Data;
      }
      set
      {
         this.Data = value;
      }
   }
}

interface IValidate
{
   bool Validate();
}

class MyControl : FancyControl, IValidate
{
   public MyControl()
   {
      data = "my grid data";
   }
```

```
    public bool Validate()
    {
        Console.WriteLine("Validating...{0}", data);
            return true;
    }
}

class InterfaceApp
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        // Call a function to place the control on
        // the form. Now, when the editor needs to
        // validate the control, it can do
        // the following:

        IValidate val = (IValidate)myControl;
        bool success = val.Validate();
    Console.WriteLine("The validation of '{0}' was {1}successful",
                    myControl.data,
                    (true == success ? "" : "not "));
    }
}
```

使用上述类和接口，编辑器可以向控件查询其是否实现了 IValidate 接口(在下一节中，您将会了解如何实现这一步骤)。如果控件实现了 IValidate 接口，编辑器可以确认并调用所实现的接口方法。那么您可能会问，“为什么不能仅仅定义一个基础类来使用这个编辑器(这个基础类具有名为 Validate 的纯虚拟函数)？编辑器就会只接受从该基础类派生出来的控件，不是吗？”

这个方案也是可以正常工作的，但是它有很强的局限性。让我们假设您创建了自己的控件，并且这些控件都是从这个假想的基础类派生的。从而，这些控件都实现了这个 Validate 虚方法。这种方案一直都可以正常工作，直到有一天您发现有一个很强大的控件，您希望可以用来使用编辑器。假设您发现一个由其他人编写的网格(grid)。这样的话，该网格就无法从编辑器的强制控件基础类派生了。在 C++中，解决的方法是使用多继承，并且让您的网格同时继承第三方网格和编辑器的基础类。然而，C#并不支持多继承。

使用接口，您可以在单个类中实现多种行为特征。在 C#中，您可以从单个类中派生，并且，除了拥有那些通过继承得到的功能以外，还可以按照类的需要来实现任意多的接口。例如，如果您要编辑器应用程序对控件的内容进行有效性检查、把控件绑定到数据库上、

以及把控件的内容串行地输出到磁盘上，您就应该按照如下形式声明自己的类：

```
public class MyGrid : ThirdPartyGrid, IValidate,
                      ISerializable, IDataBound
{
...
}
```

正如前面所说的，下一节将回答这样的问题，“一段代码如何才能知道一个类何时实现一个给定的接口？”

9.3.1 使用 is 来查询实现

在 InterfaceApp 例子中，您已经看到了下面的代码，这些代码被用来将一个对象(MyControl)转换到这个类的一个已实现的接口(IValidate)，然后调用这些接口成员的其中一个(Validate)：

```
MyControl myControl = new MyControl();
...
IValidate val = (IValidate)myControl;
bool success = val.Validate();
```

如果一个客户端试图使用一个类，仿佛这个类已经实现了一个方法，但是实际上这个类并没有实现这个方法，这时会发生什么？下面的例子将可以编译，因为 Iserializable 是一个有效的接口。虽然如此，在运行的时候还是会抛出一个 System.InvalidCastException 异常，因为 MyGrid 并没有实现接口 Iserializable。这个应用程序然后就会终止，除非这个异常可以被显式地截获。

```
using System;

public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
```

```
    }
}

interface ISerializable
{
    bool Save();
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = "my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine("Validating...{0}", data);
        return true;
    }
}

class IsOperator1App
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        ISerializable ser = (ISerializable)myControl;

        // NOTE: This will throw a System.InvalidateCastException
        // because the class does not implement the ISerializable
        // interface.
        bool success = ser.Save();

        Console.WriteLine("The saving of '{0}' was {1}successful",
 myControl.data,
 (true == success ? "" : "not "));
    }
```

```
}
```

当然，截获这个异常并不能改变这样的事实，即您所希望执行的代码在这种情况下实际上无法执行。您所需要的是一种在试图转换对象之前先向其进行查询的方法。一种做法是，使用 is 操作符。is 操作符使您能够在运行时检查一种类型是否与另外一种类型相兼容。这个操作符采用以下格式，其中 expression 是一种引用类型：

expression is *type*

这个操作符产生一个布尔型的值，并且也因此可以在条件语句中使用。在下面的例子中，已经修改了代码，以便于在试图使用方法 ISerializable 之前先检验类 MyControl 和接口 Iserializable 之间的兼容性：

```
using System;

public class FancyControl
{
   protected string Data;
   public string data
   {
      get
      {
         return this.Data;
      }
      set
      {
         this.Data = value;
      }
   }
}

interface ISerializable
{
   bool Save();
}

interface IValidate
{
   bool Validate();
}

class MyControl : FancyControl, IValidate
{
   public MyControl()
   {
```

```
            data = "my grid data";
        }

        public bool Validate()
        {
            Console.WriteLine("Validating...{0}", data);
            return true;
        }
    }

    class IsOperator2App
    {
        public static void Main()
        {
            MyControl myControl = new MyControl();

            if (myControl is ISerializable)
            {
                ISerializable ser = (ISerializable)myControl;
                bool success = ser.Save();

                Console.WriteLine("The saving of '{0}' was {1}successful",
    myControl.data,
    (true == success ? "" : "not "));
            }
            else
            {
             Console.WriteLine("The   ISerializable   interface   is   not
implemented.");
            }
        }
    }
```

既然您已经看到 is 操作符是如何使您能够检验两种类型之间的兼容性，以确保用法正确的，那么让我们再来看一看 is 操作符的“近亲”——as 操作符——并且将这两个操作符加以比较。

9.3.2 使用 as 来查询实现

如果您仔细地观察由前面那个 IsOperator2App 例子而生成的 MSIL 代码——就是这一段下面的那些 MSIL 代码——您就会注意到一个与 is 操作符有关的问题。您可以发现紧接着该对象被分配并设置了堆栈之后就调用了 isinst。操作符 isinst 是编译器为 C# 操作符 is

而产生的一个操作码。这个操作码检测对象是否是一个类或接口的一个实例。请注意几行代码以后之处——假定条件检查通过了——编辑器生成了 castclass IL 操作码。castclass 操作码进行自己的检验，而且，尽管这个操作码的工作方式与 isinst 有一点不同，结果是所生成的 IL 作了一些低效率的工作，因为对于转换的检查进行了两次。

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       72 (0x48)
  .maxstack  4
  .locals (class MyControl V_0,
          class ISerializable V_1,
          bool V_2)
  IL_0000:  newobj    instance void MyControl::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  isinst    ISerializable
  IL_000c:  brfalse.s  IL_003d
  IL_000e:  ldloc.0
  IL_000f:  castclass  ISerializable
  IL_0014:  stloc.1
  IL_0015:  ldloc.1
  IL_0016:  callvirt   instance bool ISerializable::Save()
  IL_001b:  stloc.2
  IL_001c:  ldstr     "The saving of '{0}' was {1}successful"
  IL_0021:  ldloc.0
  IL_0022:     call                  instance   class   System.String
FancyControl::get_data()
  IL_0027:  ldloc.2
  IL_0028:  brtrue.s   IL_0031
  IL_002a:  ldstr     "not "
  IL_002f:  br.s      IL_0036
  IL_0031:  ldstr     ""
  IL_0036:   call     void  [mscorlib]System.Console::WriteLine(class
System.String,
                                                  class System.Object,
                                                  class System.Object)
  IL_003b:  br.s   IL_0047
  IL_003d:  ldstr  "The ISerializable interface is not implemented."
  IL_0042:    call    void  [mscorlib]System.Console::WriteLine(class
System.String)
  IL_0047:  ret
} // end of method IsOperator2App::Main
```

使用 as 操作符可以使检验处理更加有效率。as 操作符在可兼容的类型之间进行转换，并且采用如下格式，其中那个 expression 是任何一种引用类型：

object = *expression* as *type*

如果所讨论的两种类型是相兼容的话，您可以把 as 看作是 is 操作符和转换的组合。as 操作符和 is 操作符之间的一个重要不同就是，如果 expression 和 type 之间不兼容，as 操作符把 object 设定为等于 null，而不是返回一个布尔值。现在我们可以以一种更有效率的方式来重新编写这个例子：

```
using System;

public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface ISerializable
{
    bool Save();
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = "my grid data";
    }
```

```
        public bool Validate()
        {
            Console.WriteLine("Validating...{0}", data);
            return true;
        }
    }

    class AsOperatorApp
    {
        public static void Main()
        {
            MyControl myControl = new MyControl();

            ISerializable ser = myControl as ISerializable;
            if (null != ser)
            {
                bool success = ser.Save();

                Console.WriteLine("The saving of '{0}' was {1}successful",
    myControl.data,
    (true == success ? "" : "not "));
            }
            else
            {
              Console.WriteLine("The   ISerializable   interface   is   not
implemented.");
            }
        }
    }
```

现在，确保有效转换所作的检验只需要进行一次，很明显，这更有效率。在这一点上，让我们重新回到 MSIL 代码来看一看使用 as 操作符所带来的影响：

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       67 (0x43)
  .maxstack  4

  .locals (class MyControl V_0,
          class ISerializable V_1,
          bool V_2)
  IL_0000:  newobj     instance void MyControl::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
```

```
      IL_0007:  isinst    ISerializable
      IL_000c:  stloc.1
      IL_000d:  ldloc.1
      IL_000e:  brfalse.s  IL_0038
      IL_0010:  ldloc.1
      IL_0011:  callvirt   instance bool ISerializable::Save()
      IL_0016:  stloc.2
      IL_0017:  ldstr      "The saving of '{0}' was {1}successful"
      IL_001c:  ldloc.0
      IL_001d:     call                  instance   class   System.String
FancyControl::get_data()
      IL_0022:  ldloc.2
      IL_0023:  brtrue.s   IL_002c
      IL_0025:  ldstr      "not "
      IL_002a:  br.s       IL_0031
      IL_002c:  ldstr      ""
      IL_0031:   call      void  [mscorlib]System.Console::WriteLine(class
System.String,
                                                  class System.Object,
                                                  class System.Object)
      IL_0036:  br.s       IL_0042
      IL_0038:  ldstr      "The ISerializable interface is not implemented."
      IL_003d:   call      void  [mscorlib]System.Console::WriteLine(class
System.String)
      IL_0042:  ret
    } // end of method AsOperatorApp::Main
```

9.4 显式的接口成员名字限定

到目前为止，您已经见到，类在实现接口时指定一个访问限定符 public，紧接着后面的是接口方法名字。然而，有时候您希望(甚至是需要)明确地用接口的名字来限定成员的名字。在这一节里，我们将考虑这么做的两个理由。

9.4.1 接口的名字隐藏

调用一个从接口实现的方法的最常见做法是把那个类的一个实例转换为该接口类型，然后调用所需要的方法。尽管这种做法是有效的，而且许多人都使用这种技术，但是从技术角度来说，您并不一定要把对象转换到它所实现的接口上来调用这个接口的方法。这是因为当一个类实现了一个接口的方法，这些方法同时也就是该类的 public 方法。请看下面

的C#代码，特别是请注意Main方法，这样就会理解上述意思：

```
using System;

public interface IDataBound
{
    void Bind();
}

public class EditBox : IDataBound
{
    // IDataBound implementation.
    public void Bind()
    {
        Console.WriteLine("Binding to data store...");
    }
}

class NameHiding1App
{
    // Main entry point.
    public static void Main()
    {
        Console.WriteLine();

        EditBox edit = new EditBox();
        Console.WriteLine("Calling EditBox.Bind()...");
        edit.Bind();

        Console.WriteLine();

        IDataBound bound = (IDataBound)edit;
        Console.WriteLine("Calling (IDataBound)EditBox.Bind()...");
        bound.Bind();
    }
}
```

这个例子将会产生如下输出：

```
Calling EditBox.Bind()...
Binding to data store...
Calling (IDataBound))EditBox.Bind()...
Binding to data store...
```

请注意，虽然这个应用程序以两种不同的方式调用了已经实现的Bind方法——一种是

利用了转换，而另一种则没有使用转换——这两种方式都正确地调用了函数，因为 Bind 方法被处理了。虽然乍一看，能够直接调用所实现的方法而无需把对象转换到一个接口上，这种功能似乎是种优点，但是有时候这并不是我们所需要的。最为一种明显的原因是，几个不同接口的实现——每一个接口都可能含有大量的成员——可能很快就会使您的类的 public 名称空间遭到那些在实现类的作用域以外没有意义的成员的污染。您可以防止接口中被实现的成员成为该类的 public 成员，这要用到被称为名字隐藏(name hiding)的技术。

最为简单的名字隐藏就是，能够对那些除被派生类或实现类以外的任何代码(这些代码通常被称为外部世界(outside world))隐藏其所继承的成员名字。让我们假设有一个与我们前面所使用的例子相同的例子，其中一个 EditBox 类需要实现接口 IDataBound——然而，这次 EditBox 类并不想要把 IDataBound 方法暴露给外部世界。它只是为了实现自己的目的才需要使用该接口，或者仅仅可能是程序员不希望那些一般客户端不会用到的大量方法把类的名称空间搞混乱。为了隐藏一个被实现接口的成员，您只需要把成员的 public 访问限定符去掉，并且用接口的名字来限制成员的名字，如下面所示：

```
using System;

public interface IDataBound
{
    void Bind();
}

public class EditBox : IDataBound
{
    // IDataBound implementation.
    void IDataBound.Bind()
    {
        Console.WriteLine("Binding to data store...");
    }
}

class NameHiding2App
{
    public static void Main()
    {
        Console.WriteLine();

        EditBox edit = new EditBox();
        Console.WriteLine("Calling EditBox.Bind()...");

        // ERROR: This line won't compile because
        // the Bind method no longer exists in the
```

```
        // EditBox class's namespace.
        edit.Bind();

        Console.WriteLine();

        IDataBound bound = (IDataBound)edit;
        Console.WriteLine("Calling (IDataBound)EditBox.Bind()...");

        // This is OK because the object was cast to
        // IDataBound first.
        bound.Bind();
    }
}
```

上述代码将无法通过编译，因为成员名字 Bind 不再是 EditBox 类的一部分。因此，这种技术使您能够把成员从类的名称空间中去掉，同时仍然允许使用转换操作来进行显式的访问。

在这里需要重申的一点是，当您隐藏一个成员时，您不能使用访问限定符。如果您试图在一个被实现的接口成员上使用访问限定符，就会产生一个编译错误。可能您发现这有点奇怪，但是考虑到隐藏的原因就是为了防止对当前类的外部可见，也就不以为怪了。既然访问限定符的存在只是为了定义对于基础类外部的可见级别，您就可以明白在使用名字隐藏时，使用访问限定符是没有意义的了。

9.4.2 避免名字模糊性

C#不支持多继承的一个主要原因就在于名字冲突的问题，这是由名字模糊性造成的。虽然 C#并不在(从一个类的派生)对象一级支持多继承，但是它支持从一个类的继承，此外还支持多接口的实现。然而，这样的功能会带来一些代价：名字冲突。

在下面的例子中，我们有两个接口，*ISerializable* 和 *IdataStore*，这两个接口支持以两种不同格式进行读取和存储——一种是以二进制格式从对象到磁盘进行读取和存储，另一种是向一个数据库读取和存储。问题是这两个接口都含有名为 SaveData 的方法：

```
using System;

interface ISerializable
{
    void SaveData();
}

interface IDataStore
```

```
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    public void SaveData()
    {
        Console.WriteLine("Test.SaveData called");
    }
}

class NameCollisions1App
{
    public static void Main()
    {
        Test test = new Test();

        Console.WriteLine("Calling Test.SaveData()");
        test.SaveData();
    }
}
```

现在写出的这些代码确实可以通过编译。但是，我们已经得到通知，在以后所构造出来的C#编译器中，这样的代码会导致一个编译时的错误产生，因为所实现的方法SaveData是不明确的。无论这些代码是否可以编译，在运行时都会有问题产生，因为调用SavaData方法的行为结果对于使用这个类的程序员来说是不清楚的——您用的SaveData方法究竟是要串行化地把数据从对象保存到磁盘，还是要把数据保存到数据库呢？

此外，请看下面的代码：

```
using System;

interface ISerializable
{
    void SaveData();
}

interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
```

```
        public void SaveData()
        {
            Console.WriteLine("Test.SaveData called");
        }
    }

    class NameCollisions2App
    {
        public static void Main()
        {
            Test test = new Test();

            if (test is ISerializable)
            {
                Console.WriteLine("ISerializable is implemented");
            }

            if (test is IDataStore)
            {
                Console.WriteLine("IDataStore is implemented");
            }
        }
    }
```

在这段代码中，is 操作符对于这两个接口都是成功的，这就表示这两个接口都被实现了，但是我们知道事实并非如此。甚至连编译器在编译这个例子的代码时也给出了下面的警告：

```
NameCollisions2.cs(27,7): warning CS0183: The given expression is
always of the provided ('ISerializable') type
NameCollisions2.cs(32,7): warning CS0183: The given expression is
always of the provided ('IDataStore') type
```

问题在于这个类所实现的要么是 Bind 方法的串行化版本，要么是数据库版本(而不是同时实现两个版本)。然而，如果客户端检查其中一个接口的实现——两个接口的实现都是成功的——而同时又碰巧试图使用那个并没有被真正实现的那个接口，这时就会发生意外的结果。

您可以使用显式的成员名字限定，这样就可以解决这个问题：去掉访问限定符，并且预先在考虑使用成员名的时候——在这个例子里的成员名是SaveData——要附带上接口名字：

```
using System;
interface ISerializable
```

```
{
    void SaveData();
}

interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    void ISerializable.SaveData()
    {
        Console.WriteLine("Test.ISerializable.SaveData called");
    }
    void IDataStore.SaveData()
    {
        Console.WriteLine("Test.IDataStore.SaveData called");
    }
}

class NameCollisions3App
{
    public static void Main()
    {
        Test test = new Test();

        if (test is ISerializable)
        {
            Console.WriteLine("ISerializable is implemented");
            ((ISerializable)test).SaveData();
        }

        Console.WriteLine();

        if (test is IDataStore)
        {
            Console.WriteLine("IDataStore is implemented");
            ((IDataStore)test).SaveData();
        }
    }
}
```

现在这里没有关于哪个方法被调用的不明确问题了。这两个方法都用完整的限定名实

现了，而且程序输出的结果也正是您所预期的：

```
ISerializable is implemented
Test.ISerializable.SaveData called
IDataStore is implemented
Test.IDataStore.SaveData called
```

9.5 接口和继承

有两个常见问题与接口和继承有关。第一个问题涉及的是从一个基础类派生的事项，这个类中含有的方法名与该类所需要实现的接口方法名完全相同，我们用一个例子来演示这一点：

```
using System;

public class Control
{
    public void Serialize()
    {
        Console.WriteLine("Control.Serialize called");
    }
}

public interface IDataBound
{
    void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh1App
{
    public static void Main()
    {
        EditBox edit = new EditBox();
        edit.Serialize();
    }
}
```

正如您已经知道的，要想实现一个接口，您必须在该接口的声明中为每个成员提供定

义。然而，在上述例子中，我们并没有这么做，而这段代码居然也通过编译了！能够通过编译的原因在于C#要在类EditBox中查找一个已经实现的Serialize方法，而且也找到了。然而，编译器所作出的判断是不正确的，它所找到的并不是一个已经实现的方法。编译器所找到的Serialize方法是从类Control类继承来的Serialize方法，并不是所需要实现的IDataBound.Serialize方法。因此，虽然代码通过了编译，但是并不能按照预期的那样工作，正如我们接下来将看到的。

现在我们更进一步看一下该问题。请注意，下面的代码首先通过as操作符来检查接口是否被实现，然后试着调用被实现的Serialize方法。代码可以通过编译并运行。然而，正如我们所知道的，类EditBox并没有真正地实现一个Serialize方法作为IDataBound继承的结果。EditBox已经具有一个从Control类继承的Serialize方法。这就意味着客户端可能会得到一些意外的结果。

```
using System;

public class Control
{
    public void Serialize()
    {
        Console.WriteLine("Control.Serialize called");
    }
}

public interface IDataBound
{
    void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh2App
{
    public static void Main()
    {
        EditBox edit = new EditBox();

    IDataBound bound = edit as IDataBound;
    if (bound != null)
    {
        Console.WriteLine("IDataBound is supported...");
        bound.Serialize();
```

```
        }

        else
        {
            Console.WriteLine("IDataBound is NOT supported...");
        }
        }
    }
```

当某个被派生的类中有一个方法，这个方法的名字与基础类实现的一个接口方法名字相同时，可能会发生另外一个问题。让我们看一看下面的代码：

```
using System;

interface ITest
{
    void Foo();
}

// Base implements ITest.
class Base : ITest
{
    public void Foo()
    {
        Console.WriteLine("Base.Foo (ITest implementation)");
    }
}

class MyDerived : Base
{
    public new void Foo()
    {
        Console.WriteLine("MyDerived.Foo");
    }
}

public class InterfaceInh3App
{
    public static void Main()
    {
        MyDerived myDerived = new MyDerived();
        myDerived.Foo();

        ITest test = (ITest)myDerived;
        test.Foo();
```

```
    }
}
```

这段代码在屏幕上产生以下输出：

```
MyDerived.Foo
Base.Foo (ITest implementation)
```

在这个例子中，Base 类实现了 Itest 接口和这个接口的 Foo 方法。然而，MyDerived 类从 Base 类派生出一个新类，并且也为新的类实现了一个新的 Foo 方法。那么哪一个 Foo 方法才会被调用呢？这取决于您有哪一个引用。如果您有一个指向 MyDerived 对象的引用，那么就会调用 MyDerived 对象的 Foo 方法。这是因为即使 MyDerived 对象实现了一个继承来的 ITest.Foo 方法，运行时也会执行 MyDerived.Fol 方法，因为关键字 new 指定了一个被继承方法的覆盖。

然而，当您显式地把对象 myDerived 转换到接口 ITest 上时，编译器要判断接口是否已经实现。MyDerived 类有一个同名方法，但是这个同名方法并不是编译器所要找的那个方法。当您把一个对象转换到一个接口时，编译器遍历继承树，直到找到一个类，这个类在其基本列中含有该接口。这就是为什么 Main 方法的最后两行代码会调用 ITest 所实现的 Foo 方法的原因。

现在，这些潜在问题中的一些，包括名字冲突和接口继承，已经支持了以下建议的正确性：永远都要把对象转换到您所要使用的成员变量所在的那个接口。

9.6 合并接口

C#的另一个强大的功能就是，能够把两个或以上的接口合并到一起，这样一个类只要实现合并后的接口就可以了。例如，让我们假设您想创建一个新的 TreeView 类，这个类既要实现 IDragDrop 接口，也要实现 ISortable 接口。我们可以假设其他控件，例如 ListView 和 ListBox，可能也想把这些特性合并起来。因为这样的假想是合理的，所以您可能希望把 IDragDrop 接口和 Isortable 接口合并成一个接口：

```
using System;

public class Control
{
}

public interface IDragDrop
{
    void Drag();
```

```
    void Drop();
}

public interface ISerializable
{
    void Serialize();
}

public interface ICombo : IDragDrop, ISerializable
{
    // This interface doesn't add anything new in
    // terms of behavior as its only purpose is
    // to combine the IDragDrop and ISerializable
    // interfaces into one interface.
}

public class MyTreeView : Control, ICombo
{
    public void Drag()
    {
        Console.WriteLine("MyTreeView.Drag called");
    }

    public void Drop()
    {
        Console.WriteLine("MyTreeView.Drop called");
    }

    public void Serialize()
    {
        Console.WriteLine("MyTreeView.Serialize called");
    }
}

class CombiningApp
{
    public static void Main()
    {
        MyTreeView tree = new MyTreeView();
        tree.Drag();
        tree.Drop();
        tree.Serialize();
    }
```

利用合并接口的功能，您就不仅可以简化把语义上相关的接口聚集在一个接口中的能

力，还可以在需要的时候，向新的“复合”接口添加方法。

9.7 本章小结

C#中的接口允许开发出这样的一些类，这些类之间可以共享特性，但是这些类并不是同一个类层次关系中的一部分。接口在 C#的开发中起着特殊的作用，因为 C#并不支持多继承。为了共享语义上相关的方法和属性，类可以实现多个接口。同样，is 操作符和 as 操作符可以被用来判断一个特定的接口是否已经被对象所实现了，这可以有助于防止发生与使用接口成员有关的错误。最后，显式的成员命名和名字隐藏可以用来控制接口的实现，并可以防止错误的发生。

第III部分　编写代码

第 10 章　表达式和操作符

在这一章内，我们将学习对任何编程语言都是最为基本的部分：通过使用操作符来表示赋值和比较的能力。我们将看一看在 C#中有什么样的操作符以及操作符的优先级是怎样的，此外我们将深入研究几类特定的表达式，这些表达式可以执行数学运算、赋值以及在操作数之间进行比较等。

10.1　已定义的操作符

操作符就是一种符号，这种符号表示对一个或多个参数值执行的某种操作。这种操作会产生一个结果。使用操作符的语法与进行方法调用的语法有些不同，但是 C#中，使用操作符的表达式的格式对您来说也应该是自然的。就像许多其他编程语言中的操作符一样，C#中操作符的语义也遵循着我们还是孩子的时候在数学课上所学到的那些规则和概念。最基本的 C#操作符包括有乘号(*)、除号(/)、加号和一元加号(+)、减号和一元减号(—)、取模号(%)以及赋值号(=)。

操作符经过了特别的设计，以用来从被操作的值上产生一个新的值。这些被操作的值被称为操作数。一个操作的结果必须被保存在内存的某个地方。在有些情况下，一个操作所产生的结果值被保存在含有原来操作数的变量里。如果您使用了一个操作符，而又没有新值可以被判断或保存，C#编译器就会给出一条错误消息。在下面的例子中，有一部分代码所进行的操作没有做出任何改变。编译器给出了错误消息，因为编写一个数学表达式，而这个表达式甚至没有对任何一个值做出改变，对于开发人员来说这样的做法通常都是一种错误。

```
class NoResultApp
{
    public static void Main()
    {
        int i;
```

```
            int j;

            i + j; // Error because call is not assigned to a variable.
        }
    }
```

绝大多数操作符都只能使用数字的数据类型，例如 Byte、Short、Long、Integer、Single、Double、以及 Decimal 等。在比较操作符(==和!=)和赋值操作符中有一些例外，这些操作符也可以与对象一起使用。此外，C#为 string 类定义了+和+=操作符，并且甚至允许在独一无二的语言结构上，例如代表(delegate)，使用增量操作符(++)和(--)。最后一个例子将在第 14 章中进行深入讨论。

10.2 操作符的优先级

当一个表达式或者语句中含有多个操作符时，编译器会判断对不同操作符进行求值的顺序。编译器做出这种判断时所遵循的规则被称为操作符优先级(operator precedence)。在语句的运行结果与所设想结果不同的时候，理解操作符优先级有助于您弄明白编写表达式与写一行代码之间的区别。

请看下面的表达式：42+6*10。 如果您用 42 加上 6，然后再乘以 10，结果就是 480。如果您用 6 乘以 10，然后再加上 42，结果就是 102。编译器有一个特殊的部分，这部分被称作词法分析器，当代码被编译时，词法分析器负责判断如何读取这段代码。当一个表达式中有多种不同类型的操作符时，正是词法分析器来判断操作符之间的相对优先级。词法分析器是通过为每一个支持的操作符指定一个值(或优先级)来完成这一工作的。高优先级的操作符先执行。在我们的例子中，操作符*的优先级比操作符+的优先级要高，因为*操作符在+操作符之前得到操作数。*比+优先的原因来自于基本数学规则：先乘除，后加减。因此，回到这个例子中来：在 42+6*10 和 42*6+10 这两个表达式中，6 都是被*操作符所采用，这样，这两个表达式就分别等价于 42+(6*10)以及(42*6)+10。

10.2.1 C#如何判断优先级

让我们来看一看 C#是如何把优先级赋给操作符的。表 10.1 从高到低显示了 C#操作符的优先级。学完这一节后，我们将对 C#中支持的各种不同种类操作符的细节作进一步的了解。

表 10.1　C# 操作符优先级

操作符类别	操作符
初级操作符	(x), x.y, f(x), a[x], x++, x--, new, typeof, sizeof,checked, unchecked
一元操作符	+, -, !, ~, ++x, --x, (T)x
乘除操作符	*, /, %
加减操作符	+, -
位移操作符	<<、>>
关系操作符	<, >, <=, >=, is
等于操作符	==
逻辑与	&
逻辑异或	^
逻辑或	|
条件与	&&
条件或	||
条件操作符	?:
赋值操作符	=, *=, /=, %=, +=, -=, <<=、>>=, &=, ^=, |=

10.2.2　左联合性和右联合性

联合性(Associativity)决定了表达式的哪一边先被求值。作为一个例子，下面的表达式根据一操作符的左联合性或右联合性，其结果分别为 21 或 33：

```
42-15-6
```

操作符一被定义为左联合，这意味着先计算 42－15，然后再用计算结果减去 6。如果操作符一被定义为右联合，那么操作符右边的表达式就先被计算：先计算 15－6，然后用 42 减去计算出的结果。

所有的二元操作符——也即带有两个操作数的操作符——除了赋值操作符以外，都是左联合的。这就意味着，表达式是从左到右求值的。因此，a+b+c 与(a+b)+c 是相同的，其中都是先对 a+b 进行求值，然后再把 c 加到前面的求值结果上。赋值操作符和条件操作符是右联合的——表达式从右到左求值。换句话说，a=b=c 等价于 a=(b=c)。那些想要

在一行上使用多个赋值操作符的人往往会在这里犯错，所以让我们来看一些代码：

```
using System;

class RightAssocApp
{
    public static void Main()
    {
        int a = 1;
        int b = 2;
        int c = 3;

        Console.WriteLine("a={0} b={1} c={2}", a, b, c);
        a = b = c;
        Console.WriteLine("After 'a=b=c': a={0} b={1} c={2}", a, b, c);
    }
}
```

运行这个例子，所得到的结果如下所示：

```
a=1 b=2 c=3
After 'a=b=c' : a=3 b=3 c=3
```

看到一个表达式从右开始计算，这在一开始可能有些让人迷惑，但是请这样去想：如果赋值操作符是左联合的，编译器就会先对 a=b 求值，这会把一个值 2 赋给 a，然后编译器会计算 b=c，这会把值 3 赋给 b。最后的结果就是 a=2b=3c=3。不用多说，这样的表达式不会产生我们所预期的 a=b=c 的结果，而这正是为什么赋值和条件操作符都是左联合的原因。

10.2.3 实际使用

没有什么比花了很长时间去追踪程序中的一个错误，最后才发现这是由于开发人员不懂优先级或者联合性规则而导致的更倒霉了。作者在邮件列表上看到，有些聪明人已经建议养成一种编程习惯，在编写表达式时使用空格来指示出他们所认为具有的操作符优先级——一种自我确定(self-documenting)机制。这样的话，例如，因为我们知道乘除操作符的优先级比加减操作符的优先级高，所以我们可以编写类似于下面所示的代码，其中表达式的空格暗示了想要实现的优先级。

```
a = b*c +d;
```

这种方法有着严重的缺陷——如果没有特定的语法的话，编译器是无法正确地解析代码的。编译器根据开发编译器的人员所决定的规则来解析表达式。为了能够表示优先级，

语言中存在一个符号，这个符号允许您设定优先级和联合性：圆括号。例如，您可以把表达式 a=b*c+d 改写成 a=(b*c)+d 或者是 a=b*(c+d)，而编译器会先对圆括号内的操作符求值。如果表达式中存在着两个或更多的圆括号对，那么编译器就会根据在前面已经介绍过的优先级和联合性规则，先求出每对括号中的值，然后再求出整个语句的值。

作者认为，当一个表达式中有多个操作符时，您应该使用圆括号。即使您知道优先级的顺序，也建议您这么做，因为维护您所编写的代码的人可能并不是很熟悉这些优先级规则。

10.3　C#操作符

按照操作符的优先级来考虑操作符是有益的。在这一节里，我们将按照前面表 10.1 中列出的操作符顺序来介绍最为常用的操作符。

10.3.1　初级表达式操作符

我们要介绍的第一类操作符是初级表达式操作符。因为这些操作符中的绝大部分都是一些相当基础的操作符，所以列出这些操作符并对其功能作一些简短的介绍。然后对其他一些不太常用的操作符作一些解释。

- **(X)**　这种形式的圆括号操作符被用来控制优先级，要么是在数学操作中使用，要么是在方法调用中使用。
- **X.Y**　点操作符被用来指定类或结构的一个成员，其中 x 代表含有 y 的实体，而 y 代表成员。
- **f(x)**　这种形式的圆括号操作符被用来列出一个方法的参数。
- **a[x]**　方括号用在数组的索引中。方括号也与索引器一起使用，其中可以把对象当作数组来对待。要想了解更多关于索引器的信息，请参阅第 7 章。
- **x++**　因为这里并没有更多的讨论，我们将在这章的后面“增量操作符和减量操作符”中介绍增量操作符。
- **x--**　减量操作符也将在本章的后面加以介绍。
- **new**　操作符 new 被用来根据类的定义实例化一个对象。

10.3.1.1　typeof

“反射”(reflection)是一种在运行时查找类型信息的功能。这些信息包括类型的名字、类的名字以及结构的成员。在.NET 框架中构成这种功能所需要的是一个名为 System.Type 的类。这个类是所有反射操作符的根类，而且我们可以通过使用操作符 typeof 来获得这个

类。在这里，我们不过于详细地讨论反射——这将在第 16 章中进行介绍——但是，这里有一个简单的例子，演示在运行时使用 typeof 操作符查找您想要的关于类型或对象的几乎任何信息是多么容易：

```
using System;
using System.Reflection;

public class Apple
{
    public int nSeeds;
    public void Ripen()
    {
    }
}

public class TypeOfApp
{
    public static void Main()
    {
        Type t = typeof(Apple);
        string className = t.ToString();

        Console.WriteLine("\nInformation about the class {0}",
            className);

        Console.WriteLine("\n{0} methods", className);
        Console.WriteLine("----------------------------");
        MethodInfo[] methods = t.GetMethods();
        foreach (MethodInfo method in methods)
        {
            Console.WriteLine(method.ToString());
        }

        Console.WriteLine("\nAll {0} members", className);
        Console.WriteLine("----------------------------");
        MemberInfo[] allMembers = t.GetMembers();
        foreach (MemberInfo member in allMembers)
        {
            Console.WriteLine(member.ToString());
        }
    }
}
```

程序中含有一个名为 Apple 的类，这个类中只有两个成员：一个成员变量名为 nSeeds，

另一个方法名为 Ripen。我们要做的第一件事就是使用 typeof 操作符和类名来得到一个 System.Type 对象，然后把这个对象保存在一个名为 t 的变量中。至此，我们可以使用 System.Type 方法来获得 Apple 类的所有方法和成员。我们分别使用方法 GetMethods 和 GetMembers 来执行这些任务。这些方法调用的结果被打印成标准输出，如下所示：

```
Information about the class Apple
Apple methods
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Ripen()
System.Type GetType()
All Apple members
Int32 nSeeds
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Ripen()
System.Type GetType()
Void .ctor()
```

在我们继续下去之前，还有两点需要提醒您注意。第一，请注意类中那些继承来的成员也在输出中列了出来。因为并没有显式地从另一个类派生这个类，我们知道，Apple 类中所有未被定义的成员都是从隐式的基础类 System.Object 中继承来的。第二，请注意，您也可以使用方法 GetType 来查找 System.Type 对象。这个方法是从 System.Object 中继承的，使您能够使用对象。下面的两段代码可以被交替地使用来查询 System.Type 对象。

```
// Retrieve the System.Type object from the class definition.
Type t1 = typeof(Apple);
// Retrieve the System.Type object from the object.
Apple apple = new Apple();
Type t2 = apple.GetType();
```

10.3.1.2 sizeof

操作符 sizeof 可以被用来得到给定类型的大小，以字节来表示。当使用这个操作符时，您应该记住十分重要的两点。第一，您只可以对值类型使用 sizeof 操作符。因此，虽然这个操作符可以在类的成员上使用，但是它不可以在类自身上使用。第二，这个操作符只可以在标记为 unsafe 的方法或代码段中使用。我们将在第 17 章中深入地讨论非安全代码。这里有一个例子，在这个例子中，标记为 unsafe 的方法使用了 sizeof 操作符。

```
using System;
class BasicTypes
```

```
{
    // NOTE: You must declare that code that uses the sizeof
    // operator as unsafe.
    static unsafe public void ShowSizes()
    {
        Console.WriteLine("\nBasic type sizes");
        Console.WriteLine("sizeof short = {0}", sizeof(short));
        Console.WriteLine("sizeof int = {0}", sizeof(int));
        Console.WriteLine("sizeof long = {0}", sizeof(long));
        Console.WriteLine("sizeof bool = {0}", sizeof(bool));
    }
}
class Unsafe1App
{
    unsafe public static void Main()
    {
        BasicTypes.ShowSizes();
    }
}
```

运行这个程序所得到的结果如下所示：

```
Basic type sizes
sizeof short = 2
sizeof int = 4
sizeof long = 8
sizeof bool = 1
```

除了一些简单的内置类型以外，操作符 sizeof 还可以被用来判断用户定义的类型的大小，例如结构体。然而，sizeof 操作符的结果有时候并不明显，例如下面这个例子所示：

```
// Using the sizeof operator.
using System;

struct StructWithNoMembers
{
}

struct StructWithMembers
{
    short s;
    int i;
    long l;
    bool b;
}
```

```
struct CompositeStruct
{
    StructWithNoMembers a;
    StructWithMembers b;

    StructWithNoMembers c;
}

class UnSafe2App
{
   unsafe public static void Main()
   {
    Console.WriteLine("\nsizeof StructWithNoMembers structure = {0}",
                      sizeof(StructWithNoMembers));
    Console.WriteLine("\nsizeof StructWithMembers structure = {0}",
                      sizeof(StructWithMembers));
    Console.WriteLine("\nsizeof CompositeStruct structure = {0}",
                      sizeof(CompositeStruct));
   }
}
```

您可能以为这个应用程序对于没有成员的结构体(StructWithNoMembers)会打印出值 0，对于含有 4 个基本成员的结构体(StructWithMembers)打印出值 15，对于含有这两个结构体的复合结构体(CompositeStruct)会打印出值 15，但是实际结果如下所示：

```
sizeof StructWithNoMembers structure = 1
sizeof StructWithMembers structure = 16
sizeof CompositeStruct structure = 24
```

原因在于填充和结构的排列，这与编译器为一个结构体在输出文件图像中设定的布局有关。例如，如果一个结构体的长度是 3 个字节，而字节的排列是设定为每 4 个一组，编译器就会自动向其中填充一个额外的字节，而 sizeof 操作符会报告这个结构体的长度为 4 个字节。因此，在 C# 中，当判断一个结构的大小时需要仔细地考虑。

10.3.1.3　checked 和 unchecked

这两个操作符可以控制数学操作的溢出检查。因为这两个操作符与错误处理特别相关，所以我们将在第 12 章中对此加以详细介绍。

10.3.2　数学操作符

C# 语言支持几乎所有编程语言都支持的基本数学操作符：乘(*)、除(/)、加(+)、减(-)

和取模(%)。前四个操作符的意义很明显；取模操作符产生的是整数除法的余数。下面的代码演示了如何使用这些数学操作符：

```
using System;

class MathOpsApp
{
    public static void Main()
    {
        // The System.Random class is part of the .NET
        // Framework class library. Its default constructor
        // seeds the Next method using the current date/time.
        Random rand = new Random();
        int a, b, c;

        a = rand.Next() % 100; // Limit max to 99.
        b = rand.Next() % 100; // Limit max to 99.

        Console.WriteLine("a={0} b={1}", a, b);

        c = a * b;
        Console.WriteLine("a * b = {0}", c);

        // Note the following code uses integers. Therefore,
        // if a is less than  b, the result will always
        // be 0. To get a more accurate result, you would
        // need to use variables of type double or float.
        c = a / b;
        Console.WriteLine("a / b = {0}", c);

        c = a + b;
        Console.WriteLine("a + b = {0}", c);

        c = a - b;
        Console.WriteLine("a - b = {0}", c);

        c = a % b;
        Console.WriteLine("a % b = {0}", c);
    }
}
```

10.3.2.1 一元操作符

这里有两种一元操作符：加和减。一元减操作符向编译器指出操作数应该变为负。因

此，下面的 a 应该等于-42：

```
using System;

using System;

class Unary1App
{
    public static void Main()
    {
        int a = 0;

        a = -42;
        Console.WriteLine("{0}", a);
    }
}
```

然而，下面的代码中有一些含糊不清的地方：

```
using System;

class Unary2App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = 42;

        a = b * -c;
        Console.WriteLine("{0}", a);
    }
}
```

表达式 a=b*—c 令人感到有点迷惑。同样，使用圆括号有助于使这行代码更加清楚：

```
// With parentheses, it's obvious that you're multiplying
// b by the negative of c.
a = b * (-c);
```

您可能会认为，既然一元减返回操作数的负值，那么一元加就会返回操作数的正值。然而，一元加操作符除了返回操作数的原先值以外什么都不做，因此也就对操作数没有影响。例如，下面的代码会产生一个输出值-84：

```
using System;
```

```
class Unary3App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = -42;

        a = b * (+c);
        Console.WriteLine("{0}", a);
    }
}
```

要想得到操作数的正数值形式，要使用函数 Math.Abs。下面的代码产生的结果是 84：

```
using System;

class Unary4App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = -42;
        a = b * Math.Abs(c);
        Console.WriteLine("{0}", a);
    }
}
```

此处要简单提到的最后一个操作符是(T)操作符。这是圆括号操作符的一种形式，可以把一个类型转换为另一种类型。因为这个操作符可以被创建用户定义转换的功能所重载，所以我们将在第 13 章中对此加以介绍。

10.3.2.2 复合赋值操作符

复合赋值操作符指的是一个二元操作符和一个赋值操作符(=)的组合。它的语法是：

x *op*=y

其中 op 表示操作符。请注意，rvalue 并没有被 lvalue 值所取代，相反，复合表达式等效于下面的表达式：

x=x *op* y

使用 lvalue 作为操作结果的基础。

请注意，此处用了“等效于”这个字眼。编译器并不是逐字逐句地把类似于 x+=5

这样的表达式翻译成 x=x+5。它只是逻辑上按照这种方式进行工作。事实上，当 lvalue 是一个方法时，使用这样的操作会产生警告。现在让我们来仔细看一看下面的代码：

```
using System;

class CompoundAssignment1App
{
    protected int[] elements;
    public int[] GetArrayElement()
    {
        return elements;
    }

    CompoundAssignment1App()
    {
        elements = new int[1];
        elements[0] = 42;
    }

    public static void Main()

    {
        CompoundAssignment1App app = new CompoundAssignment1App();

        Console.WriteLine("{0}", app.GetArrayElement()[0]);
        app.GetArrayElement()[0] = app.GetArrayElement()[0] + 5;
        Console.WriteLine("{0}", app.GetArrayElement()[0]);
    }
}
```

请注意代码中粗体标出的一行代码——对于 CompoundAssignment1App.GetArrayElements 方法的调用以及进而对其第一个元素的修改——在这行代码中使用了赋值语法：

x=x *op* y

下面是生成的 MSIL 代码：

```
// Inefficient technique of x = x op y.

.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       79 (0x4f)
  .maxstack  4
  .locals (class CompoundAssignment1App V_0)
  IL_0000:  newobj     instance void CompoundAssignment1App::.ctor()
```

```
        IL_0005:  stloc.0
        IL_0006:  ldstr     "{0}"
        IL_000b:  ldloc.0
        IL_000c:        call                          instance    int32[]
CompoundAssignment1App::GetArrayElement()
        IL_0011:  ldc.i4.0
        IL_0012:  ldelema    ['mscorlib']System.Int32
        IL_0017:  box       ['mscorlib']System.Int32
        IL_001c:  call      void ['mscorlib']System.Console::WriteLine
      (class System.String,
       class System.Object)
        IL_0021:  ldloc.0
        IL_0022:  call
      instance int32[] CompoundAssignment1App::GetArrayElement()
        IL_0027:  ldc.i4.0
        IL_0028:  ldloc.0
        IL_0029:  call
      instance int32[] CompoundAssignment1App::GetArrayElement()
        IL_002e:  ldc.i4.0
        IL_002f:  ldelem.i4
        IL_0030:  ldc.i4.5
        IL_0031:  add
        IL_0032:  stelem.i4
        IL_0033:  ldstr     "{0}"
        IL_0038:  ldloc.0
        IL_0039:  call
      instance int32[] CompoundAssignment1App::GetArrayElement()
        IL_003e:  ldc.i4.0
        IL_003f:  ldelema    ['mscorlib']System.Int32
        IL_0044:  box       ['mscorlib']System.Int32
        IL_0049:  call  void ['mscorlib']System.Console::WriteLine
                                (class System.String, class System.Object)
        IL_004e:  ret
      } // end of method. 'CompoundAssignment1App::Main'
```

请看 MSIL 中用粗体标出的几行，您可以发现实际上 CompoundAssignment1App.GetArrayElements 方法被调用了两次！这种情况至少会导致低效率。在最坏的情况下，这甚至可能是灾难性的，这取决于方法所做的其他操作。

现在来看一看下面的代码，并请注意，我们把赋值改为使用复合赋值操作符语法：

```
using System;

class CompoundAssignment2App
{
```

```
        protected int[] elements;
        public int[] GetArrayElement()
        {
            return elements;
        }

        CompoundAssignment2App()
        {
            elements = new int[1];
            elements[0] = 42;
        }

        public static void Main()
        {
            CompoundAssignment2App app = new CompoundAssignment2App();

            Console.WriteLine("{0}", app.GetArrayElement()[0]);
            app.GetArrayElement()[0] += 5;
            Console.WriteLine("{0}", app.GetArrayElement()[0]);
        }
    }
```

使用复合赋值操作符产生了更有效率的 MSIL 代码，如下所示：

```
    // More efficient technique of x op= y.

    .method public hidebysig static void Main() il managed
    {
      .entrypoint
      // Code size       76 (0x4c)
      .maxstack  4
      .locals (class CompoundAssignment1App V_0, int32[] V_1)
      IL_0000:  newobj     instance void CompoundAssignment1App::.ctor()
      IL_0005:  stloc.0
      IL_0006:  ldstr      "{0}"
      IL_000b:  ldloc.0
      IL_000c:                    call          instance          int32[]
CompoundAssignment1App::GetArrayElement()
      IL_0011:  ldc.i4.0
      IL_0012:  ldelema    ['mscorlib']System.Int32
      IL_0017:  box        ['mscorlib']System.Int32
      IL_001c:  call   void ['mscorlib']System.Console::WriteLine
                                (class System.String, class System.Object)
      IL_0021:  ldloc.0
      IL_0022:                    call                instance        int32[]
CompoundAssignment1App: :GetArrayElement()
```

```
        IL_0027:  dup
        IL_0028:  stloc.1
        IL_0029:  ldc.i4.0
        IL_002a:  ldloc.1

      IL_002b:  ldc.i4.0
      IL_002c:  ldelem.i4
      IL_002d:  ldc.i4.5
      IL_002e:  add
      IL_002f:  stelem.i4
      IL_0030:  ldstr     "{0}"
      IL_0035:  ldloc.0
      IL_0036:            call                        instance      int32[]
CompoundAssignment1App::GetArrayElement()
      IL_003b:  ldc.i4.0
      IL_003c:  ldelema    ['mscorlib']System.Int32
      IL_0041:  box       ['mscorlib']System.Int32
      IL_0046:  call  void ['mscorlib']System.Console::WriteLine
                                  (class      System.String,      class
System.Object)
      IL_004b:  ret
    } // end of method 'CompoundAssignment1App::Main'
```

我们可以发现其中使用了 MSIL dup 操作码。dup 操作码复制堆栈的顶端元素，因而得到了一个复制来的值，这个值是从对 CompoundAssignment1App.GetArrayElements 方法的调用中检索到的。

这个练习的意义在于向我们展示了：虽然从概念上来说 x＋＝y 是与 x＝x＋y 等价的，但是在所生成的 MSIL 中还是有一些微妙的区别。这种区别使您必须仔细考虑在每种环境中使用哪一种语法。一个基本的规则，同时也是作者的建议，就是尽可能地使用复合赋值操作符。

10.3.2.3 增量操作符和减量操作符

增量操作符和减量操作符一开始是在 C 语言中引入的，然后被 C++和 Java 所继续沿用，这两个操作符允许您更加简明地表述这样的操作，即要把一个代表数字值的变量增加 1 或者是减小 1。因此，i＋＋就等价于把 i 的当前值加上 1。

增量操作符和减量操作符现在都有两个版本，这常常会引起冲突。增量操作符和减量操作符的这两种类型通常被称为前缀(prefix)和后缀(postfix)，这指出了在何时修改变量。当使用增量操作符和减量操作符的前缀版本时——分别是＋＋a 和－－a——程序就先执行操作，然后再产生值。当使用增量操作符和减量操作符的后缀版本时——分别是 a＋＋和 a－－——程序就先产生值，然后再执行操作。请看下面的例子：

```
using System;

class IncDecApp
{
    public static void Foo(int j)
    {
        Console.WriteLine("IncDecApp.Foo j = {0}", j);
    }

    public static void Main()
    {
        int i = 1;

        Console.WriteLine("Before call to Foo(i++) = {0}", i);
        Foo(i++);
        Console.WriteLine("After call to Foo(i++) = {0}", i);

        Console.WriteLine("\n");

        Console.WriteLine("Before call to Foo(++i) = {0}", i);
        Foo(++i);
        Console.WriteLine("After call to Foo(++i) = {0}", i);
    }
}
```

这个程序产生如下结果：

```
Before call to Foo(i++) = 1
IncDecApp.Foo j = 1
After call to Foo(i++) = 2
Before call to Foo(++i) = 2
IncDecApp.Foo j = 3
After call to Foo(++i) = 3
```

区别在于何时产生值以及何时修改操作数。在对 Foo(＋＋i)的调用中，(未被改变的)i 被传递给 Foo，并且在 Foo 方法返回以后，i 才被增加。您可以在下面的 MSIL 摘录中看到这一点。请注意，MSIL 操作符 add 直到值被放入堆栈中以后才被调用。

```
IL_0013: ldloc.0
IL_0014: dup
IL_0015: ldc.i4.1
IL_0016: add
IL_0017: stloc.0
IL_0018: call void IncDecApp::Foo(int32)
```

现在让我们来看一看在调用 Foo(++a)中用到的前增量(preincrement)操作符。在这里例子中，生成的 MSIL 与下面所示的代码相类似。请注意，MSIL 操作符 add 先被调用，然后才把值放在堆栈中，以便于后来对 Foo 方法进行调用。

```
IL_0049: ldloc.0
IL_004a: ldc.i4.1
IL_004b: add
IL_004c: dup
IL_004d: stloc.0
IL_004e: call void IncDecApp::Foo(int32)
```

10.3.3 关系操作符

绝大多数操作符都返回一个数字结果。但是关系操作符有所不同，因为关系操作符产生的是布尔型结果。关系操作符并不是在一组操作数上执行一些数学操作，而是比较操作数之间的关系，并且当关系为真时返回 true，关系为假时返回 false。

10.3.3.1 比较操作符

关系操作符中一组被称为比较操作符的操作符包括：小于(<)、小于或等于(<=)、大于(>)、大于或等于(>=)、等于(==)以及不等于(!=)。在处理数字的时候，这些比较操作符每一个意思都很明显，但是每个比较操作符如何处理对象就不是很明显了。这里有一个例子：

```
using System;

class NumericTest
{
    public NumericTest(int i)
    {
        this.i = i;
    }

    protected int i;
}

class RelationalOps1App
{
    public static void Main()
    {
        NumericTest test1 = new NumericTest(42);
        NumericTest test2 = new NumericTest(42);

        Console.WriteLine("{0}", test1 == test2);
```

```
    }
}
```

如果您是一个 Java 程序员，您就会知道将要发生什么。但是，大多数 C++开发人员可能会很惊讶地发现这个程序打印一个 false 语句。当您实例化一个对象时，请记住您所得到的只是指向由堆分配的对象的一个引用。因此，当您使用一个关系操作符来比较两个对象的时候，C#编译器并不比较这两个对象的内容。相反，编译器比较的是这两个对象的地址。再一次，为了完整理解所发生的过程，我们看一下这段代码的 MSIL：

```
.method public hidebysig static void Main() il managed
{
  .entrypoint
  // Code size       39 (0x27)
  .maxstack  3
  .locals (class NumericTest V_0,
           class NumericTest V_1,
           bool V_2)
  IL_0000:  ldc.i4.s   42
  IL_0002:  newobj     instance void NumericTest::.ctor(int32)
  IL_0007:  stloc.0
  IL_0008:  ldc.i4.s   42
  IL_000a:  newobj     instance void NumericTest::.ctor(int32)
  IL_000f:  stloc.1
  IL_0010:  ldstr      "{0}"
  IL_0015:  ldloc.0
  IL_0016:  ldloc.1
  IL_0017:  ceq
  IL_0019:  stloc.2
  IL_001a:  ldloca.s   V_2
  IL_001c:  box        ['mscorlib']System.Boolean
  IL_0021:  call   void ['mscorlib']System.Console::WriteLine
                              (class System.String,class System.Object)
  IL_0026:  ret
} // end of method 'RelationalOpslApp::Main'
```

请看.local 行。编译器声明 Main 方法将有三个局部变量。前两个局部变量是 NumericTest 对象，而第三个局部变量是一个布尔型。现在跳到 IL_002 行和 IL_007 行。正是在这里，MSIL 实例化了一个 test1 对象，并且利用操作码 stloc 把返回的引用保存到第一个局部变量中。然而，关键的一点在于 MSIL 保存的是新创建的对象的地址。然后，在 IL_000a 行和 IL_000f 行，您可以看到 MSIL 操作码创建了 test2 对象，并不返回的引用保存在第二个局部变量中。最后，IL_0015 行和 IL_0016 行简单地通过调用 ldloc 来把局部变量加载到堆栈中，然后 IL_0017 行调用了 ceq 操作码，这个操作码比较堆栈顶部的两个值(也就是说，对

test1 对象和 test2 对象的引用)。然后返回值被保存在第三个局部变量中，稍后就调用 System.Console.Writeline 方法来打印结果。

那么怎样才能对两个对象按照成员一个一个地进行比较呢？答案就在所有的.NET 框架对象的隐式基础类中。类 System.Object 有一个名为 Equals 的方法，这个方法专门是为了这个目的而设计的。例如，下列代码对对象的内容执行了比较，这时返回的结果就是您所预期的 true：

```
using System;

class RelationalOps2App
{
    public static void Main()
    {
        Decimal test1 = new Decimal(42);
        Decimal test2 = new Decimal(42);

        Console.WriteLine("{0}", test1.Equals(test2));
    }
}
```

注意，例子 RelationalOps1App 中使用的是一个自制的类(NumericTest)，而第二个例子使用的是一个.NET 类(Decimal)。这样做的原因是，System.Object.Equals 方法必须被覆盖以便于按照实际的成员进行比较。因此，在 NumericTest 类上使用 Equals 方法就无法工作，因为我们并没有覆盖这个方法。然而，因为 Decimal 类覆盖了其继承来的 Equals 方法，所以可以按照我们预计的那样工作。

另外一种处理对象之间比较的方法就是通过操作符重载。重载操作符就定义了在特定类型的对象之间发生的操作。例如，对于 string 对象，操作符＋把几个字符串连接起来，而不是执行一个加操作。我们将在第 13 章中详细讨论操作符重载。

10.3.4 简单赋值操作符

赋值操作符左边的值被称为 lvalue；赋值操作符右边的值被称为 rvalue。rvalue 可以是任何常数、变量、数字、或者是返回值与 value 相兼容的方法。但是，lvalue 必须是一个已定义类型的变量。原因是这个值要从右边复制到左边。因此，在内存中必须分配相应的物理空间，这个物理空间最后就是新值所存放的地方。作为一个例子，您可以使用语句 i=4，因为 i 代表内存中的一个物理地址，要么是在堆栈中，要么是在堆中，这取决于 i 变量的实际类型。但是，您不可以执行语句 4=i，因为 4 是一个值，而不是内存中的一个内容可以被改变的变量。从技术上来说，C# 中的规则就是 lvalue 可以是变量、属性或者索引器。要

想了解更多关于属性和索引器的信息，请参阅第 7 章。为了使内容保持简单化，此处仍使用在这一章里找到的使用变量的例子。

虽然数字的赋值很直接，但是涉及到对象的赋值就是另一回事了。记住，当您处理对象的时候，您并不是在处理简单的、在堆栈中分配的易于复制和移动的元素。当处理对象时，您实际上所得到的是一个堆分配的实体的引用。因而，当您试图把对象(或者任何引用类型)赋值给一个变量时，您并不是像对值类型那样在复制数据。您只是简单地把一个引用从一个地方复制到另一个地方。

让我们假设您有两个对象：test1 和 test2。如果写语句 test1＝test2，那么 test1 不是 test2 的一个复制，而是同一个对象！test1 对象指向与 test2 相同的内存。因而，任何对 test1 对象所进行的修改同时也就是对 test2 对象的修改。这里有一个程序来演示这一点：

```
using System;

class Foo
{
    public int i;
}

class RefTest1App
{
    public static void Main()
    {
        Foo test1 = new Foo();
        test1.i = 1;

        Foo test2 = new Foo();
        test2.i = 2;

        Console.WriteLine("BEFORE OBJECT ASSIGNMENT");
        Console.WriteLine("test1.i={0}", test1.i);
        Console.WriteLine("test2.i={0}", test2.i);
        Console.WriteLine("\n");

        test1 = test2;

        Console.WriteLine("AFTER OBJECT ASSIGNMENT");
        Console.WriteLine("test1.i={0}", test1.i);
        Console.WriteLine("test2.i={0}", test2.i);
        Console.WriteLine("\n");

        test1.i = 42;

        Console.WriteLine("AFTER CHANGE TO ONLY TEST1 MEMBER");
```

```
        Console.WriteLine("test1.i={0}", test1.i);
        Console.WriteLine("test2.i={0}", test2.i);
        Console.WriteLine("\n");
    }
}
```

运行这段代码，您将会看到下列输出：

```
BEFORE OBJECT ASSIGNMENT
test1.i=1
test2.i=2
AFTER OBJECT ASSIGNMENT
test1.i=2
test2.i=2
AFTER CHANGE TO ONLY TEST1 MEMBER
test1.i=42
test2.i=42
```

让我们从头到尾来看一遍这个例子，以便于理解每一步骤。Foo 是一个简单的类，这个类定义了一个成员名 i。这个类的两个实例——test1 和 test2——在方法 main 中被创建。而且每次创建实例时，新对象的成员 i 都被相应地设定为 1 和 2。在这里，我们打印了这两个值，而这时看上去确实如您预期的那样：test1.i 的值是 1，而 test2.i 的值是 2。此处请注意：下一行代码把 test2 对象赋给了 test1。Java 程序员可能会知道接下来将会发生什么。但是，大多数 C++程序员都会认为对象 test1 的成员 i 现在等于对象 test2 中的成员 i(之所以这么假设是因为所编译的应用程序必须执行一种隐式的成员灵活复制的操作符)。事实上，这只是打印这两个对象成员的表象。但是，这两个对象之间的新关系现在更为复杂。把 42 赋给 test1.i 对象并再次打印这两个对象的成员 i 的值，会怎么样？对 test1 对象所作的改变也改变了 test2 对象！这是因为 test1 已经不是前面我们所知道的那个对象了。在把 test2 赋值给 test1 时，test1 对象已经丢失了，因为程序中没有任何引用指向它，稍后就会被垃圾收集器(garbage collector ，GC)回收 。 test1 和 test2 对象现在指向堆中的同一块内存。因而，对其中任何一个变量做出改变，其他变量也会相应改变。

请注意，输出的最后两行显示，即使代码设置了 test1.i 的值，test2.i 的值也被影响了。同样的道理，这也是因为这两个变量现在指向的是内存的同一地址，如果是 Java 程序员应该能够预料到这一点。然而，这对于 C++程序员来说就不容易理解了，因为在 C++中，复制一个对象仅仅意味着每个变量都有这个成员的一个复制，这样的话，一个对象所做出的修改不会影响其他对象。因为这是理解如何处理 C#中的对象的关键，所以让我们来看一看，如果把一个对象传递给一个方法会发生什么：

```
using System;
class Foo
```

```
{
    public int i;
}

class RefTest2App
{
    public void ChangeValue(Foo f)
    {
        f.i = 42;
    }

    public static void Main()
    {
        RefTest2App app = new RefTest2App();

        Foo test = new Foo();
        test.i = 6;

        Console.WriteLine("BEFORE METHOD CALL");
        Console.WriteLine("test.i={0}", test.i);
        Console.WriteLine("\n");

        app.ChangeValue(test);

        Console.WriteLine("AFTER METHOD CALL");
        Console.WriteLine("test.i={0}", test.i);
        Console.WriteLine("\n");
    }
}
```

在大多数语言中——不包括 Java——这里的代码会导致在方法 RefTest2App.ChangeValue 的局部堆栈中创建 test 对象的一个复制。如果是这样的话，在 Main 方法中所创建的 test 对象不会看到方法 ChangeValue 中对 f 对象所做的任何改变。但是，这里所发生的是，Main 方法传递了一个指向堆分配的 test 对象的引用。当方法 ChangeValue 处理其局部变量 f.i 时，它实际上是在直接处理 Main 方法中的 test 对象。

10.4 本章小结

对任何编程语言都很关键的一个部分就是其处理赋值、数学、关系和逻辑操作的方式，这些操作是处理现实中的应用所必需的。在代码中通过操作符来控制这些操作。影响代码中操作符作用的因素包括优先级以及左和右联合性。除了提供一个强大的预定义操作符集以外，C# 还通过用户定义的实现来扩展这些操作符，我们将在第 13 章中加以讨论。

第 11 章　程序流程控制

在 C#程序中，使您能够控制程序流程的语句主要可以归为以下三类：选择语句、迭代语句和跳转语句。在每种语句中，都要执行一个检测，这个检测产生一个布尔型的值，而这个布尔值被用于控制应用程序的执行流程。在这一章里，您将要学到如何使用各种语句类型来控制结构流程。

11.1　选 择 语 句

选择语句可以用来判断执行什么代码以及何时执行这些代码。C#提供了两条选择语句：switch 语句根据值来选择运行的代码，而 if 语句根据一个布尔条件来选择运行的代码。这两条选择语句中，最为常用的是 if 语句。

11.1.1　if 语句

如果表达式的计算值是 true 的话，那么 if 语句就执行一条或多条语句。if 语句的语法如下所示——方括号表示 else 语句的使用是可选的(我们很快就会讨论这一点)：

```
if (expression)
    statement1
[else
    statement2]
```

这里，expression 是任何可以产生布尔结果的测试。如果 expression 的结果为 true，控制就被传递到 statement1。如果结果为 false，而且还有 else 子句，那么控制就会被传递到 statement2。此外，要注意 statement1 和 statement2 既可以由单条语句组成，用分号表示终止(这被称为简单语句)，也可以由多条语句组成，这些语句被花括号括起来(这被称为复合语句)。下面的例子说明了如果 expression1 的值为 true，那么就使用复合语句：

```
if (expression1)
{
    statement1
    statement2
```

```
}
```

在下面的例子中，应用程序要求用户类型是一个 1 到 10 之间的数字。然后产生一个随机数，用户被告知他们所选择的数字是否与随机数字相匹配。这个简单的例子说明了如何在 C# 中使用 if 语句。

```
using System;

class IfTest1App
{
    const int MAX = 10;

    public static void Main()
    {
        Console.Write("Guess a number between 1 and {0}...", MAX);
        string inputString = Console.ReadLine();

        int userGuess = inputString.ToInt32();

        Random rnd = new Random();
        double correctNumber = rnd.NextDouble() * MAX;
        correctNumber = Math.Round(correctNumber);

        Console.Write("The correct number was {0} and you guessed {1}...",
                      correctNumber, userGuess);
        if (userGuess == correctNumber) // They got it right!
        {
            Console.WriteLine("Congratulations!");
        }
        else // Wrong answer!
        {
            Console.WriteLine("Maybe next time!");
        }
    }
}
```

11.1.2 多个 else 子句

if 语句的 else 子句使得在 if 的判断语句结果为 false 的时候，可以选择一个替代的动作过程。在猜数的例子中，应用程序在用户所猜测的数字和随机产生的数字之间执行一个简单的比较。在这种情况下只存在两种可能性：用户的猜测要么是错误的，要么是正确的。If 语句和 else 语句也可以被组合到一起，来处理这样的情况，即测试多于两种条件。在下

面的这个例子中将会问用户，他或她现在在使用什么语言(包括 C#)。其中可供选择的语言一共有三种，因此 if 语句必须能够处理四种可能的回答：选择三种语言中的一种，以及用户可能选择了一门未知的语言。下面就是用 if/else 语句来编程处理这种情况的方法：

```
using System;

class IfTest2App
{
    const string CPlusPlus = "C++";
    const string VisualBasic = "Visual Basic";
    const string Java = "Java";

    public static void Main()
    {
        Console.Write("What is your current language of choice " +
                      "(excluding C#)?");
        string inputString = Console.ReadLine();

        if (0 == String.Compare(inputString, CPlusPlus, true))
        {
            Console.WriteLine("\nYou'll have no problem picking " +
                              "up C# !");
        }
        else if (0 == String.Compare(inputString, VisualBasic, true))
        {
            Console.WriteLine("\nYou'll find lots of cool VB features" +
                              "in C# !");
        }
        else if (0 == String.Compare(inputString, Java, true))
        {
            Console.WriteLine("\nYou'll have an easier time " +
                              "picking up C# <G> !!");
        }
        else
        {
            Console.WriteLine("\nSorry - doesn't compute.");
        }
    }
}
```

注意==操作符的使用，这是为了把 string.Compare 方法的返回值与 0 相比较。这是因为如果第一个字符串比第二个字符串小，String.Compare 方法就返回-1，如果第一个字符串比第二个字符串大，string.compare 方法就返回 1，而如果两个字符串是完全相同的话，

string.compare 方法就返回 0。但是，这也显示了一些很有趣的细节，关于 C# 是如何强制使用 if 语句的，我们将在下一节加以介绍。

11.1.3　C# 如何强制使用 if 规则

C# 中的 if 语句检测表达式必须产生一个布尔型的值，这一点遵循了新的 C# 编程边界检查的要求。这与 C++之类的语言形成了鲜明的对比，在 C++中，除了 0 以外，任何含有其他值的变量都可以被程序员来使用 if 语句加以检测。下面的例子显示了 C++开发人员第一次使用 C# 中的 if 语句时常见的几个错误：

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class InvalidIfApp
{
    protected static TestClass GetTestClass()
    {
        return new TestClass();
    }

    public static void Main()
    {
        int foo = 1;
        if (foo) // ERROR: attempting to convert int to bool.
        {
        }

        TestClass t = GetTestClass();
        if (t) // ERROR: attempting to convert TestClass to bool.
        {
            Console.WriteLine("{0}", t);

            ITest i = t as ITest;
            if (i) // ERROR: attempting to convert ITest to bool.
            {
```

```
                // ITest methods
            }
        }
    }
}
```

如果对这些代码进行编译，C#编译器会给出下列错误消息：

```
    invalidIf.cs(22,7): error CS0029: Cannot implicitly convert type 'int'
to 'bool'
    invalidIf.cs(27,7): error CS0029: Cannot implicitly convert type
'TestClass' to 'bool'
    invalidIf.cs(31,14): warning CS0183:
    The given expression is always of the provided ('ITest') type
    invalidIf.cs(32,8): error CS0029: Cannot implicitly convert type
'ITest' to 'bool'
```

正如您可以看见的，程序中三次试图在if语句判断中使用不产生布尔值的操作，而编译器则给出了三次响应。原因在于，C#设计者希望帮助您避免写出那些含糊不清的代码，并且相信编译器可以加强使用if语句的原先目的——也即，根据布尔检测的结果来控制程序流程的执行。下面我们对刚才这个例子的代码进行了重写，编译器没有产生错误。刚才那个程序中每一行导致编译器产生错误的代码都被修改了，这些语句都各自含有了一个表达式，这个表达式产生了一个布尔结果，这样就满足了编译器的要求。

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class ValidIfApp
{
    protected static TestClass GetTestClass()
    {
        return new TestClass();
    }

    public static void Main()
    {
        int foo = 1;
```

```
            if (foo > 0)

            {
            }

            TestClass t = GetTestClass();
            if (t != null)
            {
                Console.WriteLine("{0}", t);

                ITest i = t as ITest;
                if (i != null)
                {
                    // ITest methods.
                }
            }
        }
    }
```

11.1.4 switch 语句

利用 switch 语句，您可以指定一个表达式，这个表达式返回一个整数值，此外还有一部分或是多部分的代码，程序根据表达式的结果来运行这些代码。使用多个 if/else 语句与这条语句的作用相似。尽管您能在多个 if/else 语句中指定多个(可能并不相关的)条件，但是毕竟一条 switch 语句只包含一个条件语句，紧接着后面的就是条件语句的结果，而程序中代码就准备处理这些结果。下面是 switch 语句的语法：

switch *(switch_expression)*
{
case *constant-expression:*
statement
jump-statement
...
case *constant-expressionN:*
statementN
[default]
}

这里要记住两条主要的规则。第一，switch_expression 必须是 sbyte、byte、short、int、uint、long、ulong、char 或者是 string(或是基于这些类型的 enum)类型中的一种(或是可以被

隐式转换成这些类型)。第二，您必须为每条 case 语句提供一条跳转语句(jump-statement)，除非这条 case 语句是 switch 语句中的最后一个 case 语句，包括 break 语句。因为这一点与其他几种语言相比有一些不同，所以将会在本章后面进行更加详细的解释。

从概念上来说，switch 语句的工作原理与 if 语句相似。首先，对 switch_expression 进行求值，然后求值的结果会与每个 constant_expression 或 “情况标签”(case labels)进行比较，这些 constant_expression 和情况标签是在不同的 case 语句中定义的。一旦在某个 case 语句中可以匹配，程序的控制流程就会传递到这个 case 语句的第一行代码。

除了可以指定不同的 case 语句以外，switch 语句也允许定义一条 default 语句。这有点像 if 语句的 else 子句。注意，每条 switch 语句只能够有一条 default 语句，此外，如果程序没有为 switch_expression 找到合适的情况标签，那么控制流程就会被传递到 switch 语句结尾括号后代码的第一行。这里有一个例子——一个 Payment 类使用 switch 语句来判断选择哪一个 “tender”：

```
using System;

enum Tenders : int
{
    Cash = 1,
    Visa,
    MasterCard,
    AmericanExpress
};

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }
        set
        {
            this.tender = value;
        }
```

```
    }

    public void ProcessPayment()
    {
        switch ((int)(this.tender))
        {
            case (int)Tenders.Cash:
                Console.WriteLine("\nCash - Accepted");
                break;

            case (int)Tenders.Visa:
                Console.WriteLine("\nVisa - Accepted");
                break;

            case (int)Tenders.MasterCard:
                Console.WriteLine("\nMastercard - Accepted");
                break;

            case (int)Tenders.AmericanExpress:
                Console.WriteLine("\nAmerican Express - Accepted");
                break;

            default:
                Console.WriteLine("\nSorry - Invalid tender");
                break;
        }
    }
}

class SwitchApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Visa);
        payment.ProcessPayment();
    }
}
```

因为我们在实例化一个 payment 类时传递给它一个参数——Tender.Visa，所以运行这个程序会得到如下输出：

```
Visa - Accepted.
```

11.1.5　组合情况标签

在 Payment 例子中，我们为每种可能的 Payment.tender 字段求值使用了不同的情况标

签。然而，如果您想把几个情况标签组合到一起该怎么办？例如，对于 Tender enum 中承认的三种信用卡的每一种，您可能都只想要显示一个信用卡授权的对话框。在这种情况下，您可以把情况标签逐个放置，如下所示：

```
using System;

enum Tenders : int
{
    Cash = 1,
    Visa,
    MasterCard,
    AmericanExpress
};

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }
        set
        {
            this.tender = value;
        }
    }

    public void ProcessPayment()
    {
        switch ((int)(this.tender))
        {
            case (int)Tenders.Cash:
                Console.WriteLine
                        ("\nCash - Everyone's favorite tender.");
                break;

            case (int)Tenders.Visa:
```

```
            case (int)Tenders.MasterCard:
            case (int)Tenders.AmericanExpress:
                Console.WriteLine
("\nDisplay Credit Card Authorization Dialog.");
                break;

            default:
                Console.WriteLine("\nSorry - Invalid tender.");
                break;
        }
    }
}

class CombiningCaseLabelsApp
{

    public static void Main()
    {
        Payment payment = new Payment(Tenders.MasterCard);
        payment.ProcessPayment();
    }
}
```

如果用 Tender.Visa、Tenders.MasterCard 或者 Tenders.AmericanExpress 作为参数来实例化一个 Payment 类，那么会得到下列输出：

```
Display Credit Card Authorization Dialog.
```

11.1.6　在 switch 语句中没有“贯行”

在 C# 的整个设计阶段，当判断语言中是否要加入一个特性时，其设计者意识到应用“风险/回报”测试的重要性。而“贯行”(fall-through)特性就是一个没有通过该测试的例子。通常在 C++中，当一个 case 语句的常量表达式与 switch_expression 相匹配时，这个 case 语句就可以执行。然后这条 switch 语句就会利用 break 语句来退出。贯行则导致在没有 break 语句的情况下，switch 语句中的下一条 case 语句被执行。

虽然在 C# 中不支持贯行，但是在有的情况下一般是要使用贯行的，例如程序中有两个情况标签，而第二个标签代表的是一个操作，这个操作在每条 case 语句中都要执行。例如，作者曾经用 C++编写过一个数据库编辑器，这个编辑器允许最终用户以图形方式创建他们自己的表和字段。这些表中的每一张都在类似于 Windows Explore 的用户界面中以树形方式的视图显示出来。如果最终用户用鼠标右键点击树形视图，就会显示一个带有一些

选项的菜单，其中包括打印所有的表，以及创建新的表。如果最终用户用鼠标右键在某一个特定的表上点击，那么将显示这张表的上下文环境菜单。但是，所有的树形视图选项中也都需要有这个菜单。从概念上来说，该程序应该与以下代码相似：

```
// Dynamically create menu in C++.
switch(itemSelected)
{
   case TABLE:
   // Add menu options based on current table;
   // break left out intentionally.

   case TREE_VIEW:
   // Add menu options for tree view.
   break;
}
// Display menu.
```

第一个情况标签相当于两个标签的组合，无需编写重复的代码或是在程序中加入对同一方法的两个调用。然而，尽管这种特性的使用方便了程序员，但是 C# 的设计者们仍然认为，使用该特性所带来的风险比好处要多，因为在大多数情况下，break 语句都被无意中遗漏了，这就导致了程序中的错误，而且这些错误更加难以追踪。为了在 C# 中实现前面所说的这一点，您可能最好还是使用 if 语句来代替：

```
// Dynamically create menu.
if (itemSelected == TABLE)
{
   // Add menu options based on current table.
}
// Add menu options for tree view.
// Display menu.
```

11.2 迭代语句

在 C# 中，语句 while、do/while、for 和 foreach 使您能够执行可控制的迭代或者循环。除了 foreach 以外，在每种情况下，都执行一条特定的简单语句或者是复合语句，直到布尔表达式的计算结果为 false 为止。而 foreach 语句通常用于迭代一系列对象。

11.2.1 while 语句

while 语句采用以下的形式：

while *(Boolean-expression)*
embedded-statement

我们再次使用本章前面所使用过的猜数例子，这次我们重新编写了这个例子的代码，其中使用了一条 while 语句，这样您就可以一直玩这个游戏，直到您猜到了这个数，或是决定退出游戏。

```
using System;

class WhileApp
{
    const int MIN = 1;
    const int MAX = 10;
    const string QUIT_CHAR = "Q";

    public static void Main()
    {
        Random rnd = new Random();
        double correctNumber;

        string inputString;
        int userGuess;

        bool correctGuess = false;
        bool userQuit = false;

        while (!correctGuess && !userQuit)
        {
            correctNumber = rnd.NextDouble() * MAX;
            correctNumber = Math.Round(correctNumber);

            Console.Write
                ("Guess a number between {0} and {1}...({2} to quit)",
                MIN, MAX, QUIT_CHAR);
            inputString = Console.ReadLine();

            if (0 == string.Compare(inputString, QUIT_CHAR, true))
                userQuit = true;
            else
            {
                userGuess = inputString.ToInt32();
                correctGuess = (userGuess == correctNumber);

                Console.WriteLine
```

```
                    ("The correct number was {0}\n",
                     correctNumber);
                }
            }

            if (correctGuess && !userQuit)
            {
                Console.WriteLine("Congratulations!");
            }
            else

            {
                Console.WriteLine("Maybe next time!");
            }
        }
    }
```

编译并运行这个应用程序，将会得到下列相似的输出：

```
C:\>WhileApp
Guess a number between 1 and 10...(Q to quit)3
The correct number was 5
Guess a number between 1 and 10...(Q to quit)5
The correct number was 5
Congratulations!
C:\>WhileApp
Guess a number between 1 and 10...(Q to quit)q
Maybe next time!
```

11.2.2 do/while 语句

回头看一看 while 语句的语法，您就可以发现其中存在着发生某种问题的可能性。布尔表达式的求值是在嵌入语句之前执行的。因此，前一节中的应用程序把 correctGuess 和 userQuit 变量初始化为 false，这样就保证了可以进入 while 循环。从这里开始，这些变量的值就由用户是否猜中了这个数字或是否想要退出游戏来控制了。然而，如果我们想要确保嵌入语句总是至少执行一次，而不需要先手工地设置变量值时该怎么办？这时就应该使用 do/while 语句了。

do/while 语句采用以下形式：

do

embedded-statement

while (*Boolean-expression*)

因为对 while 语句中的布尔表达式求值发生在嵌入语句语句体之后的，这样就可以保证嵌入语句至少会被执行一次。我们重写的猜数程序就使用了 do/while 语句，如下所示：

```
using System;

class DoWhileApp
{
    const int MIN = 1;
    const int MAX = 10;
    const string QUIT_CHAR = "Q";

    public static void Main()
    {
        Random rnd = new Random();
        double correctNumber;

        string inputString;
        int userGuess = -1;

        bool userHasNotQuit = true;

        do
        {
            correctNumber = rnd.NextDouble() * MAX;
            correctNumber = Math.Round(correctNumber);

            Console.Write
                ("Guess a number between {0} and {1}...({2} to quit)",
                MIN, MAX, QUIT_CHAR);
            inputString = Console.ReadLine();

            if (0 == string.Compare(inputString, QUIT_CHAR, true))
                userHasNotQuit = false;
            else
            {
                userGuess = inputString.ToInt32();
                Console.WriteLine
                    ("The correct number was {0}\n", correctNumber);
            }
        } while (userGuess != correctNumber
            && userHasNotQuit);

        if (userHasNotQuit
            && userGuess == correctNumber)
```

```
            {
                Console.WriteLine("Congratulations!");
            }
            else // wrong answer!
            {
                Console.WriteLine("Maybe next time!");
            }
        }
    }
```

这个经过改写的应用程序功能与前面用 while 语句的那个程序的功能是相同的。这两个程序之间惟一的区别就在于循环是如何控制。实际上，您可以发现，while 语句比 do/while 语句的使用更为普遍。然而，因为您可以很容易地通过初始化一个布尔类型的变量来控制循环的入口，所以对于这些语句的选择只是一个个人偏好的问题。

11.2.3 for 语句

到目前为止，您所见到的最为常见的语句就是 for 语句。for 语句由三个部分组成。第一个部分用于在循环开始处执行初始化——这个部分只会被执行一次。第二个部分是条件检验，判断循环是否应该再次运行。而最后一部分被称为“步幅”(stepping)，通常(但不是必须)被用于对计数器进行增值，而计数器则控制循环的继续——通常在第二个部分对这个计数器加以检验。For 通常采用以下格式：

for *(initialization; Boolean-expression; step)*
    embedded-statement

注意，这三个部分(*initialization*, *Boolean-expression*, *step*)中的任何一个部分都可以为空。当布尔表达式的求值结果为 false，控制从循环的顶部传递到紧接着嵌入语句的第一行代码。因此，除了给出另外两个部分(initialization 和 step)以外，for 语句的运行方式与 while 语句是一样的。这里有一个 for 语句的例子，这个例子显示了可打印的 ASCII 字符：

```
using System;

class ForTestApp
{
    const int StartChar = 33;
    const int EndChar = 125;

    static public void Main()
    {
        for (int i = StartChar; i <= EndChar; i++)
        {
```

```
            Console.WriteLine("{0}={1}", i, (char)i);
        }
    }
}
```

for 循环中所发生的事件顺序如下所示：

1. 堆栈中分配了一个数值类型的变量(i)，并被初始化为 33。注意，一旦 for 循环结束，这个变量就不在作用域范围内了。
2. 当变量 i 的值小于 126 时，执行 for 语句体中嵌入的语句。在这个例子中，使用了复合语句。然而，由于这条 for 语句中只包含一条语句，所以即使没有花括号，程序仍然可以得到相同的结果。
3. 每循环一次，变量 i 的值就增加 1。

11.2.4　嵌套循环

在一个 for 循环的嵌入语句中，还可以有其他 for 循环，这样的 for 循环一般被称为嵌套循环(nested loops)。还是使用前一节的例子，在其中加入了一个嵌套循环，这个循环使得应用程序在每一行打印三个字符，而不是每行打印一个字符：

```
using System;

class NestedForApp
{
    const int StartChar = 33;
    const int EndChar = 125;
    const int CharactersPerLine = 3;

    static public void Main()
    {
        for (int i = StartChar; i <= EndChar; i+=CharactersPerLine)
        {
            for (int j = 0; j < CharactersPerLine;  j++)
            {
                Console.Write("{0}={1} ", i+j, (char)(i+j));
            }
            Console.WriteLine("");
        }
    }

}
```

请注意，在外层循环中定义的变量 i 即使在内层循环中仍然处于作用域内。然而，变

量 j 对于外层循环来说是不可用的。

11.2.5 使用逗号操作符

逗号不仅可以在方法参数中起到分隔符的作用，而且还可以作为 for 语句的一个操作符。在 for 语句的 *initialization* 和 *step* 部分，逗号操作符可以被用来对按顺序处理的多个语句进行分界。在下面这个例子中，还要使用前一节的嵌套循环例子，并且会使用逗号操作符把这个嵌套循环转换成为一个 for 语句的单循环：

```
using System;

class CommaOpApp
{
    const int StartChar = 33;
    const int EndChar = 125;

    const int CharactersPerLine = 3;

    static public void Main()
    {
        for (int i = StartChar, j = 1; i <= EndChar; i++, j++)
        {
            Console.Write("{0}={1} ", i, (char)i);
            if (0 == (j % CharactersPerLine))
            {
                // New line if j is divisible by 3.
                Console.WriteLine("");
            }
        }
    }
}
```

在 for 语句中，逗号操作符的功能是很强大的，但是同时这也会导致产生一些不协调或是难以维护的代码。下面使用逗号操作符的例子中也使用了常量，尽管从技术上来说是正确的，但是使用的方式并不是很恰当。

```
using System;

class CommaOp2App
{
    const int StartChar = 33;
    const int EndChar = 125;
```

```
        const int CharsPerLine = 3;
        const int NewLine = 13;
        const int Space = 32;

        static public void Main()
        {
            for (int i = StartChar, extra = Space;
                i <= EndChar;
                i++, extra = ((0 == (i - (StartChar-1)) % CharsPerLine)
                ? NewLine : Space))
            {
                Console.Write("{0}={1} {2}", i, (char)i, (char)extra);
            }
        }
    }
```

11.2.6 foreach 语句

多年以来，像Visual Basic这样的语言已经有了一些特殊的语句，这些语句专门被设计用来对数组和集合进行迭代。C#也有这样的结构，那就是foreach语句，这条语句采用如下格式：

foreach (*type* in *expression*)
    embedded-statement

请留意下面的数组类：

```
class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
    }
}
```

从我们已经学到的各种不同迭代语句中，我们可以了解到，这个数组的遍历可以使用几种不同的语句来完成。然而，对于绝大多数 Java 和 C++程序员来说，编写这样一个应用程序的最符合逻辑的方法就是如下所示：

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
    }
}

class Foreach1App
{
    public static void Main()
    {
        MyArray myArray = new MyArray();

        for (int i = 0; i < myArray.words.Count; i++)
        {
            Console.WriteLine("{0}", myArray.words[i]);
        }
    }
}
```

但是这种方法中存在着一些问题：

- 如果 for 语句的初始化变量(i)没有被正确地初始化，那么就无法完整地迭代整个一列数组。
- 如果 for 语句的布尔表达式不正确，那么就无法完整地迭代整个一列数组。
- 如果 for 语句的步幅(step)不正确，那么就无法完整地迭代整个一列数组。
- 在请求集合和数组的 count 值时，它们具有不同的方法和属性。
- 在取出一个特定的元素时，集合和数组具有不同的语义。
- for 循环中的嵌入语句需要把从数组中取出的元素放入正确类型的变量中。

代码中可能有各种不同的错误。使用 foreach 语句，您就可以避免这些问题，此外还可以以一种统一的方式来迭代任何集合和数组。利用 foreach 语句，可以对前面的代码进行重新编写，如下面所示：

```
using System;
```

```
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
    }
}

class Foreach2App
{
    public static void Main()
    {
        MyArray myArray = new MyArray();

        foreach (string word in myArray.words)
        {
            Console.WriteLine("{0}", word);
        }
    }
}
```

请注意，使用 foreach 语句是多么“直接”。这样就可以保证取到每个元素，因为您不必手工地设置循环和请求 count 值，而且这条语句可以自动地把元素放在您所命名的变量中。您只需要在嵌入的语句中指出变量即可。

11.3 跳转语句

在前面部分所涉及的任何迭代语句中的嵌入语句内部，您都可以利用几条语句来控制程序的执行流程，这些语句被称为跳转语句：break、continue、goto 和 return。

11.3.1 break 语句

使用 break 语句可以中断当前的循环或者循环中出现的条件语句。然后控制流程就被

传递到紧接着循环或者条件语句的嵌入语句后面的第一行代码处。break 语句具有最简单的语法，没有圆括号或参数值，并且在您需要跳出循环或条件语句的地方采用以下格式：

break

在下面的例子中，程序将会打印 1 到 100 之间的每个可以被 6 整除的数。然而，当计数到 66 时，break 语句将会导致 for 循环中断。

```
using System;

class BreakTestApp
{
   public static void Main()

   {
      for (int i = 1; i <= 100; i ++)
      {
         if (0 == i % 6)
         {
            Console.WriteLine(i);
         }

         if (i == 66)
         {
            break;
         }
      }
   }
}

}
```

11.3.2 中断无穷循环

break 语句的另一个用处是创建一个无穷循环，在这个无穷循环中，只有当到达 break 语句时控制流程才会被跳转到这个无穷循环以外。下面的例子演示了一个编写猜数游戏的方法，我们在本章前面的部分中已经使用过这个游戏的例子。在这个例子中，我们用一条 break 语句来退出循环。一旦用户输出字母 Q，程序就会退出循环。注意，此处把 while 语句改写成为 while(true)，这样的话循环就会一直不结束，直到遇到一条 break 语句。

```
using System;

class InfiniteLoopApp
```

```
{
    const int MIN = 1;
    const int MAX = 10;
    const string QUIT_CHAR = "Q";

    public static void Main()
    {
        Random rnd = new Random();
        double correctNumber;

        string inputString;
        int userGuess;

        bool correctGuess = false;
        bool userQuit = false;

        while(true)
        {
            correctNumber = rnd.NextDouble() * MAX;
            correctNumber = Math.Round(correctNumber);

            Console.Write
                ("Guess a number between {0} and {1}...({2} to quit)",
                MIN, MAX, QUIT_CHAR);
            inputString = Console.ReadLine();

            if (0 == string.Compare(inputString, QUIT_CHAR, true))
            {
                userQuit = true;
                break;
            }
            else
            {
                userGuess = inputString.ToInt32();
                correctGuess = (userGuess == correctNumber);

                if ((correctGuess = (userGuess == correctNumber)))
                {
                    break;
                }
                else
                {

                    Console.WriteLine
```

```
                    ("The correct number was {0}\n", correctNumber);
                }
            }
        }

        if (correctGuess && !userQuit)
        {
            Console.WriteLine("Congratulations!");
        }
        else
        {
            Console.WriteLine("Maybe next time!");
        }
    }
}
```

最后一点：也可以在这里使用一条空的 for 语句来代替 while(true)，其格式为 for(;;)。这两条语句的效果是相同的，因此这是一个个人偏好的问题。

11.3.3 continue 语句

同 break 语句一样，continue 语句也可以改变循环的执行。然而，continue 语句不是终止当前循环中嵌入的语句，而是停止当前的循环并把控制返回到当前循环的顶部，以便于进行下一次循环。在下面的例子中，将对一个字符串数组进行检查，看其中是否有重复的字符串。要做到这一点，其中的一种方法是用一个嵌套循环迭代整个数组，把每个元素都与其他元素相比较。然而，我们当然不想把元素与其自身相比较，否则就会得到无效的重复数目。因此，如果数组中的一个索引(i)与数组中的另一个索引(j)相等的话，这就意味着此时两个相比较的元素实际上是同一个元素，这样就可以不比较这两个元素。遇到这种情况时，可使用 continue 语句来中止当前循环，并且把控制传递到循环的顶部。

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
```

```
        words.Add("baz");
        words.Add("bar");
        words.Add("ba");
        words.Add("foo");
    }
}

class ContinueApp
{
    public static void Main()
    {
        MyArray myArray = new MyArray();
        ArrayList dupes = new ArrayList();

        Console.WriteLine("Processing array...");
        for (int i = 0; i < myArray.words.Count; i++)
        {
            for (int j = 0; j < myArray.words.Count; j++)
            {
                if (i == j) continue;
                if (myArray.words[i] == myArray.words[j]
                    && !dupes.Contains(j))
                {
                    dupes.Add(i);
                    Console.WriteLine("'{0}' appears on lines {1} and {2}",
                        myArray.words[i],
                        i + 1,
                        j + 1);
                }
            }
        }
        Console.WriteLine("There were {0} duplicates found",
            ((dupes.Count > 0) ? dupes.Count.ToString() : "no"));
    }
}
```

注意，也可以用 foreach 循环来迭代这个数组。然而，在这个例子中，由于希望追踪到当前所比较的元素，所以才使用了 for 循环。

11.3.4　“声名狼藉”的 goto 语句

在编程的历史上，可能从来没有一种结构像 goto 语句那样遭人非议了。因此，在我们

详细介绍 goto 语句的语法和一些用法之前，让我们先来看一看为什么有人会如此强烈地反对使用这条语句，以及使用这条语句可以解决的问题类型。

11.3.4.1 goto 语句简史

从 Edsger W. Dijkstra 在 1968 年发表的论文“Go To Statement Considered Harmful”的标题中，就可以看出他对 goto 语句所持的不赞成态度。在编程历史的那个年代里，曾经有一场争论围绕着结构化编程而展开。不幸的是，注意力并不是主要集中在结构化编程的总体方面，而是放在了一些相对较小的细节上：特定的语句，例如 goto 语句(现在通常被称为 goto 关键字)，是否应该在现代编程语言中提供。常常会出现这样的情况，许多人都采纳了 Dijkstra 的建议，并且走向了极端，认为 goto 语句的所有用法都是不好的，而且应该不惜一切代价来避免使用 goto 语句。

使用 goto 语句所产生的问题原因并不在于 goto 语句本身——而更应该说是 goto 语句在不正确场合的使用导致了这些问题。在结构化编程流程中，goto 语句也可以是一种有用的工具，并且与其他分支语句和迭代机制相比，可以用来编写出更易于理解的代码。这方面的一个例子就是 Dijkstra 所提出的“loop-and-a-half”问题。这里有一个 loop-and-a-half 问题的传统流程，下面是这个问题的伪代码：

```
loop
   read in a value
   if value == sentinel then exit
   process the value
end loop
```

从循环中退出是利用在循环中部的 exit 语句来完成的。然而，在那些认为 goto 语句的所有用法都是错误的人看来，loop/exit/end loop 周期是相当令人头痛的。他们会编写这样的代码：

```
read in a value
while value != sentinel
   process the value
   read in a value
end while
```

不幸的是，正如斯坦福大学的 Eric S. Roberts 所指出的那样，第二种方法有两个主要的缺陷。第一，这种方法需要重复一些语句，这些语句用来读取一个值。任何时候，如果您要重复写代码，这都会导致一个很明显的维护问题，因为对任何一条语句做出修改的同时，都必须对其他语句也做出同样的修改。第二个问题更为微妙，可能也更令人头疼。编写易于理解且易于维护的可靠代码的关键，在于编写的代码能够以一种自然的方式阅读。在关于代码用途的任何非代码介绍中，一个人可能将解决方案描述为：首先，我需要读入一个

值，如果这个值是某个标记(sentinel)的话，那么就停止。如果不是，就处理这个值，并继续读入下一个值。因此，那些省略了 exit 语句的代码实际上更不直观，因为这种方法违反了考虑问题的自然方式。现在，让我们来看一下某些情况，其中使用 goto 语句是产生结构化控制流程的最佳方法。

11.3.4.2 使用 goto 语句

goto 语句可以采用以下格式中的任一种：

goto *identifier*

goto case *constant-expression*

goto default

在 goto 语句的第一种用法中，identifer 的目标是一条标签语句。标签语句采取以下格式：

identifer:

如果当前方法中不存在标签，那么就会发生编译时的错误。另外一条要记住的重要规则就是，goto 语句可以被用来跳出一个嵌套循环。然而，如果 goto 语句不处于标签的作用域范围以内，那么就会发生一个编译时的错误。因此，您不可能跳转进入一个嵌套循环。

在下面的例子中，应用程序迭代一个简单数组，读取每个值，直到读到一个标记值(sentinel value)，此时就退出循环。注意，goto 语句的行为很像 break 语句，因为这条语句导致了程序流程跳转出 foreach 循环。

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;
    public const string TerminatingWord = "stop";

    public MyArray()
    {
        words = new ArrayList();

        for (int i = 1; i <= 5; i++) words.Add(i.ToString());
        words.Add(TerminatingWord);
        for (int i = 6; i <= 10; i++) words.Add(i.ToString());
    }
}

class Goto1App
```

```
{
    public static void Main()
    {
        MyArray myArray = new MyArray();

        Console.WriteLine("Processing array...");

        foreach (string word in myArray.words)
        {
            if (word == MyArray.TerminatingWord) goto finished;
            Console.WriteLine(word);
        }

        finished:
            Console.WriteLine("Finished processing array");
    }
}
```

关于 goto 语句的使用，人们可能认为，即使不用标签，break 语句也可以达到同样的效果。我们将看一些其他格式的 goto 语句，而且您将发现，有些问题只能用 goto 语句才可以解决。

在介绍 switch 语句的那一部分中，讨论过这样的事实，即 C#中不支持贯行(fall-through)。即使支持贯行，也无法解决下面的问题。假设我们有一个 Payment 类(从本章的前面部分重用)，这个类接受几种格式的支付或投标方式：Visa、American Expression、MasterCard、现金以及记账(基本上是信用卡)。因为 Visa、American Expression、MasterCard 都是信用卡，我们可以把它们组合到一条标签，并且以相同的方式来处理它们。如果遇到记账的情况，我们将需要调用特定的方法，而如果遇到现金购买的情况时，我们只需要打印一张收据。同样，我们可能希望在所有的情况下都打印收据。我们怎样才能有三种不同情况的标签，而把前两种——信用卡和记账情况——都转移到 cash 标签上呢？而且在什么时候这样做呢？请看下面的代码，这个问题可能是一个很好的例子，来演示何时使用 goto 语句：

```
using System;

enum Tenders : int
{
    ChargeOff,
    Cash,
    Visa,
    MasterCard,
    AmericanExpress
```

```
};

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }

        set
        {
            this.tender = value;
        }
    }

    protected void ChargeOff()
    {
        Console.WriteLine("Charge off.");
    }

    protected bool ValidateCreditCard()
    {
        Console.WriteLine("Card approved.");
        return true;
    }

    protected void ChargeCreditCard()
    {
        Console.WriteLine("Credit Card charged");
    }

    protected void PrintReceipt()
    {
        Console.WriteLine("Thank you and come again.");
    }
```

```
        public void ProcessPayment()
        {
            switch ((int)(this.tender))
            {
                case (int)Tenders.ChargeOff:
                    ChargeOff();
                    goto case Tenders.Cash;

                case (int)Tenders.Visa:
                case (int)Tenders.MasterCard:
                case (int)Tenders.AmericanExpress:
                    if (ValidateCreditCard())
                        ChargeCreditCard();
                    goto case Tenders.Cash;

                case (int)Tenders.Cash:
                    PrintReceipt();
                    break;

                default:
                    Console.WriteLine("\nSorry - Invalid tender.");
                    break;
            }
        }
    }

    class GotoCaseApp
    {
        public static void Main()
        {
            Payment payment = new Payment(Tenders.Visa);
            payment.ProcessPayment();
        }
    }
```

我们并不需要用一种很不直观的方法来解决这个问题，相反，只需要简单地告诉编译器，当关于信用卡或者是记账情况结束时，我们希望转移到 cash 标签。这里最后要注意的是，在 C# 中如果需要转移出一个 case 标签的话，不应该使用 break 语句，这会导致产生编译错误“unreachable code”。

goto 语句的最后一种格式使程序能够转移到 switch 语句的 default 标签处，从而我们又有了一种编写单个代码块的方法，可以用来作为 switch 语句的多个求值结果来执行。

11.3.5 return 语句

return 语句有两个功能。这条语句指定一个返回值，这个值被返回给当前执行代码的调用函数(在当前代码没有被定义为返回 null 时)，并且这导致了立即返回给调用函数。return 语句以下面的语法来定义：

return *[return-expression]*

当编译器遇到了方法的 return 语句时，而这条语句指定一个返回表达式，那么编译器就会计算是否可以把该返回表达式隐式地转换成与当前方法所定义的返回值相兼容的格式。转换后的结果被传递回调用函数。

如果使用 return 语句的时候还使用了异常处理，您就必须确保理解一些规则。如果 return 语句是一个 try 块，这个块中含有相联系的 finally 块，控制实际上被传递到 finally 块的第一行，并且当代码块结束时，控制被传递回调用函数。如果 try 块被嵌套在另一个 try 块的里面，控制将会继续以这种方式备份这种链，直到最终的 finally 块被执行。

11.4 本章小结

C#条件语句使您能够控制程序的流程。流程语句的三个不同的分类包括选择语句(例如 if 和 switch)、迭代语句(while、for 和 foreach)，以及各种不同的跳转语句(break、continue、goto 和 return)。根据本章的内容来判断使用语句的最佳方法，有助于写出结构更为清晰，并且也更易于维护的应用程序。

第 12 章　用异常进行错误处理

.NET 公共语言运行环境的一个主要目标就是能够避免在运行时产生错误(例如在使用受控代码时，使用诸如自动内存和资源管理之类的特性)，或者至少是在编译时就(利用强类型系统)发现这些错误。但是，某些错误只可能在运行时发生，因此，在所有遵循公共语言规范(Common Language Specification，CLS)的语言上都必须有某种一致性的方法来处理错误。在这个意义上，本章讨论的焦点就在于由 CLR 实现的错误处理系统——异常　处理。

在本章中，您首先将了解异常处理的常规机制以及基本语法。有了这些基础知识以后，您就可以理解如何把异常处理与现在更为普遍的错误处理方法相比较，并且可以发现异常处理相对于其他技术的优点。接下来我们将深入下去，讨论一些更为具体的.NET 异常处理问题，例如使用 Exception 类，并且派生自己的 Exception 类。然后，本章的最后一部分将涉及到正确地设计系统以便于使用异常处理的问题。

12.1　异常处理概览

异常是一种错误条件，当代码路径的正常控制流程——也就是说，调用堆栈上的一系列方法调用——不符合实际或是不正确时，异常会发生。理解异常和所预期事件(例如到达文件末尾)之间的区别是十分有必要的。如果用一个方法来按顺序地读取一个文件，那么您是知道程序在某个时候会读取到文件末尾的。因此，该事件在本质上来说就不是“异常”的，当然也就不能阻止应用程序继续运行下去。然而，如果您正在试图从头到尾地读取一个文件，而操作系统发出了警告，提示检查发现有磁盘错误，这当然是一种异常情况，而且当方法试图继续读取文件时，会影响到它的正常流程。

绝大部分异常还会涉及到另外一个问题：上下文环境。让我们来看一个例子。假设您正在写一段紧耦合(cohesive)的代码——在这种代码中，一个方法负责执行一个动作——您的代码可能会是这样的：

```
public void Foo()
{
    File file = OpenFile(String fileName);
    while (!file.IsEOF())
    {
        String record = file.ReadRecord();
```

```
        }
    CloseFile();
    }
    public void OpenFile(String fileName)
    {
        // Attempt to lock and open file.
    }
```

注意，如果方法 OpenFile 失败了，该方法本身是无法对错误进行处理的。这是因为该方法的惟一职责就是打开文件。当文件无法打开时，它不能判断失败是由于很严重的错误还是一个小错误而造成的。因此，OpenFile 方法无法对这种错误情况进行处理，原因正如大家所说的那样，在于该方法没有处于正确的上下文环境之中。

这就是为什么会出现异常处理的原因：一个方法判断出某种异常条件得到满足，并且正好没有处于能处理该错误的正确上下文环境之中。该方法就向运行环境发送一个信号，通知运行时有错误发生。而运行环境逆序遍历整个调用堆栈，直到找到一个可以正确处理该错误条件的方法。很明显，当代码路径深度达到五层方法调用，而第五个方法满足了错误条件，但是只有第一个方法才能正确地处理该错误时，这就变得尤为重要了。现在让我们来看一看异常处理中所用到的语法。

12.2 异常处理的基本语法

异常处理只包含四个关键字：try、catch、throw 和 finally。这些关键字的使用方法很简单，也很“直截了当”。当一个方法无法实现其目的，并且不能继续执行下去时——也就是说，当该方法检测到一种异常情况时——它就使用关键字 throw，向调用它的方法抛出一个异常。而调用该方法的方法，假设它有足够的上下文环境来处理该异常，那么就利用关键字 catch 来接收该异常，并判断应该采取什么样的动作。在下面的章节中，我们将会就如何抛出以及截获一个异常来讨论 C# 语言的语义，并且还会用一些代码段来演示异常是如何工作的。

12.2.1 抛出异常

当一个方法需要通知调用方法有一个错误发生时，该方法以下列方式使用关键字 throw：

throw *statement*;

throw *expression* opt;

稍后我们将进一步介绍抛出异常的几种不同方法。但就现在而言，只要在抛出异常时

能够意识到这一点就可以了，您需要抛出一个 System.Exception 类型(或是派生类)的对象。接下来的一个例子里，该方法判断有一个不可恢复的错误发生了，而它需要向调用它的方法抛出一个异常。请注意该方法如何实例化一个新的 System.Exception 对象，并将这个新创建的对象抛回到调用堆栈。

```
public void SomeMethod()
{
// Some error determined.
throw new Exception();
}
```

12.2.2 截获异常

很明显，因为一个方法可以抛出异常，那么必须有相应的部分，在这部分代码中，要有一些语句来截获这个被抛出的异常。关键字 Catch 定义了一段代码，当截获了一个给定类型的异常时，就处理这段代码。这个代码段中的代码被称为异常处理器(exception handler)。

我们要记住的一点是，并非每一个方法都需要对每种可能被抛出的异常进行处理，特别是，因为该方法可能并没有处理这个错误信息所需要的上下文环境。毕竟，异常处理的要点在于，有些代码拥有处理错误所需要的上下文信息，应该由这些代码来进行异常处理——请参阅本章的后面部分“在正确的上下文中处理错误”。至于现在，让我们来介绍另外一种情况，其中，某个方法要截获从被调用方法抛出的所有异常。我们使用两个关键字来截获异常：try 和 catch。

为了截获一个异常，您需要把用于执行的代码在 try 块中括起来，然后指定 try 块内的代码可以处理 catch 块中哪一种类型的异常。除非有一个被调用的方法抛出了一个异常，否则对 try 块中所有语句的处理都是像其他语句一样按顺序进行。如果有一个被调用的方法抛出了一个异常，那么控制被传递到合适的 catch 块的第一行。至于“合适的 catch 块”，意思是指这个 catch 块中定义了这种被抛出的异常类型。下面的例子中，Foo 方法调用了另外一个方法(Bar)，并截获了由该方法所抛出的异常：

```
public void Foo()
{
    try
    {
        Bar();
    }
    catch(System.Exception e)
    {
        // Do something with error condition.
```

```
        }
    }
```

现在您的下一个问题可能是，“如果方法 Bar 抛出了一个异常，而方法 Foo 又没有截获该异常，那么会发生什么事？”(如果对于 Bar 方法的调用不在 try 块中，就会出现这种情况)。其结果取决于应用程序的设计。当一个异常被抛出，控制被传递回调用堆栈，直到找到一个方法，该方法含有合适的 catch 块，可以处理这种被抛出的异常类型。如果没有找到一个带有合适的 catch 块的方法，该应用程序就终止了。因此，如果一个方法调用另外一个抛出了异常的方法，那么在系统的设计中，就必须有某个方法在调用堆栈中处理该异常。

12.2.3　重抛异常

有时候会出现这样的情况，一个方法截获了一个异常并在其上下文环境中完成了所能完成的一切工作，然后把该异常重新抛回到调用堆栈中，要这么做很简单，只需要使用关键字 throw 就可以了。这里有一个例子对此作了演示：

```
using System;

class RethrowApp
{
    static public void Main()
    {
        Rethrow rethrow = new Rethrow();

        try
        {
            rethrow.Foo();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception)
        {
```

```
            // Handle error.
            throw;
        }
    }

    public void Bar()
    {
        throw new Exception("thrown by Rethrow.Bar");
    }
}
```

在这个例子中，Main 调用 Foo，而 Bar 调用 Bar。Bar 抛出一个异常，该异常被 Foo 截获。在这里，方法 Foo 进行了一些处理，然后利用关键字 throw 把该异常重新抛回到对 Main 的调用堆栈中。

12.2.4 用 finally 进行清除

有一个与异常处理相关的事项，就是要确保无论异常是否被截获，都要有一段代码总是在运行。例如，假设您要对资源进行分配，例如物理设备或是数据文件。然后假设您打开这个资源，并调用一个方法，该方法抛出一个异常。无论您的方法是否能继续利用这种资源，您都需要解除分配或是关闭这种资源。这时就要用到关键字 finally，如下所示：

```
using System;

public class ThrowException1App
{
    public static void ThrowException()
    {
        throw new Exception();
    }

    public static void Main()
    {
        try
        {
            Console.WriteLine("try...");
        }
        catch(Exception e)
        {
            Console.WriteLine("catch...");
        }
        finally
```

```
            {
                Console.WriteLine("finally");
            }
        }
    }
```

正如您所见到的那样，关键字 finally 可以防止您不得不既在 catch 块中放入清除代码，同时还要在 try/catch 块后加入清除代码。现在，无论一个异常是否被抛出，finally 块中的代码都会被执行。

12.3 错误处理技术之比较

既然我们已经理解了抛出和截获异常的基本部分，那么让我们花一些时间来对编程语言中采用的不同错误处理方法进行比较。

一般来说，错误处理的标准方法是向调用方法返回错误代码。由调用方法负责对返回值进行解释，并采取相应的行动。返回值可能是简单的 C 或 C＋＋类型，也可能是指向一个复杂对象的指针，该对象里含有完全识别和理解该错误所必需的所有信息。在有些设计得更为精巧的错误处理技术中，还包含了一个完整的错误子系统。在该错误子系统中，被调用方法向子系统指出错误条件，然后向调用函数返回一个错误代码。然后调用函数调用一个全局函数，该全局函数是从错误子系统中导入的，其目的是为了判断注册到该子系统的最新错误的产生原因。在 Microsoft 开放式数据库连接(ODBC)SDK 中就有这种方法的实例。然而，无论精确的语义是什么，基本的概念仍然是相同的：调用函数以某种途径来调用一个方法，并且检查返回值，检验被调用方法的相关成功或失败。虽然多年以来这都是一种标准的错误处理方法，但是在许多方面都有重大的缺陷。接下来的几节讨论一些异常处理的方法，相对于使用返回代码而言，这些方法提供了更多的优点。

12.3.1 异常处理相对于返回代码的优点

当使用返回代码的方法时，被调用的方法返回一个错误代码，而调用方法则处理错误条件。因为错误处理发生在被调用方法的作用域以外，所以无法保证调用函数会对被返回的错误代码进行检查。作为一个例子，假设让我们来写一个名为 CommaDelimitedFile 的类，该类封装了读取和写入标准逗号分界文件的功能。类中需要向外暴露的部分包括打开文件的方法，以及从文件中取数据的方法。如果在报告错误时采取早期那种返回代码的方法，这些方法就会返回一些变量类型，而调用函数则不得不对这些变量进行检查，判断方法调用是否成功。如果该类的用户调用了方法 CommaDelimitedFile.Open，然后试图调用方法

CommaDelimitedFile.Read，在调用的时候并不检查文件是否被成功地打开了，这就可能导致——往往有可能是您在最重要客户端面前进行演示的时候——出现我们所不希望看见的结果。然而，如果该类的 Open 方法抛出一个异常，那么调用函数将会被强制执行，来对 Open 方法的失败进行处理。这是因为，每次方法抛出一个异常时，控制将会被传递到调用堆栈，直到该异常被截获。下面是一个类似代码的例子：

```
using System;

class ThrowException2App
{
    class CommaDelimitedFile
    {
        protected string fileName;

        public void Open(string fileName)
        {
            this.fileName = fileName;

            // Attempt to open file
            // and throw exception upon error condition.
            throw new Exception("open failed");
        }

        public bool Read(string record)
        {
            // Code to read file.
            return false; // EOF
        }
    }

    public static void Main()
    {
        try
        {
            Console.WriteLine("attempting to open file");

            CommaDelimitedFile file = new CommaDelimitedFile();
            file.Open("c:\\test.csv");

            string record = "";

            Console.WriteLine("reading from file");
```

```
            while (file.Read(record) == true)
            {
                Console.WriteLine(record);
            }

            Console.WriteLine("finished reading file");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

在这个例子中，如果 CommaDelimitedFile.Open 方式或 CommaDelimitedFile.Read 方法抛出了一个异常，那么调用方法被强迫处理该异常。如果这个调用方法没有截获该异常，而且在当前代码路径中也没有其他方法来试图截获这种类型的异常，那么这个应用程序就会被终止。要特别注意这样一个事实——因为 Open 方法是放在一个代码块中的，程序不会去尝试无效的读取(使用我们介绍过的例子，其中的 Open 方法抛出了一个异常)。这是因为程序控制会被从 try 块中的 Open 调用传递到 catch 块的第一行。因此，异常处理相对返回代码的一个最大优点就是，在编程时异常更加难以被忽视。

12.3.2 在正确的上下文环境中处理错误

关于良好的编程习惯的一个通用法则是程序的紧耦合度，紧耦合(tight cohesion)指的是给定方法的目标或目的。那些执行单个任务的方法体现了紧耦合的思想。在程序中使用紧耦合的主要优点在于，当方法只执行单个任务时，这样的方法移植性更强，也就更易于在不同的环境中使用。执行单个动作的方法当然也更加易于调试和维护。然而，紧耦合确实也导致了一个与错误处理有关的主要问题。让我们看一个例子，对这个问题以及异常处理如何解决这个问题加以演示。

在这个例子中，一个类(AccessDatabase)被用于生成和处理 Microsoft Access 数据库。让我们假设这个类有一个静态方法 GenerateDatabase。因为 GenerateDatabase 方法将被用于创建新的 Access 数据库，所以这个方法必须执行几个任务来创建数据库。例如，这个方法必须创建物理数据库文件，创建指定的表(包括应用程序所需要用到的行和列)，还要定义所有必需的索引和关系。这个 GenerateDatabase 方法甚至可能不得不创建一些默认用户和许可。

下面是一些程序设计问题：如果在方法 CreateIndexes 中将发生一个错误，那么将是什么方法来处理这个错误，而又如何处理呢？很明显，在程序的某些地方，一开始调用

GenerateDatabase 方法的那个方法将不得不处理这个错误，但是如何处理呢？错误发生在代码路径的几个深度方法调用之中，这个方法可能并不知道如何对这个发生的错误进行处理。正如我们所看到的那样，调用方法不是在正确的上下文环境中处理错误的。换句话说，逻辑上创建任何有意义的错误信息的惟一方法就是失败方法自身。前面已经提到过，如果在我们的 AccessDatabase 类中使用的是返回错误代码的方法，代码路径中的每个方法都必须对其他方法可能返回的每个错误代码进行检查。与此相关的一个很明显的问题就是，调用方法可能不得不处理大量的错误代码。此外，程序的维护也可能会很困难。每次向代码路径中的所有方法加入一个错误条件时，应用程序中的每一个实例，只要含有调用了那个方法的方法，这些实例就都必须被修改，以便于处理新的错误代码。根据软件的总体拥有成本(total cost of ownership，TCO)概念，这种方式绝对不会是代价低廉的。

异常处理使得调用方法能够截获一种给定的异常类型，这就解决了所有这些问题。在我们将要提到的例子里，如果一个名为 AccessDatabaseException 的类是从 Exception 类派生的，那么在任何 AccessDatabase 方法中发生的任何类型的错误都可以用这个类来处理(在本章后面的“使用 SystemException 类”部分中，我们将会讨论 Exception 类以及派生自己的 Exception 类)。然后，如果 CreateIndexes 方法失败了，这个方法就会构造并抛出一个 AccessDatabaseException 类型的异常。而调用方法将会截获这个异常，并且能够检查 Exception 对象，这样就可以精确地解释所发生的错误。因此，如果代码路径中的任何方法发生了错误，都可以确保调用方法能够得到合适的返回错误信息，而不是对 GenerateDatabase 以及其中任何被调用方法返回的可能错误代码类型进行处理。异常处理还提供了另一个好处：因为错误信息是包含在类中的，那么可以在类中加入新的错误条件，而调用方法则可以保持不变。而面向对象编程起初的时候不正是要保证可扩展性吗(可扩展性指的是能够创建代码，并且可以在其中加入新的代码，而不需要破坏或者改变现有代码)？由于这些原因，在正确的上下文中截获并处理错误的概念是使用异常处理最为显著的优点。

12.3.3 提高代码的可读性

当使用异常处理代码时，代码可读性能够得到显著的提高。这与减少代码维护成本直接相关。返回代码处理的方式与异常处理语法之间的竞争是这种改进的源泉。如果您对前面提到的 AccessDatabase.GenerateDatabase 方法使用了返回代码，那么处理错误条件所需的相似代码如下所示：

```
public bool GenerateDatabase()
{
    if (CreatePhysicalDatabase())
    {
```

```
            if (CreateTables())
            {
                if (CreateIndexes())
                {
                    return true;
                }
                else
                {
                    // Handle error.
                    return false;
                }

            }
            else
            {
                // Handle error.
                return false;
            }
        }
        else
        {
            // Handle error.
            return false;
        }
    }
```

要想在前面的代码中加入其他一些确认代码，您就会被与商业逻辑混在一起的大量错误确认代码弄得紧张不堪。如果每个代码块的代码都缩进 4 格，那么代码中可能会有出现这种情况，某行程序的第一个字母要在 20 列以后才会出现。虽然这对代码本身没有什么损害，但是这使得代码更加难以读懂及维护。而底线就是，那些难以维护的代码是产生程序错误的温床。让我们来看一下，如果使用异常处理，同一个例子的代码是什么样的：

```
// Calling code.
try
{
    AccessDatabase accessDb = new AccessDatabase();
    accessDb.GenerateDatabase();
}
catch(Exception e)
{
    // Inspect caught exception.
}

// Definition of AccessDatabase.GenerateDatabase method.
```

```
public void GenerateDatabase()
{
    CreatePhysicalDatabase();
    CreateTables();
    CreateIndexes();
}
```

请注意第二种解决方案是多么简洁明了。这是因为错误检测和恢复代码不再与调用代码自身的逻辑混在一起的缘故。因为异常处理使得代码更加直接，所以维护起来就更加容易。

12.3.4 从构造函数抛出异常

相对于其他错误处理技术而言，在对象构造部分存在着异常处理的另一个优点。因为一个构造函数无法返回值，所以没有什么简单易行的方法来向构造函数的调用者发出信号，通知这个调用方法在对象构造时发生了错误。然而，我们可以使用异常来做到这一点，因为调用方法只需要在一个 try 块中加入对象构造的代码即可。这些代码如下所示：

```
try
{
    // If the AccessDatabase object fails to construct
    // properly and throws an exception, it will now be caught.
    AccessDatabase accessDb = new AccessDatabase();
}
catch(Exception e)
{
    // Inspect the caught exception.
}
```

12.4 使用 System.Exception 类

正如前面所提到的，所有将要被抛出的异常都必须是 System.Exception 类型的(或者是从 System.Exception 类派生出来的)。事实上，类 System.Exception 是可以在 C# 代码中使用的几个异常类的基本类。大部分从 System.Exception 继承的类都没有在基础类上增加新的功能。既然派生类与其基础类相比没有什么显著的不同，为什么 C# 还要使用派生类呢？原因就在于单个 try 块可以有多个 catch 块，而每个 catch 块都指定一个特定的异常类型(很快我们就会看到这一点)。这使得代码能够以一种应用于特定异常类型的方式来处理不同的异常。

12.4.1 构造一个 Exception 对象

要想构造一个 System.Exception 对象，可以使用四种不同的构造函数：

```
public Exception ();
public Exception(String);
protected Exception(SerializationInfo, StreamingContext);
public Exception(String, Exception);
```

上列的第一个构造函数是默认构造函数。这个构造函数没有参数，并且所有的成员变量都是默认的。典型地，一个异常是这样抛出的：

```
// Error condition reached.
throw new Exception();
```

第二个异常构造函数采用一个字符串作为其惟一的参数，这个字符串标识了一个错误消息，本章的大多数例子中都采用了这种形式的构造函数。利用 System.Exception.Message 属性，截获异常的代码可以检索到相应的错误消息。

```
using System;

class ThrowException3
{
    class FileOps
    {
        public void FileOpen(String fileName)
        {
            // ...
            throw new Exception("Oh bother");
        }

        public void FileRead()
        {
        }
    }

    public static void Main()
    {
        // Code catching exception.
        try
        {
            FileOps fileOps = new FileOps();
```

```
            fileOps.FileOpen("c:\\test.txt");
            fileOps.FileRead();
        }
        catch(System.Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

第三个构造函数用串行化数据构造了一个 Exception 类的实例。

最后，第四个构造函数不仅指定了错误消息，还指定了一个内部异常(inner exception)。使用这个构造函数的原因在于，当处理异常时，在把异常沿着调用堆栈向上传递之前，您可能要在调用堆栈的级别上对这个异常做进一步的处理。为了演示如何使用这个特性，让我们来看一个例子。

假设为了减轻类的客户端的负担，您决定只抛出一种类型的异常。这样的话，类的客户端就只需要截获一种异常，也只需要异常的一种InnerException属性引用。如果在客户端代码已经写完之后，您又决定修改一个给定方法的代码，以抛出一种新的异常类型，这个构造函数可以提供额外的好处——除非客户端代码要做某些与这个异常有关的特定工作，否则无需修改这些代码。

在下面这个例子中，请注意客户端代码截获了顶级 System.Exception，并且打印出该异常的内部异常对象中所包含的消息。如果以后 DoWork 方法抛出了其他类型的异常——只要这个方法处理时作为 System.Exception 对象的内部异常——客户端代码就可以继续正常执行。

```
using System;
using System.Globalization;

class FooLib
{
    protected bool IsValidParam(string value)
    {
        bool success = false;

        if (value.Length == 3)
        {
            char c1 = value[0];
            if (Char.IsNumber(c1))
            {
                char c2 = value[2];
                if (Char.IsNumber(c2))
```

```
                {
                    if (value[1] == '.')
                        success = true;
                }
            }
        }

        return success;
    }

    public void DoWork(string value)
    {
        if (!IsValidParam(value))throw new Exception
                ("", new FormatException("Invalid parameter specified"));
        Console.WriteLine("Work done with '{0}'", value);
    }
}

class FooLibClientApp
{
    public static void Main(string[] args)
    {
        FooLib lib = new FooLib();
        try
        {
            lib.DoWork(args[0]);
        }
        catch(Exception e)
        {
            Exception inner = e.InnerException;
            Console.WriteLine(inner.Message);
        }
    }
}
```

12.4.2　使用 StackTrace 属性

System.Exception 类的另外一个很有用的属性就是 StackTrace 属性。StackTrace 属性使您能够判断出——在任何具有一个有效的 System.Exception 对象的给定点处——当前的调用堆栈是怎样的。请看下面的代码：

```
using System;
```

```
class StackTraceTestApp
{
    public void Open(String fileName)
    {
        Lock(fileName);
        // ...
    }
    public void Lock(String fileName)

    {
        // Error condition raised.
        throw new Exception("failed to lock file");
    }
    public static void Main()
    {
        StackTraceTestApp test = new StackTraceTestApp();

        try
        {
            test.Open("c:\\test.txt");
            // Use file.
        }
        catch(Exception e)
        {
            Console.WriteLine(e.StackTrace);
        }
    }
}
```

在这个例子中，属性 StackTrace 返回了异常被截获时的调用堆栈，这对于记录和调试环境是很有用的。

12.4.3 截获多个异常类型

在另外一些情况下，您可能想用一个 try 块来截获多种不同的异常类型。例如，一个给定的方法可能会抛出几种不同的异常类型，或者是您想在一个 try 块中进行几个方法调用，其中每个被调用的方法都能够抛出一个不同的异常类型。要想处理这种情况，可以在需要处理异常的代码处为每一种异常类型增加一个 catch 块。

```
try
{
    Foo(); // Can throw FooException.
    Bar(); // Can throw BarException.
```

```
}
catch(FooException e)
{
    // Handle the error.
}
catch(BarException e)
{
    // Handle the error.
}

catch(Exception e)
{
}
```

现在每一种异常类型都可以用一个单独的 catch 块(和错误处理代码)来处理。但是，这里有一个极为重要的细节需要注意——基本类是最后才被处理的。很明显，因为所有的异常都是从 System.Exception 类派生的，如果您要把 catch 块放在最前面，就会不可到达其他的 catch 块。因为这个原因，下列代码将会被编译器拒绝：

```
try
{
    Foo(); // Can throw FooException.
    Bar(); // Can throw BarException.
}
catch(Exception e)
{
    // ***ERROR - THIS WON'T COMPILE
}
catch(FooException e)
{
    // Handle the error.
}
catch(BarException e)
{
    // Handle the error.
}
```

12.4.4　派生出自己的 Exception 类

正如前面说过的，在把异常抛出给客户端代码以前，有时候您可能会想要提供额外的错误信息或是格式化的错误信息。您可能也想为这种情况提供一个基本类，这样的话，您的类就可以发布这样的事实——它抛出了一种异常类型。这样，客户端代码就只需要注意

截获这个基本类。

还有另外一种情况，如果每次异常被抛出时，您想要程序中的代码执行一些动作——例如记录事件或是向帮助桌面发送一封电子邮件——您可能也需要派生自己的 Exception 类。在这种情况下，您就需要派生自己的 Exception 类，并且把所需要的代码加入到这个类的构造函数中，就像这样：

```
using System;

public class TestException : Exception

{
    // You would probably have extra methods and properties
    // here that augment the .NET Exception that you derive from.

    // Base Exception class constructors.
    public TestException()
        :base() {}
    public TestException(String message)
        :base(message) {}
    public TestException(String message, Exception innerException)
        :base(message, innerException) {}
}
public class DerivedExceptionTestApp
{
    public static void ThrowException()
    {
        throw new TestException("error condition");
    }
    public static void Main()
    {
        try
        {
            ThrowException();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.ToString());
        }
    }
}
```

派生自己的 Exception 类

虽然不是严格的规则，但是这是一种良好的编程风格——并且与您所见过的绝大多数 C#代码相一致——在为自己的 exception 类命名时，在名字的结尾加上单词“Exception”。例如，如果您想要为一个名为 MyFancyGraphics 的类派生一个异常类，您应该把这个类命名为 MyFancyGraphicsException。

当创建或派生自己的 Exception 类时，另一个规则就是：实现所有的三个 System.Exception 构造函数。同样，这也不是必须的，但是这种做法可以与您的客户端所使用的其他 C#代码保持一致。

这些代码将会产生以下输出。注意，方法 ToString 导致了多个属性组合在一起的显示：异常类名的文字表示、传递给异常的构造函数的消息字符串以及 StackTrace 属性。

```
TestException: error condition
at DerivedExceptionTestApp.Main()
```

12.5 设计具有异常处理功能的代码

到目前为止，我们已经介绍了使用异常处理的基本概念以及与抛出和截获异常相关的语义。现在，让我们来看一看异常处理一个相当重要的方面：理解如何利用异常处理来设计自己的代码。假设现在有三个方法：Foo、Bar 和 Baz。Foo 调用 Bar，而 Bar 则调用 Baz。如果 Baz 发布一个事实：宣布抛出了一个异常，那么即使 Bar 无法处理这个异常，是不是也一定要截获该异常并对此做一些处理工作呢？关于 Try 和 catch 块的代码是如何划分的呢？

12.5.1 使用 try 块时的设计注意事项

您已经知道了该如何截获一个被调用方法可能抛出的异常，此外，您也已经知道，控制会在调用堆栈里向上传递，直到找到一个合适的 Catch 块。所以问题就在于，一个 Try 块是不是应该截获其中被调用方法可能抛出的所有异常呢？答案是不，这是为了在程序中维护使用异常处理的最大好处：减少代码并降低维护成本。下面的例子演示了一个程序，其中有一个 catch 块重新抛出了异常，这个异常被另一个 catch 块处理。

```
using System;

class WhenNotToCatchApp
{
```

```
    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Bar()
    {
        try
        {

            Baz();
        }
        catch(Exception e)
        {
// Bar should catch this because it
// doesn't do anything but rethrow it.

throw;
        }
    }

    public void Baz()
    {
        throw new Exception("Exception originally thrown by Baz");
    }

    public static void Main()
    {
        WhenNotToCatchApp test = new WhenNotToCatchApp();
        test.Foo(); // This method will ultimately
                    // print the error message.
    }
}
```

在这个例子中，Foo 截获了 Baz 所抛出的异常 ，虽然 Foo 什么也没有做，只是把这个异常重新抛出给 Bar。而 Bar 就截获了这个被重新抛出的异常，并对信息进行了一些处理——在这个例子里，Bar 显示了 Exception 对象的错误消息属性。接下来的几个原因解释了为

什么仅仅重抛异常的方法不应该首先截获那个异常：

- 因为这个方法对所截获的异常不进行任何处理，所以最好是把方法中的多余 catch 块去掉。显然，在程序中保留一些什么功能都没有的代码并不是一个好主意。
- 如果要对 Baz 所抛出的异常类型加以改变，您必须对 Bar 的 catch 块和 Foo 的 catch 块都加以修改。没有必要把自己置于这样的境地：必须修改那些什么作用也没有的代码。

因为 CLR 自动向上查找调用堆栈，直到有一个方法截获这个异常，所以中间方法可以忽略那些它们所无法处理的异常。只要确保在设计中有某个方法截获了这个异常就可以了，这是因为如我们以前所提到的，如果一个异常被抛出，而在当前调用堆栈中又没有找到一个 catch 块来处理这个异常，程序就会终止。

12.5.2 使用 catch 块时的设计注意事项

在一个 catch 块中出现的代码都应该至少部分地处理了所截获的异常。例如，有时候一个方法会截获一个异常，尽其所能来处理该异常，并把该异常重新抛出，这样可以进行进一步的错误处理。下面的例子演示了这一点。Foo 调用了 Bar，而 Bar 将为 Foo 完成一些与数据库有关的工作。因为要使用提交控制或是事务，Bar 需要截获所发生的错误，并在向 Foo 抛出该异常以前回滚那些没有提交的改变。

```
using System;

class WhenToCatchApp
{
    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Bar()
    {
        try
        {
            // Call method to set a "commitment boundary."
```

```
            Console.WriteLine("setting commitment boundary");

            // Call Baz to save data.
            Console.WriteLine("calling Baz to save data");
            Baz();

            Console.WriteLine("commiting saved data");
        }
        catch(Exception)
        {
                // In this case, Bar should catch the exception
                // because it's doing something significant
                //(rolling back uncommitted database changes).

                Console.WriteLine("rolling back uncommited changes " +

                        "and then rethrowing exception");

            throw;
        }
    }

    public void Baz()
    {
        throw new Exception("db failure in Baz");
    }

    public static void Main()
    {
        WhenToCatchApp test = new WhenToCatchApp();
        test.Foo(); // This method will ultimately print
                    // the error msg.
    }
}
```

Bar 需要截获该异常，以便于进行自己的错误处理。当错误处理完成以后，Bar 方法就向调用堆栈重新抛出该异常，在这个调用堆栈中，Foo 截获了该异常，然后可以对其进行错误处理。调用堆栈的每一级都根据给定的上下文，只对错误中其能够完成的部分进行处理，这种能力使得异常处理对于系统设计来说非常重要。

12.6 本章小结

C#异常处理语法很简单明了，而在应用程序中实现异常处理与预先设计方法一样容易。.NET CLR 的一个主要目标就是有助于防止在强类型系统中通过一个运行时的错误，并且可以对运行时发生的错误进行适当的处理。在程序中使用异常处理代码将使得程序的健壮性、可靠性更好，并且也更加易于维护。

第 13 章　操作符重载和用户定义的转换

在第 7 章中，您已经学习了如何对类使用[]操作符，对一个对象进行编程索引，就仿佛它是一个数组。在这一章中，我们将会看一看 C#中两个关系密切的特性，这两个特性提供了创建结构体和类接口的功能，使结构体和类的接口使用起来更为容易，且更为自然。这两个特性就是操作符重载和用户定义的转换。一开始，我们将按照操作符重载的优点对其进行总体上的概览，而且还将看一看用于重新定义默认行为的实际语法。另外，我们还会介绍一个例子，例子中重载了+操作符，使多个“Invoice”对象可以相加。在此之后，您将会看到一个列表，列出的是可以重载的二元和一元操作符，以及其中涉及到的一些限制。最后，在结束对操作符重载的讨论之前，我们将提出一些设计指导方针，当您决定是否要在自己的类中重载操作符时，需要参考这些设计指导方针。学完操作符重载后，您就将开始学习一种新的概念——用户定义的转换。接下来，我们将首先大致地解释这种功能的基础，然后深入地探讨一个类，来演示如何使用转换，使一个结构体或类能计算另一种结构体、类或者 C#类型。

13.1　操作符重载

操作符重载允许对于现存的 C#操作符进行重新定义，使经过重新定义后的操作符能使用用户定义的类型。操作符重载已经被称为“syntactic sugar”(“依据造句法的糖”)，因为重载的操作符只不过是调用某个方法的另一种手段而已。也有人说这种功能从根本上说并没有为语言增加任何东西。尽管从技术上来说，这种说法是正确的，但是操作符重载确实对面向对象编程最重要的一个方面——即“抽象” ——有所帮助。

假设您想为某个特定的用户把一大堆发票加起来。如果使用操作符重载的话，写出的代码就与下面重载+操作符的代码相似：

```
Invoice summaryInvoice = new Invoice();
foreach (Invoice invoice in customer.GetInvoices())
{
    summaryInvoice += invoice;
}
```

这种代码的优点在于其语法的自然性，并且用户可以抽象地看待这些对象，不必再去

理解将这些发票加起来求和的实现细节。简而言之，操作符重载可以使软件编写和维护的代价大为减少。

13.1.1 语法和例子

正如我们所说的，操作符重载是一种调用一个方法的手段。要想为一个类重新定义一个操作符，您只需要使用以下模式，其中 op 是您要重载的操作符：

public static *retval* operator *op* (*object1* [, *object2*])

在使用操作符重载时要记住以下事实：

- 所有重载的操作符方法都要定义为 public 和 static。
- 从技术上来说，retval(返回值)可以是任何类型。然而，通常的做法是返回所定义方法使用的类型，除了操作符 true 和 false 以外，这两个操作符总是应该返回一个布尔类型的值。
- 传递参数值(object1，object2)的数目取决于所重载操作符的类型。如果重载的是一个一元操作符(仅有一个操作数的操作符)，那么就只有一个参数值。如果重载的是一个二元操作符(对两个操作数进行运算的操作符)，那么就传递两个参数值。
- 在重载一元操作符的情况下，传递给方法的参数值类型必须与封装类或结构体的类型相同。换句话说，如果您为某个称为 Foo 的类重新定义了一元操作符!，那么重新定义的方法必须采用类型为 Foo 的变量作为参数值。
- 如果被重载的是二元操作符，第一个参数值的类型必须与封装类的类型相同，并且第二个参数值可以是任意类型。

在前面部分的伪代码中，我们对类 Invoice 使用了+=操作符。因为某些您很快就会理解的原因，我们实际上是无法重载复合操作符的。可以重载的只是一些“基本”操作符——在此例中即+。下面的语法用于定义类 Invoice 的操作符+方法：

```
public static Invoice operator+ (Invoice invoice1, Invoice invoice2)
{
    // Create a new Invoice object.
    // Add the desired contents from
    // invoice1 to the new Invoice object.
    // Add the desired contents from
    // invoice2 to the new Invoice object.
    // Return the new Invoice object.
 }
```

现在让我们来看一个更为真实的例子，这个例子里面有两个主要的类：Invoice 和 InvoiceDetailLine。Invoice 有一个 ArrayList 类型的成员变量，代表所有发票详细条目的集

合。考虑到要把多张发票的条目进行加和，我们重载了+操作符(细节请参见下面的 operator+方法)。方法 Invoice.operator+创建了一个新的 Invoice 对象，并且同时迭代两个 Invoice 对象的发票条目行数组，把各条目行都加入到新的 Invoice 对象中。然后，这个 Invoice 对象被返回给调用者。很明显，现实中的发票模块要比这复杂得多，但是这里重要的是要实际地显示操作符重载究竟是如何使用的。

```
using System;
using System.Collections;

class InvoiceDetailLine
{
    double lineTotal;
    public double LineTotal
    {
        get
        {
            return this.lineTotal;
        }
    }

    public InvoiceDetailLine(double LineTotal)
    {
        this.lineTotal = LineTotal;
    }
}

class Invoice
{
    public ArrayList DetailLines;

    public Invoice()
    {
        DetailLines = new ArrayList();
    }

    public void PrintInvoice()
    {
        Console.WriteLine("\nLine Nbr\tTotal");

        int i = 1;
        double total = 0;
        foreach(InvoiceDetailLine detailLine in DetailLines)
        {
```

```
            Console.WriteLine("{0}\t\t{1}", i++, detailLine.LineTotal);
            total += detailLine.LineTotal;
        }

        Console.WriteLine("=====\t\t===");
        Console.WriteLine("Total\t\t{1}", i++, total);
    }

    public static Invoice operator+ (Invoice invoice1, Invoice invoice2)
    {
        Invoice returnInvoice = new Invoice();

        foreach (InvoiceDetailLine detailLine in invoice1.DetailLines)
        {
            returnInvoice.DetailLines.Add(detailLine);
        }

        foreach (InvoiceDetailLine detailLine in invoice2.DetailLines)
        {
            returnInvoice.DetailLines.Add(detailLine);
        }
        return returnInvoice;
    }
}

class InvoiceAddApp
{
    public static void Main()
    {
        Invoice i1 = new Invoice();
        for (int i = 0; i < 2; i++)
        {
            i1.DetailLines.Add(new InvoiceDetailLine(i + 1));
        }

        Invoice i2 = new Invoice();
        for (int i = 0; i < 2; i++)
        {
            i2.DetailLines.Add(new InvoiceDetailLine(i + 1));
        }

        Invoice summaryInvoice = i1 + i2;

        summaryInvoice.PrintInvoice();
```

```
        }
    }
```

13.1.2 可重载操作符

只有下列一元操作符和二元操作符可以被重载。

一元操作符：+, -, !, ~, ++, --, *true*, *false*

二元操作符：+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=

注意： 这里所用的逗号是为了把不同的可重载操作符分隔开。用在语句和方法调用中的逗号是无法被重载的。

13.1.3 操作符重载的限制

要想重载赋值操作符+=是不可能的。但是，当您想要重载一个二元操作符时，这个二元操作符的复合赋值等式就被隐式地重载了。例如，如果要重载操作符+，那么操作符+=就被隐式地重载，因为那样的话，调用的就是用户定义的 operator+方法。

操作符[ ]无法被重载。但是，正如您在第 7 章所见到的那样，用户定义的对象索引是通过索引器来支持的。

用于执行计算的圆括号也是无法重载的。不过，您可以用转换操作符来代替，这也被称为用户定义的转换。本章后半部分主要介绍的就是用户定义的转换。

在 C#语言中目前未经定义的操作符无法重载。例如，您无法把**定义为一种求幂的方法，因为 C#中并没有定义一个**操作符。此外，操作符的语法是无法修改的。您无法把一个二元操作符*定义成一个使用三个参数值的操作符，因为这个操作符在定义时，语法要求用两个操作数。最后，操作符的优先级是无法被改变的。运算优先级的规则是静态的，要想了解更多关于优先级规则的细节，请参见第 10 章。

13.1.4 设计准则

现在您已经知道操作符重载是怎么回事了，并且也已经学会如何在 C#中使用操作符重载了，那么就让我们来仔细地看一看这个很有用的功能中常被忽视的方面：设计准则。您要避免的是人的一种自然倾向，即对于一个新功能的态度是为了使用而使用。这样的现象有时候被称为“头痛医头，脚痛医脚”。但是，要想进行好的设计，最好还是记住这样一句话：“编写代码的目的是要让别人读的”。当要决定是否重载以及何时重载一个操作符或操作符集时，请考虑到类的客户端。这里有一种处理方法：只有在重载操作符可以使类的接

口更易于使用时，我们才会重载操作符。例如，重载操作符使得对于 invoice 对象加和的处理接近完美。

此外，别忘了时时记住要尽量按照类的客户端的方式来思考问题。我们来设想一下，假如您要编写一个发票类 Invoice，并且还希望客户端能够为发票类 Invoice 打折。大家可能都知道发票里会加上一行信用条目，但是封装的意义就在于您的客户端并不需要知道类的实现细节。因此，重载操作符*——正如这里所示的——可能是一个好主意，因为这有助于使得类的接口更加自然。

```
invoice *= .95; // 5% discount.
```

13.2 用户定义的转换

前面我们提到过，用于进行计算的圆括号不能被重载，并且必须通过用户定义的转换来代替。简而言之，用户定义的转换使您能够对于那些可以被转换成其他的结构体和类的结构体、类和基本 C#类型进行声明转换。为什么要这样做？应该在何时这样做？假设一下，您需要在您的应用程序中使用标准的摄氏温度和华氏温度，在程序中还要很方便地在这两种温度之间进行转换。通过创建用户定义的转换，您就可以使用下面的语法：

```
Fahrenheit f = 98.6F;
Celsius c = (Celsius)f; // User-defined conversion.
```

这对以下语法虽然不具有任何功能上的意义，但它编写起来更为直观，且更易于阅读。

```
Fahrenheit f = 98.6F;
Celsius c = f.ConvertToCelsius();
```

语法和例子

用户定义的转换所用的语法要用到关键字 operator，这个关键字将被用来声明用户定义的转换：

public static implicit operator *conv-type-out* (*conv-type-in operand*)

public static explicit operator *conv-type-out* (*conv-type-in operand*)

关于用户定义的转换的语法只有以下几条：

- 对于结构体或类的任何转换方法——您可以定义任意多个方法——都必须是静态的(static)。
- 转换的定义要么是隐式的，要么就是显式的。关键字 implicit 表示不需要客户端的计算，它会自动进行。相反，使用关键字 explicit 表示客户端必须显式地计算所

需要类型的值。

- 所有的转换要么采用转换定义的类型(作为参数值)，要么必须返回这个类型。
- 和操作符重载一样，关键字 operator 被用于转换方法信号中，但是没有任何附加的操作符。

为便于理解上述规则，让我们来看一个例子，这样会使之更为明确。在这个例子中，我们用到了两个结构体(Celsius 和 Fabrenbeit)，这两个结构体能够使客户端将一个 float 类型的值转换成为一个温度范围内的值。我们首先介绍 Celsius 结构体，并且对其做出一些解释，然后您将会看到完整的应用程序。

```
struct Celsius
{
    public Celsius(float temp)
    {
        this.temp = temp;
    }

    public static implicit operator Celsius(float temp)
    {
        Celsius c;
        c = new Celsius(temp);
        return(c);
    }

    public static implicit operator float(Celsius c)
    {
        return((((c.temp - 32) / 9) * 5));
    }

    public float temp;
}
```

您所见到的第一个选择是我们决定使用一个结构体，而不是一个类。对此并没有什么其他的合理解释，除了一个原因：根据类的分配方式可以判断，使用类比使用结构体代价更大。不过，这里确实没有什么必要来使用类，因为 Celsius 结构体并不需要任何 C#特殊类的功能，比如说继承。

接下来要注意，我们已经声明了一个构造函数，这个函数把一个浮点数(float)作为其惟一的参数值。这个值被存放在一个名为 temp 的成员变量中。现在来看一看紧接着在定义了结构体构造函数之后定义的转换操作符。当客户端试图把一个浮点数加到 Celsius 中或是在某处使用一个浮点数——例如，在一个使用结构体 Clesius 的方法中。这个方法不必做许多繁琐的工作，事实上，这些代码是相当公式化的，绝大多数的基本转换中都可以使用这些

代码。在这里，我们只是简单地实例化一个 Celsius 结构体，并且返回这个结构体。这个返回调用就是调用结构体中定义的最后一个方法。正如您所见到的那样，这个方法只是简单地提供了一个数学公式，用于执行从 Fahrenheit 值向 Celsius 值的转换。

下面是完整的应用程序，包括一个 Fabrenbeit 结构体：

```
using System;

struct Celsius
{
    public Celsius(float temp)

    {
        this.temp = temp;
    }

    public static implicit operator Celsius(float temp)
    {
        Celsius c;
        c = new Celsius(temp);
        return(c);
    }

    public static implicit operator float(Celsius c)
    {
        return((((c.temp - 32) / 9) * 5));
    }

    public float temp;
}

struct Fahrenheit
{
    public Fahrenheit(float temp)
    {
        this.temp = temp;
    }

    public static implicit operator Fahrenheit(float temp)
    {
        Fahrenheit f;
        f = new Fahrenheit(temp);
        return(f);
    }
```

```
    public static implicit operator float(Fahrenheit f)
    {
        return((((f.temp * 9) / 5) + 32));
    }

    public float temp;
}

class Temp1App
{
    public static void Main()

    {
        float t;

        t=98.6F;
        Console.Write("Conversion of {0} to Celsius = ", t);
        Console.WriteLine((Celsius)t);

        t=0F;
        Console.Write("Conversion of {0} to Fahrenheit = ", t);
        Console.WriteLine((Fahrenheit)t);
    }
}
```

如果您编译并且执行这个应用程序，将会得到这样的输出：

```
Conversion of 98.6 to Celsius = 37
Conversion of 0 to Fahrenheit = 32
```

这个程序工作得很好，并且，能够写出(Celsius)98.6F 当然要比调用一些静态类方法更直接。但是要注意，只有一个 float 类型的值被传递给这些转换函数。对于上述程序，下面的语句无法编译：

```
Celsius c = new Celsius(55);
Console.WriteLine((Fahrenheit)c);
```

同样，因为这里没有使用一个 Fabrenbeit 结构体(反之亦然)的 Celsius 转换方法，代码不得不假设被传递来的值就是一个需要被转换的值。换句话说，如果我们调用(Clesius)98.6F，就会得到一个值 37。然而，如果这个值又被再次传递回转换函数，转换函数无从知晓这个值是已经被转换过了的，并且在逻辑上已经表示了一个有效的摄氏温度——对这个转换函数来说，这只是个浮点数。结果，这个值再次被转换。因此，我们需要修改应用程序，这样每个结构体都可以把其他结构体作为一个有效参数值加以接收。

虽然这一任务乍一想似乎很难，但实际上却相当容易。下面就是修改过的代码和注释：

```
using System;

class Temperature
{
    public Temperature(float Temp)
    {
        this.temp = Temp;
    }

    protected float temp;
    public float Temp
    {
        get
        {
            return this.temp;
        }
    }
}

class Celsius : Temperature
{
    public Celsius(float Temp)
        : base(Temp) {}

    public static implicit operator Celsius(float Temp)
    {
        return new Celsius(Temp);
    }

    public static implicit operator Celsius(Fahrenheit F)
    {
        return new Celsius(F.Temp);
    }

    public static implicit operator float(Celsius C)
    {
        return((((C.temp - 32) / 9) * 5));
    }
}

class Fahrenheit : Temperature
{
    public Fahrenheit(float Temp)
        : base(Temp) {}
```

```
        public static implicit operator Fahrenheit(float Temp)
        {
            return new Fahrenheit(Temp);
        }

        public static implicit operator Fahrenheit(Celsius C)
        {
            return new Fahrenheit(C.Temp);
        }

        public static implicit operator float(Fahrenheit F)

        {
            return((((F.temp * 9) / 5) + 32));
        }
    }

    class Temp2App
    {
        public static void DisplayTemp(Celsius Temp)
        {
            Console.Write("Conversion of {0} {1} to Fahrenheit = ",
                Temp.ToString(), Temp.Temp);
            Console.WriteLine((Fahrenheit)Temp);
        }

        public static void DisplayTemp(Fahrenheit Temp)
        {
            Console.Write("Conversion of {0} {1} to Celsius = ",
                Temp.ToString(), Temp.Temp);
            Console.WriteLine((Celsius)Temp);
        }

        public static void Main()
        {
            Fahrenheit f = new Fahrenheit(98.6F);
            DisplayTemp(f);

            Celsius c = new Celsius(0F);
            DisplayTemp(c);
        }
    }
```

这里第一件要注意的事就是，我们把 Celsius 和 Fabrenbeit 类型从结构体改成了类。这

样做就可以有两个例子了——一个使用结构体，还有一个使用类。但是，我们之所以这样做还有一个更为实际的原因。我们可以从同一个温度类 Temperature 中派生出 Celsius 和 Fabrenbeit 类。现在我们也可以在程序的输出中使用(从 System.Object)继承的 ToString 方法。

另一个值得注意的地方是，还有一个转换，用于把其他的温度标量的值作为参数值进行转换。注意，在这两个 Celsius 转换方法中的代码是多么相似。

```
public static implicit operator Celsius(float temp)
{
    Celsius c;
    c = new Celsius(temp);
    return(c);
}

public static implicit operator Celsius(Fahrenheit f)
{
    Celsius c;
    c = new Celsius(f.temp);
    return(c);
}
```

惟一要做的不同任务就是改变传递来的参数值，并从被传递来的对象而不是从类型为 float 的值中取出温度值。这就是为什么我们在前面会说，一旦您学会了基础，这些转换方法是多么简单，且多么公式化的原因。

13.3 本章小结

操作符重载和用户定义的转换对于类接口的直接实现是非常有用的。当使用被重载的操作符时，进行类的设计过程中要时时记住与之相关的一些限制。例如，您无法重载赋值操作符+=，但是当一个二元操作符被重载了，它的复合赋值等式会被隐式地重载。决定何时使用某种功能时，要遵循设计准则。当要决定是否重载一个操作符或操作符集合时，要时时考虑到类的客户端。只要对客户端使用类的方式有深入的了解，您就可以使用这些强大的功能来定义您的类，例如，可以以一种更为自然的语法来执行某些操作。

第 14 章　代表和事件处理器

C#中另外一种很有用的创新就是代表，代表的目的基本上与 C++中的函数指针相同。但是，代表是一种具有类型保护和安全机制的对象。这意味着在运行时刻能保证一个代表总是指向一个有效的方法，这又进一步表明，我们可以得到函数指针的所有优点，却不会带来与之相关的危险，例如一个无效的地址或是一个代表破坏其他对象的存储空间。在这一章里，我们将会仔细探讨代表，它们是如何与接口相比较的，定义代表的语法，以及它们被设计用来处理的不同问题。我们还会看几个使用代表的例子，这些例子中都有回调方法和异步事件处理。

在第 9 章中，我们已经看到，在 C#中是如何定义接口和实现的。从概念的角度来说，接口只是两段完全不同代码部分之间的简单约定。然而，接口更像是类，因为它们是在编译时被定义的，而且可以包含方法、属性、索引器以及事件。换句话说，在 C#编程中，代表有两个主要的用处：回调和事件处理。下面，我们将从对回调方法的讨论作为开始。

14.1　将代表用作回调方法

回调方法广泛应用于 Microsoft Windows 编程，当需要把一个函数指针传递给另一个接下来进行回调的函数(通过被传递过来的指针)时，就要用到回调方法。这里有一个 Win32 API 函数 EnumWindows 的例子。该函数列举所有屏幕上的顶层窗口，调用为每个窗口提供的函数。回调可以用于许多目的，下面几个目的是最为常见的：

- **异步处理**　当被调用的代码将占用大量时间来处理请求时，就要在异步处理中用到回调方法了。一种典型的工作方式是这样：客户端代码调用一个方法，把一个回调方法传递给它。被调用的方法启动一个线程，并且立即返回。然后，线程完成大部分工作，调用所需的回调函数。这样做的好处很明显：可以允许客户端继续处理，无需在一个可能的长同步调用上被阻塞。
- **把自定义代码加入类的代码路径中**　回调方法的另一个常见用法就是，一个类允许客户端指定一个方法，这个方法将被调用，来进行自定义处理。让我们用一个 Windows 中的例子来对此加以说明。通过在 Windows 使用类 Listbox，可以指定各项按照升序或降序排列。除了一些其他的排序选择以外，类 Listbox 并不能给出任何真正的范围并保持一个一般类。因此，类 Listbox 也可以使您能够指定一个回

调函数，用来排序。这样的话，当 Listbox 对各项进行排序时，它调用回调函数，这样就可以用自己的代码来进行自定义的排序。

现在让我们来看一个定义和使用代表的例子。在这个例子中，我们有一个数据库管理器的类，对所有数据库上的活跃连接保持追踪，并提供一个方法来列举这些连接。假设数据库管理器位于一个远程服务器上，这样也许是一个良好的设计决策，可以使得方法异步，并且允许客户端提供一个回调方法。注意，对于一个实际的应用程序来说，我们将其创建为典型的多线程应用程序，使其成为真正的异步方式。但是，为了使例子保持简单——而且也因为目前我们还没有介绍到多线程——我们先省略多线程。

首先，让我们来定义两个主要的类：DBManager 和 DBConnection。

```
class DBConnection
{
    ...
}

  class DBManager
  {
    static DBConnection[] activeConnections;
    ...

    public    delegate    void    EnumConnectionsCallback(DBConnection
connection);
      public   static   void   EnumConnections(EnumConnectionsCallback
callback)
      {
          foreach (DBConnection connection in activeConnections)
          {
              callback(connection);
          }
      }
  }
```

方法 EnumConnectionsCallback 就是代表，并且是通过把关键字代表放在方法名前面来定义的。我们可以发现，这个代表被定义为返回 void，并且只有一个参数值：一个 DBConnection 对象。然后，EnumConnections 方法被定义为把一个 EnumConnectionsCallback 方法作为其惟一的参数值。为了调用 DBManager.EnumConnections 方法，我们需要给它传递一个实例化的 DBManager.EnumConnectionCallback 代表。

为了这样做，需要用 new 来产生一个代表，把具有相同代表符号的方法名字传递给它。下面是一个这样的例子：

```
DBManager.EnumConnectionsCallback myCallback =
```

```
new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);
DBManager.EnumConnections(myCallback);
```

还要注意，可以把上面的语句在单个调用语句中联合使用，就像这样：

```
DBManager.EnumConnections(new
   DBManager.EnumConnectionsCallback(ActiveConnectionsCallback));
```

这里是所有的代表的基本语法。现在让我们来看一看完整的应用程序例子：

```
using System;

class DBConnection
{
   public DBConnection(string name)
   {
      this.name = name;
   }

     protected string Name;
     public string name
     {
        get
        {
           return this.Name;
        }
        set
        {
           this.Name = value;
        }
     }
  }
  class DBManager
  {
     static DBConnection[] activeConnections;
     public void AddConnections()
     {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
           activeConnections[i] =
new DBConnection("DBConnection " + (i + 1));
        }
     }
     public   delegate   void   EnumConnectionsCallback(DBConnection
```

```
connection);
        public   static   void   EnumConnections(EnumConnectionsCallback
callback)
        {
            foreach (DBConnection connection in activeConnections)
            {
                callback(connection);
            }
        }
    }

    class DelegateIApp
    {
        public   static   void   ActiveConnectionsCallback(DBConnection
connection)
        {
            Console.WriteLine("Callback method called for "
                              + connection.name);
        }

        public static void Main()
        {
            DBManager dbMgr = new DBManager();
            dbMgr.AddConnections();

        DBManager.EnumConnectionsCallback myCallback =
        new
DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

        DBManager.EnumConnections(myCallback);
        }
    }
```

编译并执行这个应用程序会产生下列输出：

```
Callback method called for DBConnection 1
Callback method called for DBConnection 2
Callback method called for DBConnection 3
Callback method called for DBConnection 4
Callback method called for DBConnection 5
```

14.2 把代表定义为静态成员

因为每次要用到代表时，客户端都不得不实例化一个代表，这显得太繁琐。C#允许把用于创建代表的方法定义为类的静态成员。下面的这个例子是上一节中用到过的，但是被

修改成使用现在这种格式。注意，代表现在被定义为名为 myCallback 类的静态成员。还要注意，这个成员可以在 main 方法中使用，无需客户端对代表进行实例化。

```
using System;

class DBConnection
{
    public DBConnection(string name)
    {
        this.name = name;
    }

    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }

        set
        {
            this.Name = value;
        }
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] = new
DBConnection("DBConnection " + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection
connection);
    public static void EnumConnections(EnumConnectionsCallback
callback)
```

```
        {
            foreach (DBConnection connection in activeConnections)
            {
                callback(connection);
            }
        }
    }

    class Delegate2App
    {
        public static DBManager.EnumConnectionsCallback myCallback =
            new
DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);
        public    static    void    ActiveConnectionsCallback(DBConnection
connection)
        {
            Console.WriteLine ("Callback method called for " +
                               connection.name);
        }

        public static void Main()
        {
            DBManager dbMgr = new DBManager();
            dbMgr.AddConnections();

            DBManager.EnumConnections(myCallback);
        }
    }
```

注意： 因为代表的标准命名习惯是把以代表为参数的方法加上扩展词 callback，这样就容易在使用代表时，误把方法的名字当作代表的名字。在这种情况下，就会在编译时产生误导性的错误，告诉您刚才指定了一个方法，但是缺少这个方法的类。如果产生这样的错误，要记住实际上的问题是，在程序中指定的是一个方法，而不是代表。

14.3　仅在需要时创建代表

到目前为止，无论用到与否，在我们所见到的两个例子中都创建了代表。在这两个例子中没有什么问题，因为它总是会被调用的。但是，当定义您自己的代表时，很重要的一点就是要考虑何时才创建代表。让我们考虑一下，例如，创建一个特定的代表是需要耗费

时间的，并且这样做并非毫无代价。在有些情况下，我们知道客户端通常并不会调用一个给定的回调方法，这样就可以推迟代表的创建，直到实际需要在属性中包装(wrap)其实例的时候。为了演示怎样做，下面对 DBManager 类进行修改，使用一个只读属性——因为里面只有一个方法 getter——来实例化代表。直到这个属性被引用时才创建该代表。

```
using System;

class DBConnection
{
    public DBConnection(string name)
    {
        this.name = name;
    }

    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
        set

        {
            this.Name = value;
        }
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] = new
                DBConnection("DBConnection " + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection
connection);
```

```
        public   static   void   EnumConnections(EnumConnectionsCallback
callback)
        {
            foreach (DBConnection connection in activeConnections)
            {
                callback(connection);
            }
        }
    }

    class Delegate3App
    {
        public DBManager.EnumConnectionsCallback myCallback
        {
            get
            {
                return new DBManager.EnumConnectionsCallback
                                (ActiveConnectionsCallback);
            }
        }

        public   static   void   ActiveConnectionsCallback(DBConnection
connection)
        {
            Console.WriteLine
                ("Callback method called for " + connection.name);
        }

        public static void Main()
        {
            Delegate3App app = new Delegate3App();

            DBManager dbMgr = new DBManager();
            dbMgr.AddConnections();

            DBManager.EnumConnections(app.myCallback);
        }
    }
```

14.4 代表构成

构成代表的能力——从多个代表创建出单个代表——在一开始的时候并不能让人感到

方便，但是如果您需要的话，那么一定会感到高兴，C#设计组已经考虑到了这一点。让我们来看几个例子，在这些例子中，代表构成是很有用的。在第一个例子中，有一个分布式系统，并且有一个类迭代给定地点的每一个部分，为每一个具有“on-hand”值小于 50 的部分调用一个回调方法。在一个更为现实的分布式例子中，公式中不仅要考虑到“on-hand”值，还要考虑到与订货至交货时间有关的“on order”和“in transit”值，还要减去安全库存级别等等。但是，我们先尽量将该问题简单化：如果某个部分的 on-hand 值小于 50，就会发生一个异常。

问题在于如果某个给定部分库存不够，那么我们要调用两个不同的方法：我们要记录事件，然后还要向采购主管发电子邮件。因此，让我们来看一下怎样程序化地从多个代表创建出单个复合代表：

```
using System;
using System.Threading;

class Part
{
    public Part(string sku)
    {
        this.Sku = sku;

        Random r = new Random(DateTime.Now.Millisecond);
        double d = r.NextDouble() * 100;

        this.OnHand = (int)d;
    }

    protected string Sku;
    public string sku
    {
        get
        {
            return this.Sku;
        }
        set
        {
            this.Sku = value;
        }
    }

    protected int OnHand;
    public int onhand
    {
```

```
        get
        {
            return this.OnHand;
        }
        set
        {
            this.OnHand = value;
        }
    }
}

class InventoryManager
{
    protected const int MIN_ONHAND = 50;

    public Part[] parts;
    public InventoryManager()
    {
        parts = new Part[5];
        for (int i = 0; i < 5; i++)
        {
            Part part = new Part("Part " + (i + 1));

            Thread.Sleep(10); // Randomizer is seeded by time.

            parts[i] = part;
            Console.WriteLine("Adding part '{0}' on-hand = {1}",
                              part.sku, part.onhand);
        }
    }

    public delegate void OutOfStockExceptionMethod(Part part);
    public void ProcessInventory(OutOfStockExceptionMethod exception)
    {
        Console.WriteLine("\nProcessing inventory...");
        foreach (Part part in parts)
        {
            if (part.onhand < MIN_ONHAND)
            {
                Console.WriteLine
                    ("{0} ({1}) is below minimum on-hand {2}",
                    part.sku, part.onhand, MIN_ONHAND);

                exception(part);
```

```
                }
            }
        }
    }

    class CompositeDelegate1App
    {
        public static void LogEvent(Part part)
        {
            Console.WriteLine("\tlogging event...");
        }

        public static void EmailPurchasingMgr(Part part)
        {
            Console.WriteLine("\temailing Purchasing manager...");
        }

        public static void Main()
        {
            InventoryManager mgr = new InventoryManager();

            InventoryManager.OutOfStockExceptionMethod LogEventCallback =
                new InventoryManager.OutOfStockExceptionMethod(LogEvent);

            InventoryManager.OutOfStockExceptionMethod
                EmailPurchasingMgrCallback = new

InventoryManager.OutOfStockExceptionMethod(EmailPurchasingMgr);

            InventoryManager.OutOfStockExceptionMethod
                OnHandExceptionEventsCallback =
                EmailPurchasingMgrCallback + LogEventCallback;

            mgr.ProcessInventory(OnHandExceptionEventsCallback);
        }
    }
```

运行这个应用程序会产生下列结果：

```
Adding part 'Part 1' on-hand = 16
Adding part 'Part 2' on-hand = 98
Adding part 'Part 3' on-hand = 65
Adding part 'Part 4' on-hand = 22
Adding part 'Part 5' on-hand = 70
```

```
Processing inventory...
Part 1 (16) is below minimum on-hand 50
   logging event...
   emailing Purchasing manager...
Part 4 (22) is below minimum on-hand 50
   logging event...
   emailing Purchasing manager...
```

因而，使用语言的这种功能，我们可以动态地分辨出哪些方法中包含了回调方法，把这些方法集中到单个代表中，并且把这个复合代表当作单个代表一样地传递。运行环境将自动确保所有的方法都按顺序调用。此外，可以使用减号操作符把所需的代表从复合代表中移去。

然而，这些方法按照顺序方式被调用的事实回避了一个重要的问题：为什么不能通过让每个方法连续不断地调用下一个方法，来简单地把这些方法链接到一起呢？在这一节的例子中，我们就可以这样做。该例子中只有两个方法，而且这两个方法总是成对调用的。但是我们还是把这个例子稍微复杂化些。假设我们有几个商店位置，每个位置都指示了应该调用的方法。例如，Location1 可能是仓库，因此我们要记录事件，并向采购主管发送电子邮件。而所有其他地方低于最小 on-hand 值的部分会导致事件被记录下来，并且商店的主管会接收到电子邮件。

根据被处理对象的位置，通过动态地创建复合代表，我们可以很轻易地满足这些需求。如果没有这些代表的话，我们就不得不写一个方法，不仅要判断应该调用哪个方法，还要在调用过程中进行追踪，看哪个方法已经被调用过了，哪个方法还未被调用。正如您在下面的代码中所见到的一样，代表把原本可能非常复杂的操作简化了。

```
using System;

class Part
{
   public Part(string sku)

   {
      this.Sku = sku;

      Random r = new Random(DateTime.Now.Millisecond);
      double d = r.NextDouble() * 100;

      this.OnHand = (int)d;
   }

   protected string Sku;
   public string sku
```

```
        {
            get
            {
                return this.Sku;
            }
            set
            {
                this.Sku = value;
            }
        }

        protected int OnHand;
        public int onhand
        {
            get
            {
                return this.OnHand;
            }
            set
            {
                this.OnHand = value;
            }
        }
    }

    class InventoryManager
    {
        protected const int MIN_ONHAND = 50;

        public Part[] parts;
        public InventoryManager()
        {
            parts = new Part[5];
            for (int i = 0; i < 5; i++)
            {
                Part part = new Part("Part " + (i + 1));
                parts[i] = part;
                Console.WriteLine
                    ("Adding part '{0}' on-hand = {1}",
                     part.sku, part.onhand);
            }
        }

        public delegate void OutOfStockExceptionMethod(Part part);
```

```
    public void ProcessInventory(OutOfStockExceptionMethod exception)
    {
        Console.WriteLine("\nProcessing inventory...");
        foreach (Part part in parts)
        {
            if (part.onhand < MIN_ONHAND)
            {
                Console.WriteLine
                    ("{0} ({1}) is below minimum onhand {2}",
                    part.sku, part.onhand, MIN_ONHAND);

                exception(part);
            }
        }
    }
}

class CompositeDelegate2App
{
    public static void LogEvent(Part part)
    {
        Console.WriteLine("\tlogging event...");
    }

    public static void EmailPurchasingMgr(Part part)
    {
        Console.WriteLine("\temailing Purchasing manager...");
    }

    public static void EmailStoreMgr(Part part)
    {
        Console.WriteLine("\temailing store manager...");
    }

    public static void Main()
    {
        InventoryManager mgr = new InventoryManager();
        InventoryManager.OutOfStockExceptionMethod[] exceptionMethods
            = new InventoryManager.OutOfStockExceptionMethod[3];

        exceptionMethods[0] = new
            InventoryManager.OutOfStockExceptionMethod
                (LogEvent);
        exceptionMethods[1] = new
```

```
            InventoryManager.OutOfStockExceptionMethod
                (EmailPurchasingMgr);
        exceptionMethods[2] = new
            InventoryManager.OutOfStockExceptionMethod
                (EmailStoreMgr);

        int location = 1;

        InventoryManager.OutOfStockExceptionMethod compositeDelegate;

        if (location == 2)
        {
            compositeDelegate =
                exceptionMethods[0] + exceptionMethods[1];
        }
        else
        {
            compositeDelegate =
                exceptionMethods[0] + exceptionMethods[2];
        }

        mgr.ProcessInventory(compositeDelegate);
    }
}
```

现在编译并执行这个程序，根据对变量 location 赋值的不同，会得到不同的结果。

14.5 定义具有代表的事件

几乎所有的 Windows 应用程序都需要处理一些异步事件。这类事件中，有些事件是通用的，例如，当用户以某种方式与应用程序进行交互时，Windows 向应用程序消息队列发送消息。还有一些属于更加特殊的问题域，例如为一个更新过的订单打印发票。

C#中的事件遵循“发布——预订”的设计模式，在这种模式中，一个类公布能够出现的事件，然后任意数量的类都可以预订这个事件。一旦事件产生，运行环境就负责通知每个订户，告诉它们事件已经发生了。

方法作为所产生事件的结果，由代表来定义。但是，要时时记住在这种方式下使用代表时的严格规定。首先，代表必须被定义为采用两个参数值。第二，这些参数值总是代表两个对象：引发事件的对象(发布者)和一个事件信息对象。另外，第二个对象必须是从.NET 框架的 EventArgs 类中派生出来的。

现在假设我们要监视库存级的改变。我们可以创建一个名为 InventoryManager 的类，这个类将总是用于更新库存。无论何时，当通过类似于库存接收、销售以及实际上的库存更新等动作对库存进行改变时，InventoryManager 类将发布一个事件。然后，任何需要保持同步更新的类将预订这个事件。下面的例子表明了如何在 C#中使用代表和事件进行编码。

```
using System;

class InventoryChangeEventArgs : EventArgs
{
    public InventoryChangeEventArgs(string sku, int change)
    {
        this.sku = sku;
        this.change = change;
    }

    string sku;
    public string Sku
    {
        get
        {
            return sku;
        }
    }

    int change;
    public int Change
    {
        get
        {
            return change;
        }
    }
}

class InventoryManager // Publisher.
{
    public delegate void InventoryChangeEventHandler
        (object source, InventoryChangeEventArgs e);
    public event InventoryChangeEventHandler OnInventoryChangeHandler;

    public void UpdateInventory(string sku, int change)
    {
```

```
        if (0 == change)
            return; // No update on null change.

        // Code to update database would go here.

        InventoryChangeEventArgs e = new
            InventoryChangeEventArgs(sku, change);

        if (OnInventoryChangeHandler != null)
            OnInventoryChangeHandler(this, e);
    }
}

class InventoryWatcher // Subscriber.
{
    public InventoryWatcher(InventoryManager inventoryManager)
    {
        this.inventoryManager = inventoryManager;
        inventoryManager.OnInventoryChangeHandler += new
InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
    }
    void OnInventoryChange(object source, InventoryChangeEventArgs e)
    {
        int change = e.Change;
        Console.WriteLine("Part '{0}' was {1} by {2} units",
            e.Sku,
            change > 0 ? "increased" : "decreased",
            Math.Abs(e.Change));
    }
    InventoryManager inventoryManager;
}

class Events1App
{
    public static void Main()
    {
        InventoryManager inventoryManager =
            new InventoryManager();

        InventoryWatcher inventoryWatch =
            new InventoryWatcher(inventoryManager);

        inventoryManager.UpdateInventory("111 006 116", -2);
        inventoryManager.UpdateInventory("111 005 383", 5);
```

```
    }
}
```

让我们来看一看 InventoryManager 类的前两个成员：

```
public Delegate void InventoryChangeEventHandler
(object source, InventoryChangeEventArgs e);
public event InventoryChangeEventHandler OnInventoryChangeHandler;
```

代码的第一行是一个代表，现在我们知道这个代表是为一个方法符号定义的。正如前面所提到的，所有在事件中用到的代表都必须被定义为采用两个参数：一个发布者对象(在这种情况下，是事件源)和一个事件信息对象(一个从 EventArgs 类派生的对象)。第二行代码中用到了关键字 event，通过这个成员类型，可以指定代表和一个(或多个)当事件引发时调用的方法。

在类 InventoryManager 中的最后一个方法是方法 UpdateInventory，无论何时库存被改变，这个方法都会被调用。正如您所见到的那样，这个方法创建了一个 InventoryChangeEventArgs 类型的对象。这个对象被传递到所有的订户那里，用于描述所发生的事件。

现在来看一看接下来的两行代码：

```
if (OnInventoryChangeHandler != null)
OnInventoryChangeHandler(this, e);
```

条件语句 if 检查事件是否有什么与方法 OnInventoryChangeHandler 方法相关的订户。如果有的话——换句话说，就是 OnInventoryChangeHandler 不等于 null——实际事件就被引发了。这就是在发布方要做的全部事情。现在让我们来看一看订户代码。

在这个例子中的订户是名为 InventoryWatcher 的类。这个类所要做的就是执行两个简单的任务。第一，实例化一个新的 InventoryManager.InventoryChangeEventHandler 并把这个代表加入到事件 InventoryManager.OnInventoryChangeHandler 上去，通过这种作法，可以把自己加为一个订户。要特别注意所用的语法——它用到了复合赋值操作符＋＝，把自己加入到订户列表中，这样做不会取消以前的订户。

```
inventoryManager.OnInventoryChangeHandler
+=newInventoryManager.InventoryChangeEventHandler(OnInventoryChange);
```

这里惟一需要提供的参数值就是当事件引发时被调用的方法名字。

订户需要完成的另外一个任务就是实现其事件处理器。在这个例子中，事件处理器是 InventoryWatcher.OnInventoryChange，这个事件处理器打印一条消息，其中有该部分的号码和库存的变化。

最后，运行这个程序的代码把类 InventoryManager 和 InventoryWatcher 实例化，并在每次调用 InventoryManager.UpdateInventory 方法时，会自动引发一个事件，该事件导致了

方法 InventoryWatcher.OnInventoryChanged 被调用。

14.6 本章小结

代表是一种具有类型保护和安全机制的对象，与 C++中函数指针所要达到的目的相同。代表不同于类和接口，因为它不是在编译时被定义的，它指的是单个的方法，且在运行时被定义。代表通常情况下被用于执行异步处理，并把自定义的代码加入到类的代码路径中。代表有许多常规用途，包括作为回调方法、定义静态方法，以及定义事件等。

第Ⅳ部分　高级 C#

第 15 章　多线程编程

从技术角度讲，线程不是仅对 C#语言而言的；由于这个因素，多数关于 C#语言的书习惯于避开这部分内容。虽然本书只是在专门讲解 C#语言，但是，多线程的一般概念也是多数程序员在学习这门新语言时必须熟悉的。当然，不可能在一章中涵盖与线程相关的全部内容，但还是会介绍有关线程终止(aborting)、调度(scheduling)和生存周期管理的基本内容，甚至还有某些中高级的内容。此外，还将通过 System.Monitor 和 System.Mutex 类以及 C#中的 lock 语句来讨论线程同步。

15.1　线程的基本知识

使用单线程应用程序就像在只有一个收款员的超市中购物。只雇一个收款员对店主来说会比较省钱，客流量较低时，这个收款员已经足以应付。但如果遇到有优惠打折的日子，等候付款的队伍就会变得越来越长，某些顾客将变得十分不高兴。这时所发生的正是典型的瓶颈现象：大量的数据和过于狭窄的信息通道。而解决方案是明确的——雇佣更多的收款员。

对于线程及应用程序也同样。多线程使得一个应用程序将任务划分为几部分，它们各自独立地执行，使得最大限度地利用处理器和用户时间成为可能。但是，如果您是刚开始接触多线程编程，那么这里就有必要提醒您一下：并不是说线程对所有应用程序而言都是最正确的选择，它有时甚至使应用程序运行变慢。因此，最重要的是，在学习多线程运行的语法的同时，也要懂得何时才用得到它。在本章的末尾又加了一节(“线程运行指南”)，内容是关于何时在您的应用程序中创建多个线程。现在，让我们开始学习在 Microsoft .NET 框架中线程是如何实现的。

15.1.1 线程与多任务

一个线程就是一个处理单元，而多任务是指多个线程的并发执行。多任务形成两种类型：合作型(cooperative)和占先型(preemptive)。许久以前，Microsoft Windows 的早期版本支持合作型的多任务运行，这意味着各个线程应负责放弃对处理器的占用，以便处理器可以运行其他线程。对具有其他操作系统背景知识的人，对 IBM System/38 (CPF)和 OS/2 系统——来说，每一个人都有这样的经历，某天由于没有在代码中加入对 *PeekMessage* 的调用以便于处理器为其他线程服务，结果不得不暂停工作。我们最典型的反应就是“我该怎么办？”

但是，Microsoft Windows NT——以及随后的 Windows 95、Windows 98 和 Windows 2000——开始与 OS/2 一样，支持占先型多任务。在占先型多任务执行中，处理器负责给每个线程分配一定量的运行时间——一个时间片(timeslice)。处理器接着在不同的线程之间进行切换，给每一个线程分配时间片，而程序员不用再去考虑该在何时、怎样释放对处理器的控制，以使其他线程得以运行。因为.NET 仅能运行在占先型多任务操作系统上，这就是我们所要特别关注的地方。

附带说一句，既使是占先型多任务运行系统，假若是运行在单处理器的计算机上，您还是不能真正实现多个线程的同时运行。这是因为处理器只是每间隔几毫秒就在不同处理进程中进行切换，这样的同时运行只是表面上“感觉”如此。如果您想现在真正地并发运行多线程，那您必须在具有多个处理器的机器上开发和运行您的代码。

15.1.2 上下文切换

上下文切换(context switching)是线程运行的一部分，它对某些人来说是一个难以理解的概念。因此，有必要在这里对“线程运行基础”一节给出一个简要总结。

处理器使用一个硬件时钟来判断一个指定线程的时间片何时结束。当这个硬件计时器给出中断信号时，处理器把当前运行的线程所用的所有寄存器(registers)数据存储到堆栈中。然后，处理器把堆栈里那些相同的寄存器信息存放到一种被称为“上下文结构”的数据结构中。当处理器要切换回原来执行的线程时，它反向执行这个过程，利用与该线程相关的上下文结构，在寄存器里重新恢复与这一线程相关的信息。这样的一个完整过程称为“上下文切换”。

15.2 C#语言中的多线程应用程序

在考虑 C#语言中线程的不同使用方式之前，让我们来看一下在 C# 语言中创建第二个

线程是多么容易。在讨论这个例子后，我们将更仔细观察 System.Threading 名称空间，特别是 Thread 类。在这个例子中，将在 Main 方法中创建第二个线程。这个与第二个线程相关的方法接着会输出一个字符串，显示线程被调用了。

```
using System;
using System.Threading;

class SimpleThreadApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine("Worker thread started");
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);

        Console.WriteLine("Main - Creating worker thread");

        Thread t = new Thread(worker);
        t.Start();

    Console.WriteLine
        ("Main - Have requested the start of worker thread");
    }
}
```

如果编译并且执行这个应用程序，那么您将会看到 Main 方法打印的信息在 worker 线程信息之前，这就证明了 worker 线程实际上是以异步方式工作的。让我们来分析一下，程序中究竟发生了什么。

第一个需注意的地方是对于 System.Thread 名称空间使用 using 语句。不久我们就会明白名称空间概念。现在，只要理解这个名称空间中包含在.NET 环境下线程运行所必需的许多不同类就可以了。现在来看一下主方法的第一行：

```
ThreadStart WorkerThreadMethod =new ThreadStart(WorkerThreadMethod);
```

无论何时，当您看到下面这种形式，您就可以确定 *x* 是一个代表——或者就像在第 14 章中所学的那样——一种对方法信号的定义：

x varName = new *x(methodName)*;

这样,我们知道 *ThreadStart* 是一个代表。但是它又不仅仅是一个代表。特别地说，ThreadStart 是创建新线程时必须用到的代表，并且要用它来指定一个方法，这一方法是做

为您的线程方法来调用的。在那里实例化了一个 Thread 对象，其中的构造函数把一个 ThreadStart 代表作为其惟一参数，如：

```
Thread t =new Thread(worker);
```

在此之后调用了 Thread 对象的 Start 方法，这产生了一个对 WorkerThreadMethod 的调用。

使用三行代码建立和运行线程及线程方法本身，这样您就完成并运行了一个线程。现在，让我们更进一步，对 System.Threading 名称空间及使这一切发生的类做进一步的了解。

15.3 线程处理

线程的所有创建与管理都是通过使用 System.Threading.Thread 类来实现的。因此，我们将从这里开始。

15.3.1 AppDomain

在.NET 中，线程是在 *AppDomain* 中运行的。您也许曾经听说过，一个 *AppDomain* 类似于一个 Win32 进程，因为它拥有许多与 Win32 进程同样的优点，包括容错性以及独立开始和结束的能力。这是一个很好的比较，但这样的比较破坏了线程之间的关系。在 Win32 中，一个线程仅局限在单个的进程内，就像早先介绍上下文转换时您所看到的那样。一个进程中的线程不能调用属于另一个进程中的线程的方法。然而，在.NET 中，线程可以超越 *AppDomain* 的边界，并且，一个线程中的方法可以调用另一个 *AppDomain* 中的方法。因此，对 *AppDomain* 有一个更好的定义：在一个物理进程中的逻辑进程。

15.3.2 Thread 类

若使用线程，就一定要用到 Thread 类。这一节介绍使用 Thread 类来完成基本线程任务。

15.3.2.1 创建线程和线程对象

可以通过两种方式来实例化一个 Thread 对象。我们已经看到了其中一种方式：创建一个新的线程，并且在那个进程中，得到了一个 Thread 对象，利用这个对象对新的线程进行处理。另一种获取 Thread 对象的方式是，为正在执行的线程调用静态的 Thread. Current Thread 方法。

15.3.2.2 管理线程的生存周期

要想对一个线程的活动或生存周期进行管理，需要执行许多不同的任务。可以通过使用不同的 Thread 方法来管理所有这些任务。例如，需要使一个线程暂停一段时间是很常见的事。为达到这一目的，可以调用 Thread.Sleep 方法。这个方法只需一个参数值，这个参数值以毫秒的形式表示时间量，这个时间量就是您所希望线程停顿的时间。注意：Thread.Sleep 方法是一个静态方法，调用时不能带有一个 Thread 对象的实例。至于为什么会这样，有一个很好的理由，除了正在执行的线程外，不允许对其他任何线程调用 Thread.Sleep 方法。静态方法 Thread.Sleep 调用静态方法 CurrentThread，这样就使线程按照预设的一段时间暂停。下面有一个例子：

```
using System;
using System.Threading;

class ThreadSleepApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine("Worker thread started");

        int sleepTime = 5000;

        Console.WriteLine("\tsleeping for {0} seconds",sleepTime/1000);
        Thread.Sleep(sleepTime); // Sleep for five seconds.
        Console.WriteLine("\twaking up");
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);

        Console.WriteLine("Main - Creating worker thread");

        Thread t = new Thread(worker);
        t.Start();

        Console.WriteLine
            ("Main - Have requested the start of worker thread");
    }
}
```

还有两种方式可以调用方法 Thread.Sleep。第一种，调用 Thread.Sleep 方法时带有值 0，

这将使正在运行的线程放弃它尚未使用的时间片。在线程中传递 Timeout.Infinite 的值会导致线程的暂停(挂起)时间不定，直到另一个线程调用这个被挂起线程的 Thread.Interrupt 方法时，挂起(suspension)才会被中断。

第二种使线程执行挂起的方式是使用 Thread.Suspend 方法。在这两种技术间有某些重要区别。首先, Thread.Suspend 方法可以被当前正在执行的线程或其他线程调用。第二，一旦一个线程以这种方式挂起时，只有另外的线程才能令它恢复，这要通过 Thread.Resume 方法来实现。注意，当一个线程挂起另一个线程，第一个线程并未阻塞，调用立即返回。同样，不论 Thread.Suspend 方法被一个给定的线程调用了多少次，只要对 Thread.Resume 方法进行一次调用就可使线程恢复执行。

15.3.2.3 销毁线程

如果需要销毁(destroy)一个线程，您可以通过调用 Thread.Abort 方法完成这一操作。通过 ThreadAbortException，运行环境会强行终止一个线程。即便该方法试图截获 ThreadAbortException，运行环境也不会允许它这么做。但是，运行环境将执行这个被终止的线程中 finally 块的代码，如果这个块存在的话。随后的代码体现了这样的思想。Main 方法暂停 2 秒，来确认运行环境已经有足够时间使 worker 线程开始。一旦此线程开始后，worker 线程开始计数到 10，在每个数之间停顿 1 秒。当 Main 方法在 2 秒暂停后恢复执行时，它终止了 worker 线程。注意，在终止之后，finally 块中的代码被执行了。

```
using System;
using System.Threading;

class ThreadAbortApp
{
    public static void WorkerThreadMethod()
    {
        try
        {
            Console.WriteLine("Worker thread started");

            Console.WriteLine
                ("Worker thread - counting slowly to 10");
            for (int i = 0; i < 10; i++)
            {
                Thread.Sleep(500);
                Console.Write("{0}...", i);
            }

            Console.WriteLine("Worker thread finished");
        }
```

```
            catch(ThreadAbortException e)
            {
            }
            finally
            {
                Console.WriteLine
                    ("Worker thread -
                     I can't catch the exception, but I can cleanup");
            }
        }

        public static void Main()
        {
            ThreadStart worker = new ThreadStart(WorkerThreadMethod);

            Console.WriteLine("Main - Creating worker thread");

            Thread t = new Thread(worker);
            t.Start();

            // Give the worker thread time to start.
            Console.WriteLine("Main - Sleeping for 2 seconds");
            Thread.Sleep(2000);

            Console.WriteLine("\nMain - Aborting worker thread");
            t.Abort();
        }
    }
```

编译并执行这个应用程序，输出结果如下：

```
Main -Creating worker thread
Main -Sleeping for 2 seconds
Worker thread started
Worker thread -counting slowly to 10
0...1...2...3...
Main -Aborting worker thread
Worker thread -I can 't catch the exception,but I can cleanup
```

我们应该认识到，当 Thread.Abort 方法被调用时，线程不会立即终止执行。运行环境将会等待，直到线程到达文档中所描述的“安全点”。 因此，如果代码依赖于某些终止后发生的事件，并且还要确保线程已经完全停止，那么可以使用 Thread.Join 方法。这是一个同步调用，同步调用意味着直到线程完全停止，调用才会返回。最后，还要注意，一旦终止了一个线程，这个线程就不能够再重新开始了。在这个例子里，尽管已经有一个有效的

Thread 对象，我们也无法对其执行任何代码。

15.3.3 线程调度

一旦分配给某个线程的时间片用完了，处理器就要在线程之间切换，在切换过程中，处理器不是任意选择下一个将执行的线程的。每个线程都有一个与之相联系的优先级，这个优先级表示该线程与系统中其他线程的关系，以通知处理器它应该排在何处。线程的优先级默认为 normal——简单地说，优先级常是如此——该优先级在运行环境中产生。对那些在运行环境之外创建的线程来说，它们保持各自原来的优先级。可以使用 Thread.Priority 属性去查看和设置这个值。Thread.Priority 属性的设定要用到 Thread.ThreadPriorityI 类型的值，这是一个枚举类型，定义为以下几个值：最高(Highest)，高于正常(AboveNormal)，正常(Normal)，低于正常(BelowNormal)和最低(Lowest)。

即使是最简单的代码也会受到优先级的影响，为了证明这一点，我们用到了下面的例子，在这个例子中，一个 worker 线程从 1 计数到 10，另一个 worker 线程从 11 计数到 20。注意每个 WorkerThread 方法中的嵌套循环(nested loop)。这里的每个循环都表示线程在实际的应用程序所要做的工作。因为这些方法实际上并不真正地起任何作用，所以如果没有这些循环就会导致每个线程在其第一个时间片中就会结束其工作！

```
using System;
using System.Threading;

class ThreadSchedule1App
{
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread started");

        Console.WriteLine
           ("Worker thread - counting slowly from 1 to 10");
        for (int i = 1; i < 11; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }
            Console.Write("{0}", i);
        }
```

```
        Console.WriteLine("Worker thread finished");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread started");

        Console.WriteLine
          ("Worker thread - counting slowly from 11 to 20");
        for (int i = 11; i < 20; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }
            Console.Write("{0}", i);
        }

        Console.WriteLine("Worker thread finished");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

运行这个应用程序会产生下面的输出。注意，为了看上去显得简明一点，输出内容被截去了一部分——大部分点号都被删去了。

```
Main -Creating worker threads
Worker thread started
Worker thread started
```

```
Worker thread -counting slowly from 1 to 10
Worker thread -counting slowly from 11 to 20
......1......11......2......12......3......13
```

正如您看到的那样，两个线程获得了等量的处理器运行时间。现在用下面的代码来修改每个线程的优先属性——第一个线程的优先级设为所允许的最高级，将第二个线程的优先级设为最低——接下来看到的结果将会大不相同。

```
using System;
using System.Threading;

class ThreadSchedule2App
{
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread started");

        Console.WriteLine
            ("Worker thread - counting slowly from 1 to 10");
        for (int i = 1; i < 11; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                // Code to imitate work being done.
                int a;
                a = 15;
            }

            Console.Write("{0}", i);
        }

        Console.WriteLine("Worker thread finished");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread started");

        Console.WriteLine
            ("Worker thread - counting slowly from 11 to 20");
        for (int i = 11; i < 20; i++)
        {
            for (int j = 0; j < 100; j++)
```

```
                {
                    Console.Write(".");
                    // Code to imitate work being done.
                    int a;
                    a = 15;
                }
                Console.Write("{0}", i);
            }

            Console.WriteLine("Worker thread finished");
        }

        public static void Main()
        {
            ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
            ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

            Console.WriteLine("Main - Creating worker threads");

            Thread t1 = new Thread(worker1);
            Thread t2 = new Thread(worker2);

            t1.Priority = ThreadPriority.Highest;
            t2.Priority = ThreadPriority.Lowest;

            t1.Start();
            t2.Start();
        }
    }
```

(经过简化的)结果显示在下一页。注意，由于第二个线程被设为最低的优先性，它没有接受任何循环，直至第一个线程已经完成了工作。

```
Main -Creating worker threads
Worker thread started
Worker thread started
Worker thread -counting slowly from 1 to 10
Worker thread -counting slowly from 11 to 20
......1......2......3......4......5......6......7......8......9.....
.10......
11......12......13......14......15......16......17......18......19..
....20
```

请记住，如果当您想把线程所具有的优先性通知给处理器时，最终是操作系统来使用这个值，把这个值作为调度算法的一部分转发给处理器。在.NET 环境中，这个算法以刚刚

使用过的优先级 (用 Thread.Priority 属性)以及处理器的 priority class 和 dynamic boost 值为依据。所有的这些值都用来创建数量值(在一个 Intel 处理器上是 0–31)，这些值代表线程的优先性。具有最大数值的线程就具有最高的优先性。

在关于线程调度的问题中，最后一个要注意的地方是：使用它时要小心。假设我们有一个 GUI 应用程序，以及几个 worker 线程来完成某种异步性的工作。如果把 worker 线程的优先性设定得太高，UI 就有可能变得非常迟缓，因为 GUI 应用程序是运行在主线程中的，而主线程获得的 CPU 循环较少。除非有特殊原因，才把一个线程在调度时赋予其高优先性，否则通常最好使线程的优先性默认为 Normal。

注意： 有这样一种情况，几个线程在同样的优先性下运行时，它们将获得等量的处理器时间。这就叫做“循环调度”(round robin scheduling)。

15.4 线程安全和同步

当为单线程环境编程时，以这种方式来书写方法是很常见的，即在代码中的几个点处，对象处于暂时的无效状态。很明显，如果一次仅有一个线程在访问对象，您就要保证在另一方法被调用以前，每个方法都要完成——这就意味着对于对象的任何客户端而言，目标总是处于有效的状态。但是，当多个线程混合在一起交叉调用时，很容易会发生这样的情形：当对象处于无效状态时，处理器切换到另一个线程。如果那个线程也试图使用同样的对象，其结果是非常难以预测的。 因此，术语“线程安全”的意思是指，当多个线程并发使用某个对象时，该对象的成员总是保持有效状态。

因此，怎样防止产生这种不可预测的状态呢？实际上，由于这种情况在程序运行时发生得很普遍，这里有几种办法来解决这个众所周知的问题。在这一节中，将会介绍最为普通的办法：同步。通过同步，可以指定代码的临界区(critical section)，一次只有一个线程可以进入临界区。这就确保了任何对象暂时的无效状态都不会被对象的客户端所“见到”。我们将看到几种定义临界区的方法，包括.NET 中的 Monitor 和 Mutex 类，以及 C#中的 lock 语句。

15.4.1 使用 Monitor 类保护代码

类 System.Monitor 通过锁定与信号量的方式，使得对于代码块可以进行串行化的访问。例如，现在有一个方法，可以用来更新数据库，并且不能被两个或更多的线程同时执行。如果使用这个方法特别消耗时间，而现在又有多个线程，其中的任何一个线程都可能调用这个方法，那么很快就将发生严重的问题。这就是为什么会出现类 Monitor 的原因。看一

看下面这个同步的例子。这里我们有两个线程，这两个线程都会调用 Database.SaveData 方法。

```
using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        Console.WriteLine("Database.SaveData - Started");

        Console.WriteLine("Database.SaveData - Working");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine("\nDatabase.SaveData - Ended");
    }
}

class ThreadMonitorlApp

{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");

        Console.WriteLine
            ("Worker thread #1 -Calling Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Worker thread #1 - Returned from Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread #2 - Started");

        Console.WriteLine
            ("Worker thread #2 - Calling Database.SaveData");
```

```
            db.SaveData("o");

            Console.WriteLine("Worker thread #2 - Returned from Output");
        }

        public static void Main()
        {
            ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
            ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

            Console.WriteLine("Main - Creating worker threads");

            Thread t1 = new Thread(worker1);
            Thread t2 = new Thread(worker2);

            t1.Start();
            t2.Start();
        }
    }
```

编译并执行这个应用程序，将会看到输出的结果是 o 和 x 的混合，这显示了 Database.SaveData 被两个线程并发运行(再次注意，此处简化了输出结果)。

```
Main - Creating worker threads

Worker thread #1 - Started
Worker thread #2 - Started

Worker thread #1 - Calling Database.SaveData
Worker thread #2 - Calling Database.SaveData

Database.SaveData - Started
Database.SaveData - Started

Database.SaveData - Working
Database.SaveData - Working
xoxoxox
```

为了使这个例子中 Monitor 类更加具体化，我们使用了这个类的两个静态方法。第一个方法被称做 Enter。当执行 Enter 方法时，这个方法会试图获取对象上的 Monitor 锁。如果另一个线程已经拥有了这个锁，这个方法将会阻塞(block)，直至这个锁被释放。注意，这里并不执行隐式的框操作，因此您可以只为这个方法提供一个引用类型。方法 Monitor.Exit 接着被调用，以释放锁。这里是一个经过改写的例子，强制对 Database.SaveData 方法进行串行化访问：

```
using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        Monitor.Enter(this);

        Console.WriteLine("Database.SaveData - Started");

        Console.WriteLine("Database.SaveData - Working");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine("\nDatabase.SaveData - Ended");

        Monitor.Exit(this);
    }
}

class ThreadMonitor2App
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");

        Console.WriteLine
           ("Worker thread #1 - Calling Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Worker thread #1 - Returned from Output");
```

```
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Worker thread #2 - Started");

        Console.WriteLine
            ("Worker thread #2 - Calling Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Worker thread #2 - Returned from Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Main - Creating worker threads");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}
```

注意下面的输出，尽管第二个线程也调用了 Database.SaveData 方法，但是 Monitor.Enter 方法使其阻塞，直至第一个线程释放了它的锁：

```
Main - Creating worker threads

Worker thread #1 - Started
Worker thread #2 - Started

Worker thread #1 - Calling Database.SaveData
Worker thread #2 - Calling Database.SaveData

Database.SaveData - Started
Database.SaveData - Working
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Database.SaveData - Ended

Database.SaveData - Started
```

```
Worker thread #1 - Returned from Output

Database.SaveData - Working
oooooooooooooooooooooooooooooooooooooooo
Database.SaveData - Ended

Worker thread #2 - Returned from Output
```

15.4.2　用 C#中的 lock 语句使用 Monitor 锁

虽然 C# 的 lock 语句并不支持 Monitor 类中所有的特性，但还是可以让您获得和释放一个 Monitor 锁。要想使用 lock 语句，只需简单地用 lock 语句把需要串行化的代码括起来。括号标明了受保护代码的起始点和终止点，因此这里不需要 unlock 语句。下面的代码将产生与前一个所举例子相同的同步输出：

```
using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        lock(this)
        {
            Console.WriteLine("Database.SaveData - Started");

            Console.WriteLine("Database.SaveData - Working");
            for (int i = 0; i < 100; i++)
            {
                Console.Write(text);
            }

            Console.WriteLine("\nDatabase.SaveData - Ended");
        }
    }
}

class ThreadLockApp
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
```

```
            Console.WriteLine("Worker thread #1 - Started");

            Console.WriteLine
                ("Worker thread #1 - Calling Database.SaveData");
            db.SaveData("x");

            Console.WriteLine("Worker thread #1 - Returned from Output");
        }

        public static void WorkerThreadMethod2()
        {
            Console.WriteLine("Worker thread #2 - Started");

            Console.WriteLine
                ("Worker thread #2 - Calling Database.SaveData");
            db.SaveData("o");

            Console.WriteLine("Worker thread #2 - Returned from Output");
        }

        public static void Main()
        {
            ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
            ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

            Console.WriteLine("Main - Creating worker threads");

            Thread t1 = new Thread(worker1);
            Thread t2 = new Thread(worker2);

            t1.Start();
            t2.Start();
        }
    }
```

15.4.3 通过使用 Mutex 类同步化代码

Mutex 类——在 System.Threading 名称空间中定义——是 Win32 系统原先同样名字在运行时刻的表示。您可以使用互斥(Mutex)对代码进行串行化访问，就如使用 Monitor 锁一样。但是使用 Mutex 类要慢得多，因为它们增加了灵活性。术语“互斥(mutex)”来自于“相互排斥(mutually exclusive)”这两个单词，正如一次只能有一个线程可以获得给定对象的 Monitor 锁一样，一次只有一个线程能获得一个给定的互斥。

可以用下面三种构造函数在 C#语言中创建一个互斥(mutex)：

Mutex( )

Mutex(bool *initiallyOwned*)

Mutex(bool *initiallyOwned*, string *mutexName*)

第一种构造函数创建了一个不带名字的互斥，并且使当前线程成为互斥的拥有者。所以，互斥被当前线程锁定。第二种构造函数只采用了一个布尔型标记值，这个值用以指明创建互斥的线程是否想要拥有它(锁定它)。第三种构造函数允许您指定当前线程是否拥有互斥，并且可以指定互斥的名字。让我们现在用一个互斥来具体地实现对 Database.SaveData 方法的串行化访问：

```
using System;
using System.Threading;

class Database
{
    Mutex mutex = new Mutex(false);

    public void SaveData(string text)
    {
        mutex.WaitOne();

        Console.WriteLine("Database.SaveData - Started");

        Console.WriteLine("Database.SaveData - Working");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine("\nDatabase.SaveData - Ended");

        mutex.Close();
    }
}

class ThreadMutexApp
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Worker thread #1 - Started");
```

```
            Console.WriteLine
                ("Worker thread #1 - Calling Database.SaveData");
            db.SaveData("x");

            Console.WriteLine("Worker thread #1 - Returned from Output");
        }

        public static void WorkerThreadMethod2()
        {
            Console.WriteLine("Worker thread #2 - Started");

            Console.WriteLine
                ("Worker thread #2 - Calling Database.SaveData");
            db.SaveData("o");

            Console.WriteLine("Worker thread #2 - Returned from Output");
        }

        public static void Main()
        {
            ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
            ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

            Console.WriteLine("Main - Creating worker threads");

            Thread t1 = new Thread(worker1);
            Thread t2 = new Thread(worker2);

            t1.Start();
            t2.Start();
        }
    }
```

现在，类 Database 定义了一个 mutex 字段。我们现在并不想让线程拥有互斥，因为我们无法进入 SaveData 方法。SaveData 方法的第一行向我们显示了需要如何做才能获得互斥——使用 Mutex.WaitOne 方法。在这个方法代码的末尾是一个对 Close 方法的调用，这条调用语句释放了互斥。

WaitOne 方法也是被重载的，这是为了提供更大的灵活性，允许您定义线程等候互斥变为可用的时间。下面就是这些重载：

WaitOne( )

WaitOne(TimeSpan time, bool *exitContext*)

WaitOne(int *milliseconds*, bool *exitContext*)

这些重载之间基本的不同之处在于，第一版——例子中所用的——等待时间不确定，而第二和第三版则指定了等待的时间，要么是以一个 TimeSpan 值表示，要么就是以一个 int 值来表示。

15.4.4　线程安全和.NET 类

在新闻组和邮件列表中经常可以看到这样一个问题，即是否所有的.NET System 类都是线程安全的。答案是“不是，而且也应该不是。”如果所有的类都串行化访问它们的功能，这样会严重降低系统的性能。例如，想象一下试着使用一个 collection 类，每次调用这个类的 Add 方法时，这个类就会获得一个 monitor 锁。现在假设要实例化一个 collection 对象，并把大约 1 000 个对象加到这个对象中。性能会变得很差——直到使整个系统变得不可用。

15.5　使用线程的原则

什么时候需要用到线程，以及什么时候用线程最好，避免这些问题是否曾经让您感到头疼？在这一节里将介绍一些常见的场景，在这些场景中线程会对您的应用程序非常有价值。此外，还将介绍几种情况，在这几种情况下，最好还是避免使用多线程。

15.5.1　何时使用线程

如下一节所介绍的那样，当您想增加并发度、简化设计以及更有效地利用 CPU 时间时，应该使用线程。

15.5.1.1　增加并发度

应用程序经常会一次完成多个任务。例如，作者曾经为银行写过一个文档检索系统，这个系统从存贮在自动光盘库的光盘上访问数据。请想象一下我们这里所讨论的海量数据吧；一个自动光盘库带有一个驱动器和 50 个托碟，提供了以 G 字节来计算的数据。有时需要长达 5 至 10 秒才能载入一张盘并找到所请求的文档。显然，如果应用程序在做这一切工作时阻塞了用户输入，那么这将会降低生产率。因此，为了处理用户输入的情况，可用另外一个线程来完成检索数据的物理工作，同时允许用户继续工作。当文档被载入后，这个线程会通知主线程。这是一个很好的例子，具有独立的活动——载入一个文档并处理用户界面(UI)——可以有两个单独的线程来处理这些活动。

15.5.1.2 简化设计

简化复杂系统设计的一种常用方法就是，使用队列和同步处理。使用这样一种设计，您可以设定一个队列来处理系统中发生的不同事件。我们可以不直接调用方法，而是创建对象，并把对象放置在队列中，我们在队列中处理这些对象。在这些队列的另外一端是带有多个线程的服务器程序，这些线程被设置为“监听”来自这些队列的消息。这种简化设计的方法其优点在于提供了可靠、健壮并可扩展的系统。

15.5.1.3 更有效地利用CPU时间

当应用程序在使用其时间片时，在很多时候其实并没有实际起作用。在前面的文档检索例子中，有一个线程在等待自动光盘库载入碟片。很明显，这种等待是与硬件相关的，并不需要用到CPU。还有一个等待时间的例子，即打印一份文档或等待硬盘或CD-ROM驱动器的操作。在每一种情况下，CPU都没有得到利用。这样的情况下，就适合移至后台线程。

15.5.2 何时不应该使用线程

对于那些使用线程的新手来说，常犯的一个错误就是试图在每个应用程序中都使用线程。这样做甚至比根本没有线程还要糟！正如使用“编程武器库”中的其他工具一样，您应该只在合适的时候使用线程。至少在以下几种情况中(在接下来一节中会进一步加以介绍)，在应用程序中应该避免使用线程：当代价超过收益时、当您没有对这两种情况加以比较时，以及当没有理由使用线程时。

15.5.2.1 代价超过利益

正如我们在“线程安全和同步”中所了解到的一样，写多线程程序需要在代码设计时多花一点时间和努力。有时候会有这样的情况，进行多线程编程的效果与使代码变成线程安全所付出的时间相比，可以说是得不偿失。

15.5.2.2 没有对两种情况加以比较

如果您是刚刚学会多线程编程，您可能会很惊讶地发现，由于CPU线程创建以及调度以获得合适的CPU利用率而导致的开销，实际上导致了单线程应用程序的运行更快！这完全取决于您的作法以及您是否真正地把独立的任务分成几个线程。例如，如果必须从硬盘上读取三个文件，产生三个线程的做法对您而言并没有什么好处，因为每个线程都必须使用同一个硬盘。因此，必须确保对系统的单线程和多线程版本原型加以比较，否则就会导致花费额外的时间和设计成本，而实际上反而降低了系统的性能。

15.5.2.3　没有使用线程的理由

并不应该把使用多个线程作为一种默认做法。因为编写多线程应用程序所固有的复杂性，您应该总是默认地编写为使用单线程的代码，除非有充足的理由，才能采用其他做法。

15.6　本章小结

多线程允许应用程序把任务分割为多个，这样它们彼此之间可以独立地工作，最大限度地利用了处理器时间。然而，把多线程加到应用程序中并不是在所有情况下都是正确的选择，并且在某些时候会延缓应用程序的运行。C#中的线程管理和创建是通过System.Threading.Thread 类来进行的。有一个很重要的概念与线程的创建和使用有关，就是线程安全性。线程安全性意味着当对象的成员被多个线程并发使用时，总是维护一个有效的状态。有一点很重要，在学习多线程语法的同时，也要明白应该在何时使用多线程：为了增加并发度时、为了简化设计时，以及为了更好地利用 CPU 时间时。

第 16 章　利用反射查询元数据

在第 2 章中曾涉及到这样的事实，即编译器产生了一个 Win32 可移植、可执行(PE)文件，文件中主要包含 MSIL 和元数据(metadata)。.NET 中一个十分强大的特性就是，它允许您编写代码通过一个称为反射的进程来访问一个应用程序的元数据。简而言之，反射就是能够在运行时刻查找到类型信息。这一章将介绍反射 API，以及如何使用它来迭代一个组合体(assembly)的模块和类型，并且来检索类型设计时刻的特征。您也将看到反射的几个高级用法，例如动态使用方法以及使用类型信息(通过后期绑定)，甚至还可以在运行时刻创建并执行 MSIL 代码！

16.1　反射 API 层次

.NET 反射 API 实际上是一种类的网络结构——图 16.1 中显示了这种结构的一部分——这是在名称空间 System.Reflection 定义的。这些类使您能够在逻辑上遍历组合体和类型信息。您可以在这个层次结构的任何一个位置开始，这取决于应用程序的具体设计需要。

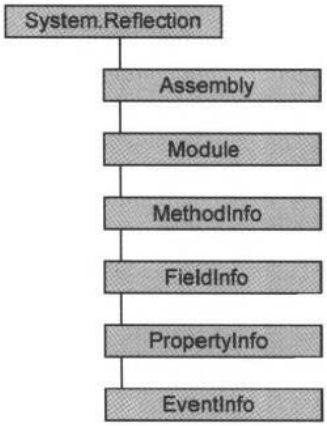


图 16.1　部分.NET System.Reflection 类层次

注意这些类中包含了大量的功能。这里并没有列出每个类的每个方法和域，而是提供一个对某些关键类的概览，并且会有一个演示，显示了在您的应用程序中最需要加入的一些功能。

16.2　Type 类

反射的中心是类 System.Type。类 System.Type 是一个抽象类，代表公用类型系统(Common Type System，CTS)中的一种类型。这个类使您能够查询类型名、类型中包含的模块和名称空间、以及该类型是一个数值类型还是一个引用类型。

16.2.1　检索一个实例的类型

下面的例子显示了如何为一个实例化的 int 检索其 Type 对象：

```
using System;
using System.Reflection;

class TypeObjectFromInstanceApp
{
    public static void Main(string[] args)
    {
        int i = 6;
        Type t = i.GetType();
        Console.WriteLine(t.Name);
    }
}
```

16.2.2　根据名字检索类型

除了可以检索一个变量的 Type 对象，您还可以从一个类型名字创建一个 Type 对象。换句话说，您并不需要有一个该类型的实例。这里就有这样的一个例子，是关于类型 System.Int32 的：

```
using System;
using System.Reflection;

class TypeObjectFromNameApp
{
    public static void Main(string[] args)
    {
        Type t = Type.GetType("System.Int32");
        Console.WriteLine(t.Name);
```

```
        }
    }
```

注意，当调用方法 Type.GetType 时，您无法使用 C#别名，因为所有的语言都使用这个方法。因此，您也就无法使用 C#特定的 int 来代替 System.Int32。

16.2.3 查询类型

类 System.Type 使您能够查询几乎所有与类型相关的属性，包括类型访问限定符、类型是否嵌套、类型的 COM 属性等等。下面的代码为了显示这一点，使用了一些演示类型，还有几个公用类型：

```
using System;
using System.Reflection;

interface DemoInterface
{
}

class DemoAttr : System.Attribute
{
}

enum DemoEnum
{
}

public class DemoBaseClass
{
}

public class DemoDerivedClass : DemoBaseClass
{
}

class DemoStruct
{
}

class QueryTypesApp
{
    public static void QueryType(string typeName)
    {
```

```
try
{
    Type type = Type.GetType(typeName);
    Console.WriteLine("Type name: {0}", type.FullName);
    Console.WriteLine("\tHasElementType = {0}",

                    type.HasElementType);
    Console.WriteLine("\tIsAbstract = {0}", type.IsAbstract);
    Console.WriteLine("\tIsAnsiClass = {0}", type.IsAnsiClass);
    Console.WriteLine("\tIsArray = {0}", type.IsArray);
    Console.WriteLine("\tIsAutoClass = {0}", type.IsAutoClass);
    Console.WriteLine("\tIsAutoLayout = {0}",
                    type.IsAutoLayout);
    Console.WriteLine("\tIsByRef = {0}", type.IsByRef);
    Console.WriteLine("\tIsClass = {0}", type.IsClass);
    Console.WriteLine("\tIsCOMObject = {0}", type.IsCOMObject);
    Console.WriteLine("\tIsContextful = {0}",
                    type.IsContextful);
    Console.WriteLine("\tIsEnum = {0}", type.IsEnum);
    Console.WriteLine("\tIsExplicitLayout = {0}",
                    type.IsExplicitLayout);
    Console.WriteLine("\tIsImport = {0}", type.IsImport);
    Console.WriteLine("\tIsInterface = {0}",
                    type.IsInterface);
    Console.WriteLine("\tIsLayoutSequential = {0}",
                    type.IsLayoutSequential);
    Console.WriteLine("\tIsMarshalByRef = {0}",
                    type.IsMarshalByRef);
    Console.WriteLine("\tIsNestedAssembly = {0}",
                    type.IsNestedAssembly);
    Console.WriteLine("\tIsNestedFamANDAssem = {0}",
                    type.IsNestedFamANDAssem);
    Console.WriteLine("\tIsNestedFamily = {0}",
                    type.IsNestedFamily);
    Console.WriteLine("\tIsNestedFamORAssem = {0}",
                    type.IsNestedFamORAssem);
    Console.WriteLine("\tIsNestedPrivate = {0}",
                    type.IsNestedPrivate);
    Console.WriteLine("\tIsNestedPublic = {0}",
                    type.IsNestedPublic);
    Console.WriteLine("\tIsNotPublic = {0}",
                    type.IsNotPublic);
    Console.WriteLine("\tIsPointer = {0}",
                    type.IsPointer);
```

```
            Console.WriteLine("\tIsPrimitive = {0}",
                              type.IsPrimitive);
            Console.WriteLine("\tIsPublic = {0}",
                              type.IsPublic);
            Console.WriteLine("\tIsSealed = {0}",
                              type.IsSealed);
            Console.WriteLine("\tIsSerializable = {0}",
                              type.IsSerializable);
            Console.WriteLine("\tIsServicedComponent = {0}",
                              type.IsServicedComponent);
            Console.WriteLine("\tIsSpecialName = {0}",
                              type.IsSpecialName);
            Console.WriteLine("\tIsUnicodeClass = {0}",
                              type.IsUnicodeClass);
            Console.WriteLine("\tIsValueType = {0}", type.IsValueType);
        }
        catch(System.NullReferenceException)
        {
            Console.WriteLine
                      ("{0} is not a valid type", typeName);
        }
    }

        public static void Main(string[] args)
        {
            QueryType("System.Int32");
            QueryType("System.Int64");
            QueryType("System.Type");

            QueryType("DemoAttr");
            QueryType("DemoEnum");
            QueryType("DemoBaseClass");
            QueryType("DemoDerivedClass");
            QueryType("DemoStruct");
        }
    }
```

16.3 使用组合体和模块

我们将在第 18 章中对组合体进行更加具体的介绍。为了达到这种交流的目的，只要知道组合体是一个物理文件，这个文件由多个.NET PE 文件组成就足够了。组合体的主要好处是，使您能够从语义上对功能进行分组，这就更加易于配置和翻译了。组合体(和反射对

象层次的顶点)在.NET 运行环境中的表示为一个类 Assembly。

类 Assembly 的用途很多。下面是一些更为普通的任务：

- 迭代一个组合体的类型
- 列出组合体的模块
- 判断标识信息，例如组合体的物理文件名和位置
- 检查版本管理和安全信息
- 检索组合体的入口点

16.3.1　迭代组合体的类型

要迭代一个给定组合体的所有类型，您只需要实例化一个 Assembly 对象，并且为这个组合体请求一个数组 Types。下面是一个例子：

```
using System;
using System.Diagnostics;
using System.Reflection;

class DemoAttr : System.Attribute
{
}

enum DemoEnum
{
}

class DemoBaseClass
{
}

class DemoDerivedClass : DemoBaseClass
{
}

class DemoStruct
{
}

class GetTypesApp
{
    protected static string GetAssemblyName(string[] args)
    {
```

```
        string assemblyName;

        if (0 == args.Length)

        {
            Process p = Process.GetCurrentProcess();
            assemblyName = p.ProcessName + ".exe";
        }
        else
            assemblyName = args[0];

        return assemblyName;
    }

    public static void Main(string[] args)
    {
        string assemblyName = GetAssemblyName(args);

        Console.WriteLine("Loading info for " + assemblyName);
        Assembly a = Assembly.LoadFrom(assemblyName);

        Type[] types = a.GetTypes();
        foreach(Type t in types)
        {
            Console.WriteLine("\nType information for: " +
                              t.FullName);
            Console.WriteLine("\tBase class = " +
                              t.BaseType.FullName);
        }
    }
}
```

注意： 如果要运行那些需要在企业内部网上安全清除的代码——例如使用反射 API 的代码，那么您就需要修改策略。其中的一种做法是用代码访问安全策略(Code Access Security Policy)工具(caspol.exe)。这里有一个使用 caspol.exe 的例子。

```
caspol -addgroup 1.2 -url "file://somecomputer/someshare/*"
SkipVerification
```

这个例子根据所运行代码的 URL，授予了额外的权限——在这个例子中，是 SkipVertification。注意，您也可以为一个给定区域的所有代码或是为一个特定的组合体来进行策略修改，这要根据加密签名或甚至是位的散列表来进行。

要想察看工具 caspol.exe 的有效参数，可以在命令行中输入 caspol—? 或是使用 MSDN 在线文档。

实际上 Main 方法的第一部分并不真正让人感兴趣——这些代码只是判断您是否向应用程序传递了一个组合体名。如果没有，Process 类的静态方法 Get ProcessName 被用来判断当前正在执行的应用程序名字。

接下来您就会发现，绝大部分反射任务很容易。要实例化一个 Assembly 对象，最容易的方法就是调用 Assembly.LoadFrom 方法。这个方法只有一个参数值：一个字符串，代表您所希望载入的物理文件名。从这里开始，对方法 Assembly.GetTypes 的调用返回一个 Type 对象的数组。在这一点上，我们有一个对象，用于描述整个组合体中的每个单类型！最后，应用程序打印了其基本类。

当指定文件 gettypes.exe 或不对应用程序传递任务参数时，执行该程序而产生的输出结果如下所示：

```
Loading info for GetTypes.exe

Type information for: DemoAttr
   Base class = System.Attribute

Type information for: DemoEnum
   Base class = System.Enum

Type information for: DemoBaseClass
   Base class = System.Object

Type information for: DemoDerivedClass
   Base class = DemoBaseClass

Type information for: DemoStruct
   Base class = System.Object

Type information for: AssemblyGetTypesApp
   Base class = System.Object
```

16.3.2　列出组合体的模块

虽然本书的大部分应用程序都只含有单个模块，但是实际上您可以创建由多个模块组成的组合体。有两种方法可以用来从一个 Assembly 对象中检索模块。第一种方法是申请一个由所有模块组成的数组。这种方法允许对所有模块进行迭代，并且可以检索出任何所需要的信息。第二种方法是检索某个特定的模块。让我们来看一看这两种方法。

为了迭代一个组合体的模块，需要创建一个含有多个模块的组合体。要想做到这一点，可以把 GetAssemblyName 移到它自己的类中，并且把那个类放到称为 AssemblyUtils.netmodule 的文件中，如下所示：

```
using System.Diagnostics;

namespace MyUtilities
{
    public class AssemblyUtils
    {
        public static string GetAssemblyName(string[] args)
        {
            string assemblyName;

            if (0 == args.Length)
            {
                Process p = Process.GetCurrentProcess();
                assemblyName = p.ProcessName + ".exe";
            }
            else
                assemblyName = args[0];

            return assemblyName;
        }
    }
}
```

然后使用下面的命令来建立一个 netmodule:

```
csc /target:module AssemblyUtils.cs
```

/target:module 致使编译器产生一个模块，这个模块以后将加入到一个组合体之中。上述命令将创建一个名为 AssemblyUtils .netmodule 的文件。在第 18 章中，当我们更详细地介绍创建组合体和模块时，还会涉及到这条命令的不同选项。

在这里，我们只不过想创建第二个模块，以作为反射的对象。下面就是将要使用类 AssemblyUtils 的应用程序。注意名称空间 MyUtilities 被引用处的 using 语句。

```
using System;
using System.Reflection;
using MyUtilities;

class GetModulesApp
{
    public static void Main(string[] args)
```

```
        {
            string assemblyName = AssemblyUtils.GetAssemblyName(args);

            Console.WriteLine("Loading info for " + assemblyName);
            Assembly a = Assembly.LoadFrom(assemblyName);

            Module[] modules = a.GetModules();
            foreach(Module m in modules)
            {
                Console.WriteLine("Module: " + m.Name);
            }
        }
    }
```

要想编译这个应用程序，并且把 AssemblyUtils .netmodule 加到它的组合体中，您将需要使用下列命令行：

```
csc /addmodule:AssemblyUtils .netmodule GetModules.cs
```

在这里，您已经有了一个带有两个不同模块的组合体了。要想看到这一点，执行应用程序即可——结果如下所示：

```
Loading info for GetModulesApp.exe
Module: GetModulesApp.exe
Module: AssemblyUtils .netmodule
```

正如您从代码中所见到的那样，此处只是简单地实例化了一个 Assembly 对象，并且调用它的 Get Modules 方法。从这里可以迭代返回的数组，并且打印每个数组元素的名字。

16.4　用反射进行后期绑定

几年前，作者在 IBM 多媒体部门工作，开发 IBM/World Book Multimedia Encyclopedia 产品。我们所面临的挑战是，提供一个应用程序，允许用户配置不用的通信协议，用于连接 World Book 服务器。这就必须是一个动态解决方案，因为用户可能不断地向系统增加或从系统删去不同的协议(例如，TCP/IP、IGN、CompuSev，等等)。但是，应用程序必须“知道”提供了什么协议，这样用户才可以选择某个特定的协议，用来配置和使用。我们所提出的解决方案就是用特殊的扩展来创建 DLL，并且把这些 DLL 安装到应用程序文件夹中。然后，当用户想察看已安装协议的列表时，应用程序将调用 Win32 函数 LoadLibrary 来载入每个 DLL，然后再调用函数 GetProcAddrdess 来得到一个函数指针，这个指针指向所需的函数。这是一个在标准 Win32 编程中后期绑定的完美例子。因为在编译时刻，编译器对

这些调用一无所知。正如您将在下面例子中所见到的那样，同样的任务，可以在.NET 中执行，通过使用类 Assembly、类型反射，以及一个称为 Activator 的新类。

让我们创建一个抽象类，称为 CommProtocal。我们将把这个类定义在它自己的 DLL 中，这样就可以被想从它那里派生的多个 DLL 之间共享(注意，命令行参数被嵌入到代码的注释中)。

```
// CommProtocol.cs
// Build with the following command line switches
//        csc /t:library commprotocol.cs
public abstract class CommProtocol
{
   public static string DLLMask = "CommProtocol*.dll";
   public abstract void DisplayName();
}
```

现在将要创建两个独立的 DLL，每个 DLL 都代表一个通信协议并有一个从抽象类 CommProtocol 派生的类。注意，在编译时，这两个 DLL 都需要引用 CommProtocol.dll。这里是 IGN DLL：

```
// CommProtocolIGN.cs
// Build with the following command line switches
//        csc /t:library CommProtocolIGN.cs /r:CommProtocol.dll
using System;

public class CommProtocolIGN : CommProtocol
{
   public override void DisplayName()
   {
      Console.WriteLine("This is the IBM Global Network");
   }
}
```

这里是 TCP/IP DLL：

```
// CommProtocolTcpIp.cs
// Build with the following command line switches
//        csc /t:library CommProtocolTcpIp.cs /r:CommProtocol.dll
using System;

public class CommProtocolTcpIp : CommProtocol
{
   public override void DisplayName()
   {
      Console.WriteLine("This is the TCP/IP protocol");
```

```
        }
    }
```

让我们看一下，动态地载入组合体，搜索一个类型，实例化该类型，以及调用其方法，这都非常容易(顺便提一句，在本书配套光盘中有一个命令文件，称为 BuildLateBinding.cmd，它编译所有文件。)

```
using System;
using System.Reflection;
using System.IO;

class LateBindingApp
{
    public static void Main()
    {

        string[] fileNames = Directory.GetFiles
                             (Environment.CurrentDirectory,
                              CommProtocol.DLLMask);
        foreach(string fileName in fileNames)
        {
            Console.WriteLine("Loading DLL '{0}'", fileName);

            Assembly a = Assembly.LoadFrom(fileName);

            Type[] types = a.GetTypes();
            foreach(Type t in types)
            {
                if (t.IsSubclassOf(typeof(CommProtocol)))
                {
                    object o = Activator.CreateInstance(t);

                    MethodInfo mi = t.GetMethod("DisplayName");

                    Console.Write("\t");
                    mi.Invoke(o, null);
                }
                else
                {
                    Console.WriteLine("\tThis DLL does not have " +
                    "CommProtocol-derived class defined");
                }
            }
        }
```

```
        }
    }
```

首先，用类 System.IO.Directory 找出当前文件中带有 CommProtocal *.dll 所有的 DLL。方法 Dirdectory.Get Files 将访问一个由 String 类型的对象组成的数组，每个 String 代表与搜索条件匹配的文件名字。然后可以使用一个 foreach 循环来迭代整个数组，这要调用本章前面部分所学过的 Assembly.Load From 方法。一旦为给定的 DLL 创建了组合体，就迭代组合体的所有类型，调用方法 Type.SubClassOf 来判断这个组合体是否有一个类型是从 CommProtocal 派生的。现在，设想如果找到一个，就有了一个有效的 DLL，可以用来完成工作。当真的找到了某个组合体，这个组合体含有从 CommProtal 处派生的类型，就实例化一个 Activator 对象，并且把 type 对象传递到它的构造函数。您可能已经从它的名字中猜出来，类 Activator 用于动态创建或激活一个类型。

然后，使用方法 Type.GetMethod 来创建一个对象 MethodInfo,指定方法名 Displayname。一旦完成，就可以使用对象 Methodinfo 的方法 Invoke，把激活的类型传递给它，然后 DLL 的 DisplayName 方法就被调用了。

16.5 在运行时刻创建并执行代码

现在您已经看到，如何在运行时刻反射类型，对后期绑定代码以及动态执行代码，让我们采取下一逻辑步骤，并“即席”创建代码。在运行时刻创建类型需要使用 System.Reflection.Emit 名称空间。使用该名称空间中的类，您就可以在内存中定义一个组合体，为组合体创建一个模块，为模块定义新类型(包括它的成员)，甚至可以为应用程序的逻辑发送 MSLL 操作码。

虽然这个例子中的代码十分简单，我们还是把服务器代码——一个 DLL，含有一个类，这个类创建一个被称为 Hello World 的方法——与客户端代码分离开，而客户端的代码则是一个应用程序，这个应用程序实例化一个代码产生类，并且调用其 Hello World 方法，如下所示：

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace ILGenServer
{
    public class CodeGenerator
    {
        public CodeGenerator()
```

```
{
    // Get current currentDomain.
    currentDomain = AppDomain.CurrentDomain;

    // Create assembly in current currentDomain.
    assemblyName = new AssemblyName();
    assemblyName.Name = "TempAssembly";

    assemblyBuilder =
           currentDomain.DefineDynamicAssembly
           (assemblyName, AssemblyBuilderAccess.Run);

    // create a module in the assembly
    moduleBuilder = assemblyBuilder.DefineDynamicModule
                     ("TempModule");

    // create a type in the module
    typeBuilder = moduleBuilder.DefineType
                    ("TempClass",
                    TypeAttributes.Public);
    // add a member (a method) to the type
    methodBuilder = typeBuilder.DefineMethod
                      ("HelloWorld",
                       MethodAttributes.Public,
                       null,null);

    // Generate MSIL.
    msil = methodBuilder.GetILGenerator();
    msil.EmitWriteLine("Hello World");
    msil.Emit(OpCodes.Ret);

    // Last "build" step : create type.
    t = typeBuilder.CreateType();
}

AppDomain currentDomain;
AssemblyName assemblyName;
AssemblyBuilder assemblyBuilder;
ModuleBuilder moduleBuilder;
TypeBuilder typeBuilder;
MethodBuilder methodBuilder;
ILGenerator msil;
object o;
```

```
            Type t;
            public Type T
            {
                get
                {
                    return this.t;
                }
            }
        }
    }
```

首先，我们从当前域实例化一个 AppDomain 对象(您将会在第 17 章中发现，app 域在功能上与 Win32 进程相类似)。此后，我们实例化一个对象 AssemblyName。我们将在第 18 章中详细介绍类 AssemblyName——简单地说，就是一个由组合体缓存管理器使用的类，用于检索关于组合体的信息。一旦有了当前 app 域和一个被实例化了的组合体名，我们就调用 AppDomain.DefineDynamicAssembly 方法，来创建一个新的组合体。注意，我们所传递的两个参数值，一个是组合体的名字，还有一个是组合体被访问的模式。AssemblyBuilderAccess.Run 指明，该组合体可以从内存中执行，但是不能被保存。方法 AppDomain.Define DynamicAssembly 返回一个对象 AssemblyBuilder，然后我们把这个对象转换为 Assembly 对象。在这里，我们在内存中有一个具备完全功能的组合体。现在我们需要创建这个组合体的临时模块和这个临时模块的类型。

我们从调用方法 Assembly.DefineDynamicModule 来检索一个 ModuleBuilder 对象为开始。一旦得到了 ModuleBuilder 对象，我们将调用这个对象的 DefineType 方法来创建一个 TypeBuilder 对象，把类型的名字(“TempClass”)和用来定义的属性(TypeAttributes.Public)传递给这个方法。现在我们就有了一个 TypeBuilder 对象，我们可以创建任何想要的成员类型。在这个例子里，我们用方法 TypeBuilder.DefineMethod 来创建了一个方法。

最后，我们有了一个全新的类型 Tempclass，这个类型中带有一个嵌入方法，叫做 HelloWorld。现在我们所要做的就是决定在这个方法中放入什么代码。要做到这一点，代码中需要用方法 MethodBuilder.GetILGenerator 实例化一个 ILGenerator 对象，并且还要调用不同的 ILGenerator 方法来把 MSIL 代码写入方法中。

注意，这里我们可以通过各种不同的 ILGenerator 方法来使用标准代码，例如 Console.Writeline，或者我们可以使用 ILGenerator.Emil 方法来“发射”MSIL 操作码。方法 ILGenerator.Emit 把与 MSIL 操作码直接相关的 Opcode 类成员作为其惟一的参数值。

最后，我们调用方法 TypeBuilder.CreateType。这总是在定义了新类型的成员以后，最后一步执行。然后我们用 Type.GetType 方法为一个新类型检索 Type 对象。这个对象被存放在一个成员变量中，以后被客户端应用程序检索。

现在，客户端要做的就是检索 CodeGenerator 的 Type 成员，创建一个活化实例，从该

类型实例化一个 MethodInfo 对象，然后使用这个方法。下面就是这些动作的代码，其中还有一些错误检查，以确保这些代码正常运行。

```
using System;
using System.Reflection;
using ILGenServer;

public class ILGenClientApp
{
    public static void Main()
    {
        Console.WriteLine("Calling DLL function to generate " +
                          "a new type and method in memory...");
        CodeGenerator gen = new CodeGenerator();

        Console.WriteLine("Retrieving dynamically generated type...");
        Type t = gen.T;
        if (null != t)
        {
            Console.WriteLine("Instantiating the new type...");
            object o = Activator.CreateInstance(t);

            Console.WriteLine("Retrieving the type's " +
                              "HelloWorld method...");
            MethodInfo helloWorld = t.GetMethod("HelloWorld");
            if (null != helloWorld)
            {
                Console.WriteLine("Invoking our dynamically " +
                                  "created HelloWorld method...");
                                  helloWorld.Invoke(o, null);
            }
            else
            {
                Console.WriteLine("Could not locate " +
                                  "HelloWorld method");
            }
        }
        else
        {
            Console.WriteLine("Could not access Type from server");
        }
    }
}
```

如果现在编译并执行这个应用程序，您将会看到下列输出：

```
Calling DLL function to generate a new type and method in memory...
Retrieving dynamically generated type...
Instantiating the new type...
Retrieving the type' s HelloWorld method...
Invoking our dynamically created HelloWorld method...
Hello World
```

16.6 本章小结

反射是指能够在运行时刻找到类型信息。反射 API 允许您完成这样的工作，例如迭代一个组合体的模块，迭代一个组合体的类型，以及检索类型在设计时的不同特征。高级反射任务包括使用反射来动态地运用方法以及使用类型(通过后期绑定完成)，甚至还可以在运行时刻创建和执行 MSIL 代码。

第 17 章　与非控代码互操作

如果一种新的语言或开发环境忽略了以前遗留的系统和代码，而且只提供一种方法来编写系统，那么，这种语言或开发环境的生存周期将是短暂的。无论一项新技术多么“伟大”，它都必须考虑与现有旧技术共存一段时间的问题。为了达到这一目的，.NET 和 C#设计组使得程序员可以很容易地通过使用非控代码与现有代码进行互操作。“非控代码”指的是不被.NET 运行环境管理或控制的代码。在这一章里将介绍三个主要的例子，这些例子是关于.NET 中的非控代码的，如下所示：

- **平台启用服务** 该服务允许.NET 代码访问现有的非控动态链接库(DLL)中的函数、结构，甚至还有回调。
- **非安全代码** 编写非安全代码允许 C#程序员在 C#中使用构造(例如指针)，但是那些代码要受.NET 运行环境的管理。
- **COM 互操作性** 该术语指的是，.NET 代码使用 COM 组件以及 COM 应用程序使用.NET 组件的能力。

17.1 平台启用服务

.NET 平台启用服务(Platform Invocation Services)——有时候也称为 PInvoke——允许受控代码与从 DLL 导出的函数和结构一起工作。在这一节里，我们将看一看如何调用 DLL 函数，还要看看用于在.NET 应用程序和 DLL 之间汇集(marshal)数据的特性。

因为您没有向 C#编译器提供 DLL 函数的源代码，所以必须向编译器指定本地方法的信号和关于所有返回值的信息，以及怎样向 DLL 汇集参数。

注意： 正如您所了解的那样，可以用 C#或其他.NET 编译器来创建 DLL。在这一节里已经避免了使用术语“非控 Win32 DLL”。每当提到 DLL 时，实际上就是指非控类的 DLL。

17.1.1 声明导出的 DLL 函数

首先要看的事例就是如何在 C#中声明一个简单的 DLL 函数。我们会使用一个很快变

成规范的.NET PInvoke 例子，Win32“MessageBox”例子，来作为开始。然后，我们将转到参数汇集的高级技术。

正如您在第 8 章中所学到的那样，特性被用于为 C#类型提供设计时刻的信息。通过反射，这些信息可以在运行时刻被查询。C#利用一个特性来使您能够向编译器描述应用程序将要调用的 DLL 函数。该特性被称为 DllImport 特性，使用它的语法如下所示：

[DllImport(*dllName*)]

accessModifier static extern *retValue dllFunction(param1, param2,...);]*

正如您所见到的，导入一个 DLL 函数与把 DllImport 特性(向它的构造函数传递 DLL 名字)附到您想调用的 DLL 程序上一样容易。要特别注意这样的事实，必须在您定义的函数上使用 static 和 extern 限定符。下面的这个 MessageeBox 例子显示了 PInvoke 是多么易于使用：

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

class PInvokeIApp
{
    [DllImport("user32.dll")]
    static extern int MessageBoxA(int hWnd,
                                  string msg,
                                  string caption,
                                  int type);

    public static void Main()
    {
        MessageBoxA(0,
                    "Hello, World!",
                    "This is called from a C# app!",
                    0);
    }
}
```

运行该程序会导致弹出一个预期的消息框，带有“Hello, World!”语句。

注意那条引用 System.Runtime.InteropServices 名称空间的 using 语句。该名称空间定义了 DllImport 特性。此后要注意定义了一个 MessageBoxA 方法，在 Main 方法中被调用。但是，如果您想调用不同于 DLL 函数名的某些内部 C#方法时会怎么样？您可以使用 DUImport 特性的一个命名参数来做到这一点。接下来的代码中，编译器将方法叫做 MessageBoxA。因为并没有在 DllImport 特性指定 DLL 函数名，所以编译器假设这两个名字相同。

```
[DllImport("user32.dll")]
   static extern int MessageBoxA(int hWnd,
                                 string msg,
                                 string caption,
                                 int type);
```

要想了解怎样改变默认行为，看一下另一个例子——这次要在调用 DLL 的 MessageBoxA 函数时使用一个 MsgBox 内部名字。

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

class PInvoke2App
{
   [DllImport("user32.dll", EntryPoint="MessageBoxA")]
   static extern int MsgBox(int hWnd,
                            string msg,
                            string caption,
                            int type);

   public static void Main()
   {
      MsgBox(0,
             "Hello, World!",
             "This is called from a C# app!",
             0);
   }
}
```

正如您所见到的，只要指定 DllImport 特性的命名参数 Entrypoint，就能够根据自己的喜好为外部 DLL 函数命名内部等价名字。

在讨论参数汇集之前，我们最后要看的是参数 CharSet。该参数允许您指定 DLL 文件所用的字符集。典型地，当编写 C++应用程序时，您并不显式地指定 MessageBoxA 或 MessageBoxW——有一条编译批注已经让编译器了解到您是在用 ANSI 字符集还是 Unicode 字符集。这就是为什么在 C++中，您调用 MessageBox 而编译器判断要调用哪个字符集版本。同样，您可以在 C#中对 DllImport 特性指定目标字符集，并且让该字符集指出调用哪个版本的 MessageBox 函数。在下列例子中，函数 MessageBoxA 仍然被作为一个值的结果调用，该值是通过命名参数 Charset 来传递给 DllImport 特性的。

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

class PInvoke3App
```

```
{
    // CharSet.Ansi will result in a call to MessageBoxA.
    // CharSet.Unicode will result in a call to MessageBoxW.
    [DllImport("user32.dll", CharSet=CharSet.Ansi)]
    static extern int MessageBox(int hWnd,
                                 string msg,
                                 string caption,
                                 int type);

    public static void Main()
    {
        MessageBox(0,
                   "Hello, World!",
                   "This is called from a C# app!",
                   0);
    }
}
```

使用参数 CharSet 的好处是，可以在应用程序中使用一个变量，并且用该变量来控制函数被调用的版本(ANSI 或 Unicode)；如果从一个版本改变为另一个版本无需改变所有代码。

17.1.2 与 C#共用回调函数

不但一个 C#程序可以调用一个 DLL 函数，而且 DLL 函数也可以在回调场景中调用应用程序中的 C#方法。回调场景包括任何 Win32 EnumXXX 函数的使用，在这些函数中，您可以调用一个函数来枚举某些对象，把 Windows 对每个找到的项调用的函数指针传递给它。这是通过 PInvoke 的联合——调用 DLL 函数——及代表——定义回调——来实现的。关于代表的一些内容，请参阅第 14 章。

下面的代码枚举并打印了系统中所有窗口的标题：

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

class CallbackApp
{
 [DllImport("user32.dll")]
 static extern int GetWindowText(int hWnd, StringBuilder text, int
 count);

 delegate bool CallbackDef(int hWnd, int lParam);
```

```
[DllImport("user32.dll")]
static extern int EnumWindows (CallbackDef callback, int lParam);

static bool PrintWindow(int hWnd, int lParam)
   {
      StringBuilder text = new StringBuilder(255);
      GetWindowText(hWnd, text, 255);

      Console.WriteLine("Window caption: {0}", text);
      return true;
   }

   static void Main()
   {
      CallbackDef callback = new CallbackDef(PrintWindow);
      EnumWindows(callback, 0);
   }
}
```

首先，用 DllImport 特性定义了 Win32 函数 EnumWindows 和 GetWindowsText。然后，定义了一个被称为 CallbackDef 的代表和一个名为 PrintWindows 的方法。之后，在 Main 中所需要做的就是实例化 CallbackDef 代表(把方法 PrintWindows 传递给它)并且调用 EnumWindows 方法。对系统中找到的每个窗口，Windows 将调用 PrintWindows 方法。

PrintWindows 方法很有趣，因为该方法使用了类 StringBuilder 来创建一个固定长度的字符串，该字符串被传递给函数 GetWindowsText。这就是为什么函数 GetWindowsText 会以如下方式定义的原因：

```
static extern int GetWindowText(int hWnd, StringBuilder text, int
count);
```

无论如何，这样做的理由是不允许 DLL 函数修改一个字符串，因此，您就无法使用该类型。而且，即使您试着通过引用来传递，这里也无法为调用代码把一个字符串实例化到正确的长度。于是，此处使用了类 StringBuilder。一个 StringBuilder 对象可以由被调用函数解除引用并修改，只要文本长度没有超出传递给 StringBuilder 构造函数的最大长度即可。

17.1.3　汇集和 PInvoke

即使您并没有见过典型的汇集或是定义其如何工作，每次您调用一个 DLL 函数时，.NET 也必须把参数汇集到那个函数，并且把值返回给进行调用的.NET 程序。在本章的前一个例子里，这可以自动发生，因为.NET 已经为每种.NET 类型定义了一个默认的本

地类型。例如，传递给 MessageBoxA 和 MessageBoxW 函数的第二个和第三个参数被定义为类型字符串，然而，C#编译器知道 Win32LPSTR 等价于一个 C#字符串。但是如果您想覆盖掉默认的.NET 汇集行为时，应该怎么办呢？要做到这一点，可以使用 MarshalAs 特性，该特性在 System.Runtime.InteropService 名称空间中也定义了。

在下面的例子中，再次使用了 MessageBox，这样可以保持例子的简单化。在这里，选择使用 Win32 MessageBox 函数的 Unicode 版本。正如我们从前一节中了解的那样，只需要为 DllImport 特性的 CharSet 命名参数指定一个枚举项 CharSet.Unicorde。但是，在该例子中，想让编译器把数据作为宽字符(LPWSTR)汇集，因此使用 MarshalAs 特性，并且用 Unmanaged Type 枚举项指定了类型想要转换成的类型。这里是代码：

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

class PInvoke4App
{
    [DllImport("user32.dll", CharSet=CharSet.Unicode)]
    static extern int MessageBox(int hWnd,
                                 [MarshalAs(UnmanagedType.LPWStr)]
                                 string msg,
                                 [MarshalAs(UnmanagedType.LPWStr)]
                                 string caption,
                                 int type);

    public static void Main()
    {
        MessageBox(0,
                   "Hello, World!",
                   "This is called from a C# app!",
                   0);
    }
}
```

注意，MarshalAs 特性可以被附加到方法参数上(正如本例一样)，方法返回值，以及结构和类的字段。还要注意，要想改变为一个类返回值的默认汇集，您需要把 MarshalAs 特性附加于方法本身。

17.2 编写非安全代码

当从 C++转向 C#时，许多人都担心这样一个问题，即他们是否可以“完全控制”内存

的处理，以防他们有这样的需要，该问题与非安全代码有关。非安全代码尽管听上去似乎不太安全，但本质上并非不安全和不可信任的——实际上是指.NET 运行环境不会为这些代码控制内存中的分配或解除分配。当您正在使用指针与以前遗留下来的代码(例如 C API)进行通信时，或者当您的应用程序要求直接处理内存(最典型的是为了性能原因)时，能够编写非安全代码是非常有好处的。

可以通过使用两种关键字来编写非安全代码：unsafe 和 fixed。关键字 unsafe 指定标记块在非控环境中运行。该关键字可以用于所有的方法，包括构造函数和属性，甚至还有方法中的代码块。关键字 fixed 负责受控对象的固定(pinning)。Pinning 是一种动作，向垃圾收集器(Garbage Collector ,GC)指定一些不能被移动的对象。当您这样做时，在程序执行期间，对象被定位以及再定位，而内存中的“空间”就开放了。为了不在内存中产生碎片，.NET 运行环境把对象四处移动，以便于最有效地利用内存。很明显，用指针指向某个具体的内存地址并不是个好主意，然后——对您而言是不得而知的——.NET 运行环境从该地址处移走对象，只留下了那个无效的指针。因为 GC 要在内存中移动对象的原因是提高应用程序效率，所以您使用时应该谨慎。

17.2.1　在 C#中使用指针

让我们来看一看在 C#中关于使用指针和非安全代码的一些规则，然后我们将深入讨论几个例子。我们只能得到值类型、数组和字符串的指针。还要注意，在数组的情况下，第一个元素必须是值类型，因为 C#实际上是返回一个指向数组第一个元素的指针，而不是返回数组自身。因此，从编译器的角度来说，它仍然返回一个指向值类型的指针，而不是引用类型。

表 17.1 显示了标准 C/C++指针的语义是如何在 C#中被支持的。

表 17.1　C/C++指针操作符

操作符	说　明
&	address-of 操作符返回一个代表变量的内存地址的指针。
*	Dereference 操作符用于指示指针指向的值。
->	dereferencing and member access 操作符用于成员存取和指针非关联化。

下面的例子对 C 或 C++开发人员来说应该很熟悉。这里调用了一个方法，该方法把两个指向 int 类型变量的指针作为参数，并且在返回到调用函数之前修改这两个变量的值。该例子体现了如何在 C#中使用指针。

```
// Compile this application with the /unsafe option.
```

```
using System;

class Unsafe1App
{
    public static unsafe void GetValues(int* x, int* y)
    {
        *x = 6;
        *y = 42;
    }

    public static unsafe void Main()
    {
        int a = 1;
        int b = 2;
        Console.WriteLine("Before GetValues() : a = {0}, b = {1}",
                          a, b);
        GetValues(&a, &b);
        Console.WriteLine("After GetValues() : a = {0}, b = {1}",
                          a, b);
    }
}
```

该例子在编译时需要带有编译器选项/unsafe。该程序的输出如下所示：

```
Before GetValues() : a = 1, b = 2
After GetValues() : a = 6, b = 42
```

17.2.2 fixed 语句

fixed 语句的语法如下：

fixed *(type* ptr = expression) statement*

正如前面所提到的，fixed 语句告诉 GC 不必关心指定的变量。注意，其中 type 是非控类型，或者是 void。而 expression 是任何产生一个 type 指针的表达式。Statement 指的是变量固定所应用的代码块。下面是一个简单的例子。

```
using System;

class Foo
{
    public int x;
}

class Fixed1App
```

```
{
    unsafe static void SetFooValue(int* x)
    {
        Console.WriteLine("Dereferenced pointer to modify foo.x");
        *x = 42;
    }

    unsafe static void Main()
    {
        // Create an instance of the structure.
        Console.WriteLine("Creating the Foo class");
        Foo foo = new Foo();

        Console.WriteLine("foo.x intialized to {0}", foo.x);

        // The fixed statement pins the foo object until
        // the enclosing compound statement ends.
        Console.WriteLine("Setting pointer to foo.x");
        // Assign the address of the foo object to a Foo*.
        fixed(int* f = &foo.x)
        {
            Console.WriteLine("Calling SetFooValue passing " +
                              "pointer to foo.x");
            SetFooValue(f);
        }

        // Show that we did alter the member via its pointer.
        Console.WriteLine("After return from " +
                          "SetFooValue, foo.x = {0}", foo.x);
    }
}
```

这段代码实例化一个称为 Foo 的类，而且在一条 fixed 语句中，当把该对象第一个成员的地址赋给一个 int*类型的变量时(该类型由方法 SetFooValue 所请求)，固定该对象。注意，fixed 语句只是用于封闭那段如果 GC 移动 Foo 对象时将要受影响的代码。对于较大或运行时间较长的代码块，而您又想把对象固定时间压缩至最少，这是个既巧妙又很重要的方法。编译并执行这段代码会产生下列输出：

```
Creating the Foo class
foo.x intialized to 0
Setting pointer to foo.x
Calling SetFooValue passing pointer to foo.x
Dereferenced pointer to modify foo.x
After return from SetFooValue, foo.x = 42
```

注意： 关于固定变量有一点很重要，C#编译器并不把对固定变量的访问限制在非安全范围内。例如，可以使用固定变量时，用 r-value 和 l-value，使得该固定变量定义在比非安全块更广的范围内。很明显，这就导致了可以在非安全块以外使用一个非安全值。在这种情况下，如果编译器发出一条警告或错误，那么当使用 r-values 固定变量时就由开发人员负责处理。

17.3 COM 互操作性

如果您很纳闷，不知您这些年来编写的 COM 组件是如何与.NET 运行环境“共舞”的话，那么这一节内容可以消除您的疑惑。本节将向您介绍传统 COM(Classic COM)组件在.NET 世界中的定位。

17.3.1 一个全新的领域

正如您在本书中所见到的，毫无疑问，.NET 环境和 C#语言结合起来形成了一种强大的方法，用于构造组件化的系统。但是，这几年来历尽千辛万苦创建的大量可重用 COM 组件怎么办呢？是否.NET 将使组件不复存在？这些组件可不可以“手拉手”地与.NET 受控运行环境一起工作呢？令那些利用 COM 编程的人欣慰的是：COM 仍然存在，而且.NET 框架——受控应用程序可以利用现有的 COM 组件。您将会发现，传统的 COM 组件通过互操作层(COM Interop)与.NET 运行环境交互。 互操作层处理在受控运行环境和非受控区域中的 COM 组件操作之间传递的所有消息。

17.3.2 入门

因为完整的 COM Interop 层一开始的时候可能会让人觉得过于繁杂，所以当您想从.NET 应用程序中使用一个 COM 组件时，我们先暂时绕开所有的技术定义，而直接以一个现实的例子作为开始。之后将解释其过程以及怎样才能把您所学到的知识应用于自己的程序中。

在该例子里，我们将假设一个 AirlineInfo COM 组件，它是用 Microsoft Visual C++和 ATL 编写的。这里不介绍创建该组件所需的完整步骤，因为重点在于.NET 和 C#。但是，将介绍一些重要的代码，并且还提到整个 Visual C++ Project，它们包括于本书的配套光盘中。

COM 组件被用来设计成产生某条特定航线的详细抵达信息。为了简单起见，让我们假

设组件返回 AirScooby IC5678 航线的细节，并且还为其他任何航线返回错误信息。此处有意把错误机制加入，这样就可以看到 COM 组件引发的错误如何被传回，并被正在进行调用的.NET 客户端程序所截获。

下面是 COM 组件的 IDL。

```
interface IAirlineInfo : IDispatch
{
   [id(1), helpstring("method GetAirlineTiming")]
      HRESULT GetAirlineTiming([in] BSTR bstrAirline, [out,retval]
BSTR* pBstrDetails);

   [propget, id(2), helpstring("property
      LocalTimeAtOrlando")] HRESULT
      LocalTimeAtOrlando([out, retval] BSTR
      *pVal);
};
```

这里没什么新的内容，即使对最初级的 COM 开发人员来说也是如此。我们把一个接口命名为 LairlineInfo，它带有两个方法：GetAirlineTiming 和 LocalTimeAtOrland。现在让我们看看方法 GetAirlineTiming 的实际实现：

```
STDMETHODIMP CAirlineInfo::GetAirlineTiming(BSTR
   bstrAirline, BSTR *pBstrDetails)
{
   _bstr_t bstrQueryAirline(bstrAirline);
   if(NULL == pBstrDetails) return E_POINTER;

   if(_bstr_t("Air Scooby IC 5678") ==
      bstrQueryAirline)
   {
      // Return the timing for this Airline.
      *pBstrDetails =
         _bstr_t(_T("16:45:00 - Will
         arrive at Terminal 3")).copy();
   }
      else
      {
         // Return an error message.
         return Error(LPCTSTR(_T("Not available" )),
                   __uuidof(AirlineInfo),
                   AIRLINE_NOT_FOUND);
   }
   return S_OK;
}
```

GetAirlineTiming 方法有两个参数值。第一个参数值(bstrAirline)是代表航线的 BSTR，而第二个参数组(pBstrDetails)是一个输出参数，返回抵达信息(本地时间和登机门)。在该方法中，我们检查输入的 bstrAirline 参数的值是否等于“Air Scooby IC 5678”。如果相等，我们返回硬编码抵达信息。如果该值不是我们想要的，我们就调用一个错误方法，返回这样的事实，即我们只支持一条航线。

在以一种“不寻常”的方式对组件进行了基本的回顾之后，然后我们再看一看从组件类型库(typelib)中生成元数据，这样，.NET 客户端就可以使用该元数据与组件“交谈”并调用其方法。

17.3.3 从 COM 类型库生成元数据

一个需要同 COM 组件“交谈”的.NET 应用程序无法直接使用由那些组件所列出的功能。为什么不能？正如我们在第 16 章中所见的，.NET 运行环境是被设计用来与具有元数据的组件一起工作的，而 COM 是设计用来通过注册和一系列由组件实现的询问方法进行工作的。因此，要想使该 COM 组件能够在.NET 世界中使用，我们需要做的第一件事就是为它生成某些元数据。在 COM 组件的情况中，运行环境用元数据层来判断类型信息。在运行时刻使用类型信息，以便于生成“运行环境可调用包装” (runtime callable wrapper，RCW)(参见图 17.1)。当.NET 应用程序与 COM 对象交互时，RCW 处理对 COM 对象的实际激活，并且还要处理汇集要求。RCW 还完成许多其他的工作，例如管理对象标识、对象生存周期以及接口缓冲区。

对象生存周期管理十分关键，因为.NET GC 把对象到处移动，并且当对象不再使用时，自动处理这些对象。RCW 服务于这样的目的：“告知”.NET 应用程序它正在与受控.NET 组件交互，同时又使非受控空间中的 COM 组件“觉得”COM 对象是被传统的 COM 客户端调用的。RCW 的创建和行为根据对 COM 对象是早期绑定还是后期绑定而有所不同。在底层，是 RCW 来完成所有困难的工作，并且把所有的方法调用变成非受控世界里 COM 组件的相应 Vtable 调用。它的行为基本上就像是受控世界和非受控 IUnknown 世界之间的“亲善大使”。

现在再为 AirlineInfo COM 组件生成元数据包装。为了做到这一点，我们需要使用一种被称为类型库导入器(Type Library Importer)的工具(tlbimp.exe)。该工具可与.NET SDK 共用，并且可以被用于读取一个 COM 类型库，还可用来生成相应的元数据包装，而且元数据包装含有能被.NET 运行环境理解的类型信息。要想这样做，您需要从本书配套光盘上安装演示程序，并且找到 AirlineInfo 组件。

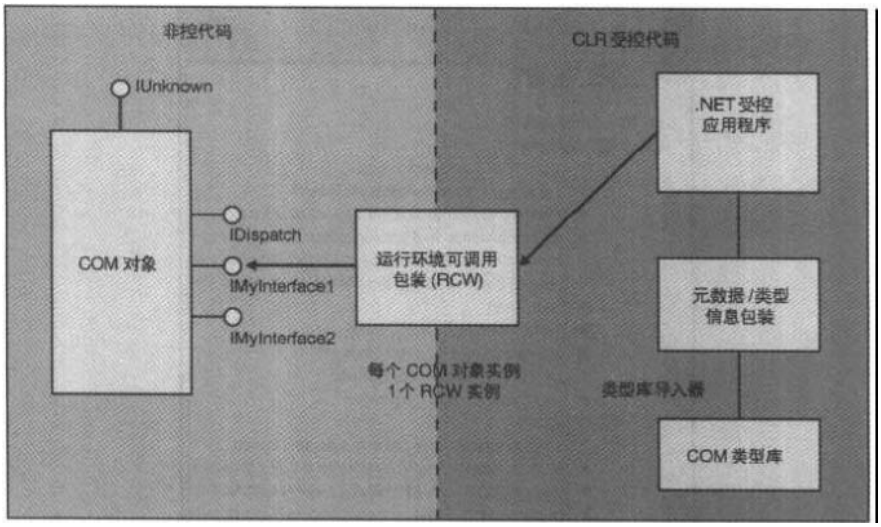


图 17.1　.NET COM 互操作性的基本组件

一旦完成了上述步骤，输入下列命令：

```
TLBIMP AirlineInformation.tlb /out:AirlineMetadata.dll
```

这条命令告诉 TLBIMP 读取 AirlineInfo COM 类型库，并且生成相应的元数据包装，称为 AirlineMetadata.dll。如果一切正常，您将会见到下列消息：

```
TypeLib imported successfully to AirlineMetadata.dll
```

那么，生成的元数据中含有什么类型的信息，还有它看上去是什么样的呢？COM 一族总是很喜欢 OleView.exe 工具，因为它能使我们看到类型库的内容。幸运的是，.NET SDK 也有类似功能：在第 2 章中介绍过的名为 ILDASM 的 IL 分解器可以让我们浏览为受控组合体生成的元数据和 MSIL 代码。正如您在第 16 章中所学到的，每个受控组合体中都含有自描述元数据，而且当您需要研究这种自描述元数据时，ILDASM 是一个非常有用的工具。接着，用 ILDASM 打开 AirlineMetadata.dll，您将看见与图 17.2 中所示内容相类似的结果。

从生成的元数据中，您可以发现方法 GetAirlineTiming 作为类 AirlineInfo 的公共成员被列出来。这里还有一个构造函数是为类 AirlineInfo 生成的。注意，方法参数已经被自动替代为等价的.NET 副本形式了。在该例子中，BSTR 已经被参数 System.String 代替了。还要注意到，在方法 GetAirlineTiming 中被标上[out, retval]的参数已经被转换成该方法的实际返回值了(以 System .String 类型返回)。此外，从 COM 对象返回的任何错误 HRESULT 值——例如一个错误或失败的商业逻辑——被作为异常提出。

图 17.2 ILDASM 是浏览元数据和受控组合体的 MSIL 的极佳工具

17.3.4 早期绑定 COM 组件

在生成一个.NET 客户端所需要的元数据之后，让我们试着从.NET 客户端调用我们的 COM 对象中的 GetAirlineTiming 方法。这里有一个 C#客户端应用程序，该程序利用我们以前生成的元数据创建了一个 COM 对象，并且调用了方法 GetAirlineTiming。注意，在该例子中，我们用到了早期绑定。简而言之，此处将向您展示两个例子，这两个例子演示了动态类型探索和后期绑定。

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient1App
{
    public static void Main()
```

```
        {
            ////////////////////////////////////////////////
            /// EARLY BINDING EXAMPLE
            ////////////////////////////////////////////////
            String strAirline = "Air Scooby IC 5678";
            String strFoodJunkieAirline = "Air Jughead TX 1234";
            try
            {
                AirlineInfo objAirlineInfo;

                // Create a new AirlineInfo Object.
                objAirlineInfo = new AirlineInfo();

                // Display the output after calling
                // the GetAirileTiming method.
                Console.WriteLine("Details for Airline {0} --> {1}",
                strAirline,objAirlineInfo.GetAirlineTiming(strAirline));

                // ERROR: The following will result in a thrown
                // exception!
                // Console.WriteLine("Details for Airline {0} --> {1}",
                // strFoodJunkieAirline,objAirlineInfo.GetAirlineTiming
                // (strFoodJunkieAirline));
            }
            catch(COMException e)
            {
                Console.WriteLine("Oops- We encountered an error " +
                                "for Airline {0}. The Error message " +
                                "is : {1}. The  Error code is {2}",
                                strFoodJunkieAirline ,
                                e.Message,e.ErrorCode);
            }
        }
    }
```

这里发生的是，运行环境构造一个 RCW，把元数据类方法和字段映射到由 COM 对象实现的接口所打开的方法和属性上去。为 COM 对象的每个实例创建一个 RCW 实例。.NET 运行环境只参与管理实际 COM 对象上的引用计数。如图 17.2 所示，AirlineInfo 元数据在一个名为 AIRLINEINFORMATIONLib 的名称空间里定义。.NET 客户端可以看见所有的接口方法，就好像这些方法是类 AirlineInfo 的成员似的。我们所需要做的就是用操作符 new 来创建类 AirlineInfo 的一个实例，并且调用所创建方法的 public 类方法。当方法被调用时，RCW 把该调用转成相应的 COM 方法调用。RCW 还处理所有的汇集和对象生存周期事务。

对.NET 客户端而言，这看上去就只不过是创建一个典型的受控对象，并且调用它的一个 public 类成员而已！

注意，无论何时，只要 COM 方法引发一个错误，RCW 就捕获该错误。该错误被转换成一个等价的 COMException 类中(在名称空间 System.Runtime.InteropServices 中可以找到该类。)当然 COM 对象仍然需要为使该错误传播能够运行而实现 ISupportErrorInfo 接口，这样 RCW 就知道了您的对象还提供了扩展的错误信息。.NET 客户端可以利用通常的 try-catch 异常处理机制来捕获错误，并且客户端可以访问到错误号、错误描述、异常的来源以及其他任何对 COM 件客户端可用的细节。现在让我们来进一步了解该例子，并看一看其他绑定 COM 组件的方式。

17.3.5 利用动态类型探索选择 COM 接口

当.NET 客户端要访问由 COM 对象实现的另一个接口时，从.NET 客户端的角度来讲，标准的 QueryInterface 场景是如何工作的呢？对于另一个接口的 QI，您所需要做的就是把当前对象转换为您所需要的另一个接口，这样，您的 QI 就完成了！现在您已经可以调用所需接口的所有方法和属性了。

同样，RCW 在底层完成所有的困难工作。在某种程度上，这与 Visual Basic 中 COM 客户端程序员无需编写任何显式的与 QuerryInterface 相关代码有一定的相似之处——当您把一个对象类型设定到另一个相关类型的对象上时，它仅仅简单地为您完成 QI。

下面我们将具体地看一下其简单过程。在我们的例子中，假设您想调用接口 IAirportFacilities 上的方法，这是由我们的 COM 对象实现的另一个接口。为了能够调用该接口上的方法，您应该把 AirlineInfo 对象转换到 IAirportFacilities 接口。现在您可以调用所有成为 IairportFacilities 接口的一部分方法了。但是在执行转换之前，您可能想要检查当前所持有的对象实例是支持还是实现您正在查询的接口类型。可以用类 System.Type 中的 IsInstanceOf 方法来检查。如果该方法返回 true，那么您就知道 QI 成功了，并且可以安全地执行转换。如果您把对象转换到某个任意接口上，而该对象对此接口并不支持，那么就有一个 System.InvalidCastException 异常被抛出。这样 RCW 就确保了您可以只转换到那些由 COM 对象实现的接口上。下面就是这些代码：

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient2App
{
    public static void Main()
```

```
        {
            ////////////////////////////////////////////////
            /// QUERY INTERFACE/ RT type Checking
            ////////////////////////////////////////////////
            try
            {
              AirlineInfo objAirlineInfo;
              IAirportFacilitiesInfo objFacilitiesInfo;

               // Create a new AirlineInfo object.
               objAirlineInfo = new AirlineInfo();

               // Invoke the GetAirlineTiming method.
               String strDetails = objAirlineInfo.GetAirlineTiming
                                       (strAirline);

               // QI for the IAirportFacilitiesInfo interface.
               objFacilitiesInfo =
                     (IAirportFacilitiesInfo)objAirlineInfo;

               //Invoke a method on the IAirportFacilitiesInfo
               //interface.
               Console.WriteLine("{0}",
               objFacilitiesInfo.GetInternetCafeLocations());
            }
            catch(InvalidCastException eCast)
            {
               Console.WriteLine("We got an InvalidCast Exception " +
                              "- Message is {0}",eCast.Message);
            }
        }
    }
```

17.3.6 后期绑定 COM 组件

到目前为止，您已经见到了两个演示——AirlineClientlApp 和 AirlineClient2APP，这两个演示都使用 RCW 元数据把.NET 客户端早期绑定到 COM 组件上。虽然早期绑定提供了各种优点——例如在编译时进行强类型检查、为开发工具自动完成类型信息(例如 Visual Studio .NET)，当然还有更佳的性能——但是，还可能出现这样的情况，在编译时还没有元数据可以用于要绑定到的 COM 组件上，因此，您需要对组件进行后期绑定。例如，如果您试图使用的组件只有一个 dispinterface，那么就会被限制为进行后期绑定来使用组件。

您可以通过在第16章中已经学过的反射机制来实现对COM对象的后期绑定。要想以这种方式绑定到一个COM组件上，您需要知道组件的ProgID。这是因为类System.Activator的静态方法CreateInstance需要一个Type对象。然而，使用组件的ProgID，您就可以调用System.Type类的GetTypeFromProgID方法。该方法返回一个有效的.NET Type对象，然后您就可以在调用System.Activator.CreateInstance方法时使用该Type对象。一旦完成该步骤，您就可以用Type对象的System.Type.InvokeMember实例来运用任何由组件默认接口支持的方法或属性，而Type对象是从GetTypeFromProgID中取得的。

您所需要知道的就是方法或属性名，以及方法调用时接收的参数类型。当您调用了某个被后期绑定组件的方法时，传递参数的方法是把这些参数捆绑放入一个通用的System.Object数组中，然后把该数组传递给该方法。根据您使用的是方法还是获取/设置某个属性值，您还要设定合适的绑定标志(binding flags)。

正如您在下面代码中见到的那样，这里要做的工作要比早期绑定稍多一些。但是，有时候后期绑定是惟一的选择，那时您会非常高兴拥有它。

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient3App
{
    public static void Main()
    {
        ////////////////////////////////////////////////
        /// LATE BINDING
        ////////////////////////////////////////////////
        try
        {
            object objAirlineLateBound;
            Type objTypeAirline;

            object[] arrayInputParams= { "Air Scooby IC 5678" };

            objTypeAirline = Type.GetTypeFromProgID
                   ("AirlineInformation.AirlineInfo");

            objAirlineLateBound = Activator.CreateInstance
                                 (objTypeAirline);

            String str = (String)objTypeAirline.InvokeMember
                                ("GetAirlineTiming",
```

```
                                BindingFlags.Default |
                                BindingFlags.InvokeMethod,
                                null, objAirlineLateBound,
                                arrayInputParams);

            Console.WriteLine("{0}",str);

            String strTime = (String)objTypeAirline.InvokeMember
                                ("LocalTimeAtOrlando",
                                BindingFlags.Default |
                                BindingFlags.GetProperty,
                                null, objAirlineLateBound,
                                new object [] {});

            Console.WriteLine ("Hi there !. The Local Time in " +
                               "Orlando,Florida is: {0}", strTime);
        }
        catch(COMException e)
        {
            Console.WriteLine("Oops- We encountered an error " +
                              "for Airline {0}. The Error message " +
                              "is : {1}. The Error code is {2}",
                              strFoodJunkieAirline,
                              e.Message,e.ErrorCode);
        }
    }
}
```

17.3.7 COM 线程模型

当大部分人开始 COM 编程时，他们对 COM 线程模块和单元几乎都是一知半解。直到这些人变得越来越有经验之后，他们才开始意识到，当一个单线程单元(single-threaded apartment，STA)客户端线程被用来创建一个多线程单元(multithreaded apartment，MTA)时，他们一直在使用的自由线程模型增大了服务器性能耗费。此外，刚刚接触 COM 的程序员往往会忽视线程安全，以及当并发线程访问他们的 COM 组件时所面临的威胁。

在线程能够调用一个 COM 对象以前，线程必须通过声明它将进入 STA 还是 MTA，这样就声明了它与一个单元的关系。STA 客户端线程调用 CoInitialize(NULL)或 CoInitializeEx(0, COINIT_APARTMENTTHREADED)来进入 STA，而 MTA 线程调用 CoInitializeEx(0, COINIT_MULTITHREADED)来进入一个 MTA。与之相似的是，在.NET 受控世界中，您可以选择允许在受控空间中进行调用的线程声明它的单元关系。默认地，

在受控程序中进行调用的线程选择存在于一个 MTA 中。这就仿佛是进行调用的线程用 CoInitializeEx(0,COINIT_MULTITHREADED)对自身进行初始化。但是还是考虑一下，如果它调用一个典型的 STA COM 组件，而该组件被设计为线程单元，这样做所导致的开销和性能降低。性质相反的单元会导致一个额外的代理/存根(proxy/stub)对开销，而这当然会导致性能的降低。

为实现该目的，您可以通过使用类 System.Threading.Thread 的 ApartmentState 属性来覆盖.NET 应用程序中一个受控线程的默认单元选择。属性 ApartmentState 采用了下列枚举值中的一个：

- **MTA** 多线程单元
- **STA** 单线程单元
- **Unknown** 等价于默认的 MTA 行为

在您对 COM 对象进行任何调用之前，也需要为进行调用的线程指定属性 ApartmentState。请注意，一旦 COM 对象被创建了，就不可能改变 ApartmentState。因此，在代码中尽可能早地设置线程的 ApartmentState 设置就很有意义了。下面的代码显示了应该怎样做：

```
// Set the client thread ApartmentState to enter an STA.
Thread.CurrentThread.ApartmentState =
   ApartmentState.STA;

// Create our COM object through the Interop.
MySTA objSTA = new MySTA();
objSTA.MyMethod()
```

17.4 本章小结

最后需要说明的是，和遗留代码——Pinvoke、非安全代码和 COM Interop——一起工作的不同机制是如何适合于和.NET 相关的总体模式的。在这一章里，您学到了以下知识：

- 关于使用标准的类 C 语言函数调用，您学到了如何同几个特性一起使用 Pinvoke，从而使得汇集不同数据类型——包括自定义数据——更为简单。
- 关于非安全代码，您了解了如何放弃 C#应用程序中受控代码的优点，而代之以得到更多的控制权。这些情形包括：需要人工操纵内存以便于提高效率，或是把代码块移到 C#程序中，且不仅是简单地把代码转换成受控代码。
- 关于 COM，您已经了解了如何将传统 COM 组件应用于.NET 程序，以及 COM Interop 如何“无缝地”允许您从受控代码处重用现有的 COM 组件。然后，您浏览了利用早期绑定和后期绑定来使用 COM 组件的方法，以及进行运行时刻类型

检查的途径。最后，您学习了当 COM 组件被运用时，受控线程是如何声明其单元关系的。

在这里，作为一个使用其中一种机制来处理非受控代码的开发人员，您可能想知道继续使用这些方法是否还有意义，或者是否应该用一种.NET 语言——例如 C#——把所有的组件和商业逻辑代码写成受控组件，从而直接转向.NET 世界。其答案取决于当前的情况。

如果您有许多遗留代码——无论是 DLL 中的类 C 语言函数、直接处理内存的代码、COM 组件，还是三者的联合——事实是，您有可能无法在一夜之间转换所有代码。在这种情况下，利用不同的.NET 机制来与遗留代码一起工作就十分有意义了。然而，如果您是从头开始写一个新的商业逻辑代码，那么强烈建议您利用诸如 C#之类的语言，把代码写成受控组件。这样，您就可以避免由于在受控和非受控边界之间转换而导致的性能降低状况。

第 18 章　使用组合体

本章将介绍使用组合体的主要优点，包括.NET 组件的打包和版本管理。在这一章里，您也将学习到如何利用组合体生成(Assembly Generation)工具(al.exe)来创建单文件组合体和多文件组合体，如何利用强名字(Strong Name)工具(sn.exe)来创建共享组合体，如何利用组合体缓存区浏览器外壳扩展(Assembly Cache Viewer shell extension)(shfusion.dll)浏览全局组合体缓存区，以及如何用全局组合体缓存(Global Assembly Cache)工具(gacutil.exe)组合体缓存区。最后，我们将通过一些演示，看一看还有什么关于版本管理的注意事项，并且了解组合体和.NET 版本管理策略是如何帮助我们避免所谓的“DLL 地狱”的。

18.1　组合体概览

第 16 章将组合体描述成一种物理文件，这种文件由通过.NET 编译器生成的一个或多个可移植的可执行(PE)文件组成。在第 16 章中，这一定义是可以接受的。可是，组合体比这要复杂得多。以下是一个更完整的定义：组合体是对清单、一个或多个模块以及可选的一种或多种资源的打包。为了部署、版本管理和维护，可以利用组合体，从语义上将功能性单元组成一个单独的文件。

所有使用.NET 运行环境的 PE 文件都含有一个或一组组合体。当您用 C#编译器编译一个应用程序时，您也实际上正在生成一个组合体。除非您正在试图将多个模块放到单个组合体中，或者想利用一些特定于组合体的特性，如版本管理等，否则您也许还没认识到这一事实。然而，认识到这一点非常重要，也即，在任何时候，当您(使用/t:library 选项)建立一个 EXE(可执行文件)或 DLL(动态链接库)时，您也正在生成一个组合体，该组合体带有一张向.NET 运行环境描述组合体的清单。另外，您也可以(用/t:module 选项)创建一个模块，该模块是一个没有附带清单的真正的 DLL(带有一个.netmodule 的扩展名)。换句话说，虽然从逻辑上讲它仍然是一个 DLL，但它不属于某一个组合体，并且，当编译一个应用程序时，必须用/addmodule 选项将它加到组合体中，或者用组合体生成工具将它加到组合体中。等我们讲到了“建立组合体”那一部分时，您将看到如何完成这一类工作。

清单数据

组合体的清单可以以不同的方式存储。如果要编译一个独立的应用程序或 DLL，需要将清单合并到作为结果的 PE 中。这种组合体被称为**单文件组合体**。也可以生成带有清单的**多文件组合体**，这里的清单既可以作为组合体中的一个独立实体存在，也可以作为组合体中一个模块的附件存在。

此外，组合体的定义在很大程度上取决于您如何使用它。从客户端的角度来看，组合体是模块、导出类型和可选资源的命名并经过版本管理的集合。而从组合体创建者的观点来看，组合体是将相关的模块、类型和资源打包，并导出供客户端使用部分的一种手段。我们曾经说过，组合体是这样一种清单，它在组合体的实现细节和客户端想要用的部分之间提供间接级别。以下是存储在组合体清单里的分析说明：

- **组合体名** 就是组合体的文字名称。
- **版本管理信息** 这个字符串由四个不同的部分组成：主版本号和次版本号，修正版号和创建号。它们组成一个版本号。
- **一个(可选的)共享名和签名组合体散列** 这一信息与组合体的部署有关，在本章后面的“组合体的部署”一节将加以介绍。
- **文件** 这一列表包含存在于组合体中的所有文件。
- **被引用的组合体** 这是一个所有外部组合体的列表，这些组合体被直接从清单的组合体里引用。
- **类型** 这是一个组合体里所有类型的列表，它可以映射到含有类型的模块上。这些数据有助于第 16 章中反射实例(它迭代一个组合体中所有的类型)快速地执行。
- **安全权限** 这是一个被组合体显式拒绝的安全权限的列表。
- **自定义特性** 第 8 章描述了自定义特性的创建。和类型一样，为了在反射期间能更快地对其进行访问，自定义特性被储存在组合体清单里。
- **产品信息** 这些信息包括公司、商标、产品和版权值等。

18.2 组合体的优点

组合体为开发人员提供了许多优点，包括打包、部署和版本管理。

18.2.1 组合体的打包

将多个模块打包成一个单一物理文件，这种功能的优点之一就是可以改善性能。当您创建一个应用程序，并用多文件组合体将它部署时，.NET 运行环境所需加载的仅仅是必需的模块。这样就会减小应用程序的工作集。

18.2.2 组合体的部署

组合体是在.NET 中进行部署的最小单位。正如前面所提到的，您可以用/t:module 选项创建一个.netmodule 模块，但如果您想部署它，就必须将那个模块加入到组合体中。另外，虽然从某种意义上讲，组合体是应用程序部署的一种手段，但从技术角度上讲，这句话是不正确的。将.NET 中的组合体看作是“类部署”(很像 Win32 中的 DLL)的一种形式会更准确些，在这种形式里，单个的应用程序可由许多个组合体组成。

因为组合体是自描述的，所以部署它们的最简单方法是，将组合体复制到我们需要的目标文件夹中。然后，当您试图运行一个包含于组合体中的应用程序时，清单将引导.NET 运行环境找到组合体中所含有的模块。此外，组合体也含有对任何外部组合体的引用，应用程序可能会需要用到这些外部组合体。

最常用的部署方式是私有组合体(private assembly)——也就是说，组合体被复制到文件夹而且不进行共享。那么，如何指定一个私有组合体呢？其实这是默认并自动发生的，除非您将组合体显式设为共享组合体(shared assembly)。将组合体设为共享需要多做一些工作，在后面的“创建共享组合体”一节将会涉及到这一内容。

18.2.3 组合体的版本管理

使用组合体的另一大优点是内置版本管理——特别是“DLL 地狱”的终结。“DLL 地狱”指的是这样一种情况，一个应用程序覆盖了另一个应用程序所需要的 DLL(动态链接库)，通常指的是同一 DLL 的早期版本，这样就破坏了第一个应用程序。虽然 Win32 资源文件格式考虑到了版本管理资源类型，但操作系统并没有执行任何版本管理规则，以便于相关的应用程序仍可以正常运行。这完全是程序员的责任。

作为解决这一问题的一种手段，清单包括了用于组合体的版本管理信息、所有被引用的组合体列表和用于那些被引用组合体的版本管理信息。由于采用了这种结构，.NET 运行环境能够保证版本管理策略得到支持，保证应用程序将继续起作用，即使在系统上安装有共享 DLL 的更新的、不兼容的版本。因为版本管理是组合体的最大优点之一，所以在“版

本管理组合体”一节中将通过介绍一些实例对它进行深层次的探讨。

18.3 建立组合体

一旦您用 / t:library 选项创建了一个 DLL，您就不能再将它添加到另一个组合体里。这是因为程序编译器自动为 DLL 生成了一张清单，由此可见，DLL 本身就是一个组合体。为了从实际上理解这一点，请看下面的例子。例子中给出的是一个 DLL(Module1Server.cs)，它有一个模拟类型，叫 Module1Server。

```
//Module1Server.cs
//build with the following command lineswitches
//csc /t:library Module1Server.cs
public class Module1Server
{
}
```

然后这个 DLL 被下面的客户端代码(Module1Client.cs)引用：

```
// Module1ClientApp.cs
// build with the following command line switches
//    csc Module1ClientApp.cs /r:Module1Server.dll
using System;
using System.Diagnostics;
using System.Reflection;

class Module1ClientApp
{
    public static void Main()
    {
        Assembly                    DLLAssembly                    =
Assembly.GetAssembly(typeof(Module1Server));
        Console.WriteLine("Module1Server.dll Assembly Information");
        Console.WriteLine("\t" + DLLAssembly);

        Process p = Process.GetCurrentProcess();
        string AssemblyName = p.ProcessName + ".exe";
        Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
        Console.WriteLine("Module1Client.exe Assembly Information");
        Console.WriteLine("\t" + ThisAssembly);
    }
}
```

现在，假设您用以下的选项建立了这两个模块：

```
csc /t:library Module1Server.cs
csc Module1ClientApp.cs /r:Module1Server.dll
```

此时运行代码，将得到如下的输出：

```
Module1Server.dll Assembly Information
Module1Server,Version=0.0.0.0,Culture=neutral,PublicKeyToken=null
Module1Client.dll Assembly Information
Module1Client,Version=0.0.0.0,Culture=neutral,PublicKeyToken=null
```

这就证明了，EXE 和 DLL 两者都存在于它们各自的不同组合体里。

其实，如果您改变 Module1Server 类的访问限定符，将 public 改为 internal，客户端的代码就无法编译了。因为由定义可知 internal 访问限定符具体指明：被修改的类型只对同一组合体中的其他代码是可访问的。

创建含有多模块的组合体

您可以以两种方式将例子中的两个模块放到同一组合体里。第一种方式是改变与编译器一起使用的选项。请看下面的例子：

```
//Module2Server.cs
//build with the following command lineswitches
//csc /t:module Module2Server.cs
internal class Module2Server
{
}
```

注意，在此我们使用了 internal 访问限定符，这样就只有组合体内部的代码可以访问类。

```
// Module2ClientApp.cs
// build with the following command line switches
//     csc /addmodule:Module2Server.netmodule Module2ClientApp.cs
using System;
using System.Diagnostics;
using System.Reflection;

class Module2ClientApp
{
    public static void Main()
    {
        Assembly DLLAssembly =
            Assembly.GetAssembly(typeof(Module2Server));
```

```
        Console.WriteLine("Module1Server.dll Assembly Information");
        Console.WriteLine("\t" + DLLAssembly);

        Process p = Process.GetCurrentProcess();
        string AssemblyName = p.ProcessName + ".exe";
        Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
        Console.WriteLine("Module1Client.dll Assembly Information");
        Console.WriteLine("\t" + ThisAssembly);
    }
}
```

请注意 Module2Server.cs 和 Module2Client.exe 是如何建立的：

```
csc /t:module Module2Server.cs
csc /addmodule:Module2Server.netmodule Module2Client.cs
```

首先，您必须去掉/r 选项，因为这一选项只是用来引用组合体的，而现在两个模块都将存在于同一组合体里。然后，您必须插入/addmodule 选项，该选项用于告诉编译器要将哪些模块加到正在创建的组合体中。

现在，编译并运行这个应用程序会得到如下结果：

```
Module1Server.dll Assembly Information
Module2Client,Version=0.0.0.0,Culture=neutral,PublicKeyToken=null
Module1Client.dll Assembly Information
Module2Client,Version=0.0.0.0,Culture=neutral,PublicKeyToken=null
```

另一种创建组合体的方式是用组合体生成工具。该工具可以利用一个或多个输入文件，这些文件既可以是.NET 模块(含有 MSIL)，也可以是资源文件和图像文件。它输出的是一个带有组合体清单的文件。例如，如果您拥有好几个 DLL，并且想将它们作为一个单一的单元来分布和进行版本管理，那么您就可以使用组合体生成工具。假设您的 DLL 是以 A.DLL、B.DLL.和 C.DLL 命名的，使用 al.exe 应用程序就可以创建一个如下的复合组合体：

```
al /out:COMPOSITE.DLL A.DLL B.DLL C.DLL
```

18.4 创建共享组合体

当一个组合体要和多个应用程序一起使用时，要使用共享组合体，并且版本管理也很重要（我们将在下一部分学习版本管理）。为了共享一个组合体，您必须用与.NET SDK 相配套的强名字工具为组合体建立一个**共享名**(也叫**强名字**)。以下列出的是使用强名字的 4 个主要优点：

- 这是.NET 中的一种机制，用于生成一个全局惟一的名字。

- 因为已生成的密钥对(我们曾简短地解释过)含有一个签名，所以您可以对照创建时的原始状态来判断它是否被篡改了。
- 强名字保证第三方无法发布已建组合体的后继版本。这也是因为签名——第三方没有您的私钥。
- 当.NET 加载一个组合体时，运行环境能够检查组合体是否来自于调用者所期望的发布者。

创建强名字的第一步是，用强名字工具为组合体新建一个密钥文件。这可以通过在含有密钥的输出文件名前面加上指定的-K 选项来完成。在此，我们仅仅随便虚拟一个文件名——InsideCSharp.key，并创建如下文件：

```
sn -k InsideCSharp.key
```

运行它之后，您将得到的确认消息如下所示：

```
Key pair written to InsideCSharp.key
```

现在，将 Assembly: AssemblyKeyFile 特性添加到源文件里。在此已创建了另一个简单的文件集，以它为例说明这是如何被完成的：

```
// Module3Server.cs
// build with the following command line switches
//    csc /t:module Module3Server.cs
internal class Module3Server
{
}

// Module3ClientApp.cs
// build with the following command line switches
//    csc /addmodule:Module3Server.netmodule Module3ClientApp.cs
using System;
using System.Diagnostics;
using System.Reflection;

[assembly:AssemblyKeyFile("InsideCSharp.key")]

class Module3ClientApp
{
    public static void Main()
    {
        Assembly DLLAssembly =
            Assembly.GetAssembly(typeof(Module3Server));
        Console.WriteLine("Module1Server.dll Assembly Information");
        Console.WriteLine("\t" + DLLAssembly);
```

```
            Process p = Process.GetCurrentProcess();
            string AssemblyName = p.ProcessName + ".exe";
            Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
            Console.WriteLine("Module1Client.dll Assembly Information");
            Console.WriteLine("\t" + ThisAssembly);
        }
    }
```

正如您所看到的，Assembly: AssemblyKeyFile 特性的构造函数采用了密钥文件的名字，该密钥文件是由强名字工具生成，并且通过这个构造函数，您可以指定一个密钥对，用于赋给组合体一个强名字。更重要的是要理解这一点，即这一特性是一个组合体级的特性。因此，从技术上讲，它可以被放到组合体的任何文件里，并且不附属于任何一个特定类。然而，我们还是习惯于把这一特性放在 using 语句之后，任何类的定义之前。

现在，当您运行那个应用程序时，要注意组合体的 PublicKeyToken 值。在前两个例子中，这个值是 null，因为那些组合体是作为私有组合体来考虑的。可是现在，这个组合体已被定义为共享组合体，于是组合体就有了一个与之关联的密钥令牌。

```
Module3Server.dll Assembly Information
    Module3Client, Version=0.0.0.0, Culture=neutral,
    PublicKeyToken=6ed7cef0c0065911
Module3Client.dll Assembly Information
    Module3Client, Version=0.0.0.0, Culture=neutral,
    PublicKeyToken=6ed7cef0c0065911
```

根据我们在这个演示组合体中实例化的 Assembly 对象可知，它是共享的。可是，我们怎样才能知道，在.NET 系统中哪些组合体是共享的呢？答案是全局组合体缓存区。下一部分中将介绍.NET 的有关内容，并且解释全局组合体缓存区在共享组合体中所起的作用。

18.5 使用全局组合体缓存区

每一个.NET 安装都有一个叫做**全局组合体**缓存区的代码缓存区。这个区域主要为了实现三个目的：

- 它被用来储存从 Internet 或其他服务器(http 和文件服务器)上下载的代码。注意，用于特定应用程序的下载代码被储存在缓存区的私有部分里——这就防止了它被其他代码访问。
- 它是一个数据存储库，服务于被多个.NET 应用程序共享的组件。用全局组合体缓存区工具安装到缓存区上的组合体，被储存在缓存区的全局部分里，并且它可被机器上的所有应用程序访问。

- 那么，被同步编译执行(jitted)的代码到底存在哪里，以使 C#代码只在它第一次执行时才被同步编译呢?答案是：经过预先编译(prejitted)的组合体本地代码版本被存储在缓存区里。

查看缓存区

我们来看一看缓存区，以了解当前安装和共享的组合体。利用 Microsoft Explorer 打开 c:\winnt\assembly 文件夹。为了有助于查看组合体的有关信息，.NET 提供一个外壳扩展特性，叫做组合体缓存区浏览器(Assembly Cache Viewer)(shfusion.dll)。这个工具使您能够看到组合体的有关信息，如版本号、风格、公用密钥令牌，甚至还可以查看组合体是否经过预先编译。

查看缓存区的另一种途径是使用全局组合体缓存区工具。这一工具使您能够通过(互斥地)指定下面任何一个命令行，完成一些基本的任务。

- *i* 这个标志将一个组合体安装到全局组合体缓存区里。下面是一个例子：

```
gacutil -i HelloWorld.DLL
```

简而言之，您将看到如何利用这个选项将 Module3Client 组合体添加到缓存区。

- *-u* 这个标志从全局组合体缓存区中卸载一个组合体，包括任何版本信息。如果您不具体指定版本信息，带有特定名的所有组合体都将被去掉。因此，这里的第一个例子，不论版本号如何，都将卸载所有 Hello World 组合体，而第二个例子只卸载特定的版本：

```
gacutil -u HelloWorld
gacutil -u HelloWorld,ver=1,0,0,0
```

- *-l* 这个标志列出了全局组合体缓存区的内容，包括组合体名、它的版本号、它的地址和它的共享名。

注意： 在一些早期的.NET beta 版本中，存在这样的一个问题，当打开 C:\winnt\assembly 文件夹时，外壳扩展没有执行。这是由于 shfusion.dll 扩展没有被正确地注册而造成的。一旦在您的系统里这种情况发生，利用命令行方式，从 C:\winnt\Microsoft.net\framework\v×××文件夹输入下面的内容，这里的×××代表您正运行的.NET 框架的版本号。很明显，因为是用一个 beta 版进行工作，所以文件夹名称将在.NET 正式出台之前改变。搜索 shfusion.dll 文件，并且使用那个文件夹。这里使用了这个文件夹来表示目前使用的.NET:

```
c:\winnt\microsoft.net\framework\v1.0.2615>regsvr32 shfusion.dll
```

在已经创建了一个公钥文件，并且把它赋给一个组合体之后，让我们将那个组合体添加到缓存区中。为了完成这项工作，要以命令行方式输入下面的内容：

```
gacutil -i Module3ClientApp.exe
```

如果一切顺利，您将得到如下的确认信息：

```
Assembly successfully added to the cache
```

在这里，您可以利用 gacutil-l 命令来查看在缓存区上列出的组合体，并找到 Module3Client，或者您也可以使用组合体缓存区浏览器。我们使用后者。如果您在 Windows Explorer 中打开了缓存区(C:\Winnt\Assembly 或 C:\Windows\Assembly)，现在您就将看到 Module3Client 组合体，它和其他组合体列在一起。右击它，选择“属性”，您就会看到诸如公钥值、版本号和组合体在硬盘上的物理地址之类的信息。需要附加说明的一点是，公钥可能会不同，但重要的一点是，它和通过执行 Module3ClientApp 应用程序而显示出来的密钥一样。

18.6 组合体的版本管理

组合体清单含有一个版本号和一张所有被引用组合体的列表，还有与这些被引用程序相关的版本信息。您很快就将看到，版本号被分为 4 部分，并采用下面的格式：

```
<major><minor><build><revision>
```

版本号工作的方式是，在运行时刻，.NET 利用该版本号信息来决定应用程序使用哪一特定的组合体版本。正如您马上将看到的一样，默认行为(称为“版本管理策略”)是，一旦安装了一个应用程序，如果版本与<major><minor>相匹配，.NET 就会自动使用被应用程序引用的组合体的最近使用版本。您可以利用部署文件来改变该默认行为。

版本管理仅适用于共享组合体——私有组合体并不需要——并且可能是决定创建和共享组合体时的一个最为重要的因素。因此，为了展示所有这些是如何工作的，以及如何使用组合体的版本管理，让我们看一看几个例子的代码。

因为本书的重点主要放在版本管理和运行环境为了执行它所做的工作，所以这里将采用的例子是版本管理实例的一个更简单版本，它与.NET SDK 配套，并且不包含 Windows Form 的资料。

此处用两个可执行文件，代表两个账号包，它们分别叫做 Personal 和 Business。这两个应用程序都使用了一个叫 Account 的通用共享组合体。Account 类具有的惟一功能是宣布它的版本，这样我们就能够确保，我们的应用程序正在使用所需的 Account 类版本。为了这一目的，演示将包括 Account 类的多个版本，这样您就能够自己发现版本管理是如何用

默认版本管理策略来工作的，以及 XML 是如何被用于在应用程序和组合体的特定版本之间创建一个关联的。

开始的时候，创建一个叫 Accounting 的文件夹。在这个文件夹里，创建一个密钥对，它将被 Account 类的所有版本使用。为了完成这项工作，在 Accounting 文件夹的命令行里键入：

```
sn /k account.key
```

创建了那个密钥对之后，在 Accounting 文件夹里创建一个叫做 Personal 的文件夹。在 Personal 文件夹里创建一个叫做 Personal.cs 的文件，如下所示：

```
// Accounting\Personal\Personal.cs
using System;

class PersonalAccounting
{
    public static void Main()
    {
        Console.WriteLine
            ("PersonalAccounting calling Account.PrintVersion");
        Account.PrintVersion();
    }
}
```

在同一个 Personal 文件夹里创建一个叫 Account1000 的新文件夹。这个文件夹将保存这个演示的 Account 类的第一个版本。完成这些工作之后，在 Account1000 文件夹里创建下面的文件(Accout.cs)：

```
// Accounting\Personal\Account1000\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("..\\..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.0")]
public class Account
{
    public static void PrintVersion()
    {
        Console.WriteLine
            ("This is version 1.0.0.0 of the Account class");
    }
}
```

正如您所看到的，此处使用 AssemblyKeyFile 和 AssemblyVersion 特性，将 C#程序编译器指向早先创建的密钥对，并且显式地指定 Account 类的版本。现在，建立如下的 Account DLL：

```
csc /t:library account.cs
```

一旦创建了 Account 类，就需要将它添加到全局组合体缓存区：

```
gacutil -i Account.dll
```

如果需要，我们可以检查 Account 组合体是否真的是存在组合体缓存区里。现在，转到 Personal 文件夹，并且建立下面的应用程序：

```
csc Personal.cs /r:Account1000 \Account.dll
```

最后，运行该应用程序，就会输出：

```
PersonalAccounting calling Account.PrintVersion
This is version 1.0.0.0 of the Account class
```

到目前为止，我们并没有进行什么新的工作。尽管如此，还是让我们看一看，当安装了另一个使用 Account 类的更新版本的应用程序以后，将出现什么情况。

在 Accounting 文件夹里创建一个叫做 Business 的文件夹。在 Busines 文件夹里创建一个叫 Account1001 的文件夹，用它来代表 Account 类的一个新版本。那个类被收藏在一个叫做 Account.cs 的文件里，它看起来与以前的版本几乎相同。

```
// Accounting\Business\Account1001\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("..\\..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.1")]
public class Account
{
    public static void PrintVersion()
    {
        Console.WriteLine
            ("This is version 1.0.0.1 of the Account class");
    }
}
```

像以前一样，利用下面的命令建立 Account 类的版本：

```
csc /t:library Account.cs
gacutil -i Account.dll
```

在这里，您应该会在全局组合体缓存区里看到 Account 类的两个版本。现在，创建如下的 Business.cs 文件(在 Accounting\Businessfolder 里)：

```
// Accounting\Business\Business.cs
using System;
```

```
class PersonalAccounting
{
    public static void Main()
    {
        Console.WriteLine
            ("BusinessAccounting calling Account.PrintVersion");
        Account.PrintVersion();
    }
}
```

用下面的命令建立 Business 应用程序：

```
csc business.cs /r:Account1001 \Account.dll
```

运行这个应用程序，将得到下面的信息：

```
BusinessAccounting calling Account.PrintVersion
This is version 1.0.0.1 of the Account class
```

信息证实了这一事实，即 Business 应用程序正在使用 Account 组合体的 1.0.0.1 版本。现在再运行 Personal 应用程序，看看将发生什么情况。

```
PersonalAccounting calling Account.PrintVersion
This is version 1.0.0.1 of the Account class
```

Personal 和 Business 两个应用程序都在使用 Account 组合体的最新版本。为什么？这与所谓的快速修复引擎(Quick Fix Engineering，QFE)更新和.NET 默认版本管理策略有关。

18.6.1 QFE 和默认版本管理策略

快速修复引擎更新，或者叫“热修复”，是一种计划外的修复，它们被发送出去以找到主要问题。因为热修复通常不修改代码的接口，所以客户端代码受到负面影响的可能微乎其微。因此，默认版本管理策略要自动使所有客户端代码与代码的新“修复”版本相关联，除非存在一个用于应用程序的部署文件，该文件显式地把应用程序与组合体的特定版本关联起来。如果版本号里惟一被改变的部分是 revision 部分，那么组合体的一个新版本号就可看作是一个 QFE。

18.6.2 创建安全模式部署文件

也许在大部分时间里默认版本管理策略都运行良好，但如果您想要指定 Personal 程序只运行与它配套的版本，您该怎么办？这就是 XML 部署文件应该出现的地方。这些文件具有与应用程序一样的名字，并且驻留于相同的文件夹里。当应用程序运行时，就会读取

这些文件，然后.NET 利用那里的 XML 标记去指定使用组合体的哪一个版本。

要想指定应用程序总是使用与它配套的那个组合体版本，您需要把应用程序的绑定模式指定为“safe”。有时这也通俗地指将应用程序放到安全模式中。为了检验，我们可以在 Accounting/Personal 文件夹里创建一个叫做 PersonalAccounting.cfg 的文件，并且像下面那样去修改它。注意<AppBindingMode>标记。

```
<?xml version ="1.0 "?>
<Configuration>
<BindingMode>
<AppBindingMode Mode="safe"/>
</BindingMode>
</Configuration>
```

现在，如果您再次运行 Personal 应用程序，您将看到下面的输出：

```
PersonalAccounting calling Account.PrintVersion
This is version 1.0.0.0 of the Account class
```

18.6.3 使用组合体的特定版本

现在让我们看看与版本管理有关的另一个常见事项。在这个演示的最后一部分，我们将把一个错误引入到等式中。在 Account\Business 文件夹里创建一个名叫 Account1002 的文件夹，在这个文件夹里接着创建下面的 Account 类。注意到，这一次我们特地编制了 AccountPrint 版本，用于引发一个异常来对类 Account 模拟非工作更新。

```
// Accounting\Business\Account1002\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("..\\..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.2")]
public class Account
{
    public static void PrintVersion()
    {
        // This is purposely thrown to simulate code failure.
        throw new Exception();

        Console.WriteLine
            ("This is version 1.0.0.2 of the Account class");
    }
}
```

现在，像下面一样建立 Account 组合体的这种版本：

```
csc /t:library Account.cs
gacutil -i Account.dll
```

运行 Personal 或者 Business 应用程序，将会产生截获不到的异常，它会引起应用程序中止。这里在软件部署中模拟了一种常见现象——安装某些并不起作用的东西，以及停止中断其他应用程序。在 Personal 应用程序中，我们不想转换为安全模型，因为我们想运行 Account 的最新工作版本(1.0.0.1 版本)，而安全模式使我们仍用最初版本(1.0.0.0)工作。

答案仍存在于应用程序的部署文件中。现在，修改 Accounting\PersonalAccounting.cfg，使得它的 XML 标记如下所示：

```
<?xml version ="1.0"?>
<Configuration>
<BindingPolicy>
<BindingRedir Name="Account"
              Originator="32ab35a4550339b1"
              Version="*"
              VersionNew="1.0.0.1"
              UseLatestBuildRevision="no"/>
</BindingPolicy>
</Configuration>
```

要注意的是，指定用于 Originator 特性的密钥不是通用的，当您创建 Account.Key 文件时，您需要用生成的那个密钥来代替文中的密钥。您可以打开组合体缓存区并找到 Account 类，这样就找到所需的值了。

最后，再次运行 Personal 应用程序，您将看到它不再存在什么问题，而它使用的 Account 类正是我们需要的版本(1.0.0.1)。

18.7 本章小结

本章讨论了使用组合体的主要优点，包括打包、部署、版本管理和安全性，也讨论了单文件组合体、多文件组合体和组合体生成工具(al.exe)，以及如何利用强名字工具(sn.exe)创建和共享组合体，还有全局组合体缓存区工具(gacutil.exe)。另外也通过创建基于 XML 的部署文件，示范了如何控制.NET 所使用的默认版本管理策略。