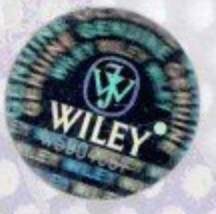
Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Beginning Visual C# 2010

C#入门经典

（第5版）



(美) Karli Watson 等著
Christian Nagel
齐立波 翻译
黄 静 审校

清华大学出版社

全面讲解C# 2010和.NET架构编程知识
为您编写卓越C# 2010程序奠定坚实基础

C#入门经典系列是屡获殊荣的C#名著和超级畅销书。最新版的《C#入门经典(第5版)》全面讲解C# 2010基础知识，浓墨重彩地描述Web和Windows编程以及数据访问(数据库和XML)等内容，详细介绍C#编程工具以及Visual Studio 2010中的Visual C# 2010开发环境。贯穿全书的分步说明和极富启迪意义的示例指引您使用高效C# 2010代码得心应手地编写程序。

本书内容

- 解释变量和表达式等基本C# 2010语法知识
- 介绍泛型的含义和用法
- 讨论Windows编程和Windows窗体
- 介绍C#改进内容、lambda表达式和扩展方法
- 解释Windows应用程序部署方法
- 讨论XML并简要介绍LINQ
- 深入探讨调试和错误处理方法
- 演示有效WPF和WCF技术

Karli Watson是Infusion Development 公司高级顾问，并担任Boost.net的技术架构师和IT自由撰稿人、作家和开发人员。他曾编著多本.NET(尤其是C#)书籍，极擅长以浅显易懂的方式阐明复杂技术主题。

Christian Nagel是微软技术代言人、微软MVP，拥有逾25年的软件开发经验。Christian熟悉各种语言和平台，曾编写多本.NET图书，并多次在国际会议上发表重要演讲。

Wrox Beginning guides are crafted to make learning programming languages and technologies easier than you think, providing a structured, tutorial format that will guide you through all the techniques involved.

源代码下载及技术支持

http://www.wrox.com

http://www.tupwk.com.cn/downpage

wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

Wrox™
An Imprint of
WILEY

上架建议：编程语言/C#(.NET)
读者信箱：wkservice@vip.163.com
投稿信箱：bookservice@263.net

ISBN 978-7-302-24130-0
9 787302 241300 >
定价：99.80元

C#入门经典

(第5版)

(美) Karli Watson
Christian Nagel 等著
齐立波 翻译

清华大学出版社
北 京

Karli Watson，Christian Nagel，et al.
Beginning Visual C# 2010
EISBN：978-0-470-50226-6

北京市版权局著作权合同登记号 图字：01-2010-2530

图书在版编目(CIP)数据

C#入门经典(第 5 版)/ (美) 沃森(Watson, K.)，(美) 内格尔(Nagel, C.) 等著；齐立波 翻译；黄静 审校.
—北京：清华大学出版社，2010.12
书名原文：Beginning Visual C# 2010
ISBN 978-7-302-24130-0
Ⅰ. C… Ⅱ. ①沃… ②内… ③齐… ④黄… Ⅲ. C 语言—程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2010)第 210906 号

责任编辑：王 军 韩宏志
装帧设计：孔祥丰
责任校对：胡雁翎
责任印制：何 芊
出版发行：清华大学出版社　　地　　址：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　邮　　编：100084
社　总　机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185×260　印　张：56.75　字　数：1523 千字
版　　次：2010 年 12 月第 1 版　　印　　次：2010 年 12 月第1 次印刷
印　　数：1～5000
定　　价：99.80 元

产品编号：035690-01

作 者 简 介

Karli Watson 是 Infusion Development (www.infusion.com)的顾问，Boost.net (www.boost.net)的技术架构师和 IT 自由撰稿专业人士、作家和开发人员。他主攻.NET(尤其是 C#和后来的 WPF)，为几家出版商编写了多本围绕这个领域的图书。他擅长以便于任何有学习热情的人理解的方式阐述复杂的理念，并投入了大量时间研究新技术，找出可教给其他人的新东西。

在工作之余(这种时间似乎很少)，Karli 喜欢到山上滑雪，或者尝试发表他的小说。他喜欢穿颜色鲜亮的衣服，他的网址是 www.twitter.com/karlequin，也许有一天他自己会建立一个网站。Karli 编写了本书的 1～14、12、25 和 26 章。

Christian Nagel 是 Microsoft 区域总监、Microsoft MVP，是 Thinktecture 的合作伙伴，CN 创新技术的拥有者，他是一位软件架构师和开发人员，为开发 Microsoft .NET 解决方案提供培训和咨询服务。他拥有超过 25 年的软件开发经验。Christian 从 PDP 11 和 VAX/VMS 系统开始踏入其计算机生涯，此后接触了各种语言和平台。自从 2000 年以来，(那时.NET 还只是一个技术框架)他就开始使用各种.NET 技术建立大量的.NET 解决方案。他具备深厚的 Microsoft 技术功底，编写了大量.NET 图书，并获得了 Microsoft 认证培训师和专业开发人员的证书。Christian 在国际会议发表演讲，例如 echEd 和 Tech Days，并启动 INETA Europe 来支持.NET 用户组。可以通过网站 www.cninnovation.com 和 www.thinktecture.com 联系 Christian，在 www.twitter.com/christiannagel 上可以了解有关他的一些信息。Christian 编写了本书的 17～20 章。

Jacob Hammer Pedersen 是 Elbek & Vejrup 的一位资深应用程序开发人员，他刚能拼写 BASIC 时就开始了编程，BASIC 也是他使用的第一种编程语言。在 20 世纪 90 年代早期，他开始使用 Pascal 在 PC 上编程，不久就改用 C++，目前，他仍非常迷恋 C++。90 年代中期，他的兴趣又改变了，这次是 Visual Basic。2000 年夏，他发现了 C#，之后开始满心欢喜地研究这门语言。他主要工作在 Microsoft 平台上，其他的工作领域包括 MS Office 开发、SQL Server、COM 和 Visual Basic.Net。

Jacob 是丹麦人，工作生活在丹麦的奥尔胡斯市，他编写了本书的 15、16 和 22 章。

Jon D. Reid 是 Metrix LLC 的一位软件工程经理，Metrix LLC 是 Microsoft 环境的区域服务管理软件的 ISV。他与他人合作编写了各种.NET 图书，包括 *Beginning Visual C# 2008*、*Beginning C# Databases: From Novice to Professional*、*Pro Visual Studio .NET* 等，Jon 编写了本书的第 23 和 24 章。

Morgan Skinner 在校时就学习 Sinclair ZX80，开始了计算机生涯，当时他对教师编写的一些代码不感兴趣，便开始用汇编语言编程，从那时开始他使用了所有的语言和平台，包括 VAX 宏汇编、Pascal、Modula2、Smalltalk、X86 汇编语言、PowerBuilder、C/C++、VB 和目前的 C#，自从 2000 年发布 PDC 以来，他就用.NET 编程，而且非常喜欢.NET，所以在 2001 年加盟 Microsoft 公司，他现在是开发人员的主要支持人员，而且花费了大多数时间帮助客户使用 C#。Morgan 编写了本书的第 27 章。在 www.morganskinner.com 上可以联系到 Morgan。

技术编辑简介

Doug Holland 从 2007 年 3 月起担任英特尔公司的蓝带.NET 架构师和开发人员，是 Visual Computing Group 的成员，目前在高级工具和开发团队中工作，主要从事芯片集和驱动程序测试。Doug Holland 获得了牛津大学软件工程专业的硕士学位，已荣获 Microsoft MVP 和 Intel Black Belt Developer 奖。在工作之余，Doug 喜欢与妻子和 4 个孩子在一起享受快乐家庭生活，他还是 Civil Air Patrol/U.S. Air Force Auxiliary 的一位官员。除了构建和开发软件之外，Doug 还常常在加州的本地机场亲自驾驶 Cessnas 飞机在高空翱翔。

前　言

C#是 Microsoft 在 2000 年 7 月推出.NET Framework 的第 1 版时提供的一种全新语言。C#迅速流行开来，成为使用.NET Framework 的 Windows 和 Web 开发人员无可争议的选择。他们喜欢 C#的一个原因是其派生于 C/C++的简洁明了的语法，这种语法简化了以前一些给程序员带来困扰的问题。尽管做了这些简化，但 C#仍保持了 C++原有的功能，所以现在没有理由不从 C++转向 C#。C#语言并不难，也非常适合于学习基本编程技术。易于学习，再加上.NET Framework 的功能，使 C#成为开始您编程生涯的绝佳方式。

C#的最新版本 C# 4 是.NET Framework 4 的一部分，它建立在已有的成功基础之上，还添加了一些更吸引人的功能。Visual Studio 的最新版本 Visual Studio 2010 和开发工具的 Express 系列(包括 Visual C# 2010 Express)也有许多变化和改进，这大大简化了编程工作，显著提高了效率。

本书将全面介绍 C#编程的所有知识，从该语言本身一直到 Windows 和 Web 编程，再到数据源的使用，最后是一些新的高级技术。我们还将学习 Visual C# 2010 Express、Visual Web Developer 2010 Express 和 Visual Studio 2010 的功能和利用它进行应用程序开发的各种方式。

本书文笔优美流畅，阐述清晰，每一章都以前面章节的内容为基础，便于读者掌握高级技术。每个概念都会根据需要来介绍和讨论，而不会突然冒出某个技术术语来妨碍读者的阅读和理解。本书尽量减少使用的技术术语数量，但如果需要，将根据上下文进行正确的定义和布置。

本书的作者都是各自领域的专家，都是 C#语言和.NET Framework 的爱好者，没有人比他们更有资格讲授 C#了，他们将在您掌握从基本规则到高级技术的过程中为您保驾护航。除了基础知识之外，本书还有许多有益的提示、练习、完全成熟的示例代码(可以从 p2p.wrox. com 上下载)，在您的职业生涯中一定会反复用到它们。

本书将毫无保留地传授这些知识，希望读者能通过阅读本书成长为最优秀的程序员。

0.1　本书读者对象

本书主要针对想学习如何使用.NET Framework 编写 C#程序的所有人。本书前面的章节介绍该语言本身，读者不需要具备任何编程经验。以前对其他语言有一定了解的开发人员，会觉得这些章节的内容非常熟悉。C#语法的许多方面都与其他语言相同，许多结构对所有的编程语言来说都是相通的(例如，循环和分支结构)。但是，即使是有经验的程序员也可以从这些章节中获益，理解这些技术应用于 C#的特征。

如果读者是编程新手，就应从头开始学习。如果读者对.NET Framework 比较陌生，但知道如何编程，就应阅读第 1 章，然后快速跳读后面几章，这样就能掌握 C#语言的应用方式了。如果读者知道如何编程，但以前从未接触过面向对象的编程语言，就应从第 8 章开始阅读以后的章节。

如果读者对 C#语言比较了解，就可以集中精力学习详细论述最新.NET Framework 和 C#语言开发的章节，尤其是集合、泛型和 C# 4 语言的新增内容(第 11～14 章)，或者完全跳过本书的第 I 部分，从第 15 章开始学习。

本书章节的编排方式可以达到两个目的：可以按顺序阅读这些章节，将其视为 C#语言的一个完整教程。还可以按照需要深入学习这些章节，将其作为一本参考资料。

除了核心内容之外，从第 3 章开始，每章末尾还包含一组练习，完成这些练习有助于读者理解所学的内容。练习包括简单的选择题、判断题以及需要修改或建立应用程序的较难问题。练习答案在 www.wrox.com 的本书 Web 页面上和 http://www.tupwk.com.cn 联机提供。

0.2 本版的新内容

本书特别注重与 C# 4、.NET 4 的一致性。每一章都进行了彻底的检查，删除了不太相关的内容，增加了新材料。所有代码都在最新版本的开发工具上进行了测试，所有屏幕图都在 Windows 7 上重新截取，以提供最新的窗口和对话框。

尽管我们不喜欢承认失误，但还是修订了前几版中的错误，处理了许多其他的读者评论。我们希望不要出现太多的新错误，但一旦发现了错误，我们的 Web 专家就会联机修改它们。

本版本的亮点包括：

- 增加并改进了代码示例。
- 涵盖 C# 4 的所有新内容，包括简单的语言改进，例如方法的命名参数和可选参数，还包括高级技术，例如泛型类型中的变体。
- 十分合理地介绍高级技术，重点是适合于新手、较易理解的内容。

0.3 本书结构

本书分为 6 个部分。

- **前言**：概述本书的内容。
- **C#语言**：介绍了 C#语言的所有内容，从基础知识到面向对象的技术，一应俱全。
- **Windows 编程**：介绍如何用 C#编写 Windows 应用程序，如何部署它们。
- **Web 编程**：描述 Web 应用程序的开发、Web 服务和 Web 应用程序的部署。
- **数据访问**：介绍如何在应用程序中使用数据，包括存储在硬盘文件上的数据、以 XML 格式存储的数据和数据库中的数据。
- **其他技术**：讲述使用 C#和.NET Framework 的一些额外方式，包括由.NET 3.0 引入然后经.NET 4 改进的 WPF、WCF 和 WF 技术。

下面介绍本书 5 个重要部分中的章节。

0.3.1 C#语言(第 1～14 章)

第 1 章介绍 C#及其与.NET 的关系，了解在这个环境下编程的基础知识，以及 Visual C# 2010

Express(VCE)和 Visual Studio 2010(VS)与它的关系。

第 2 章开始介绍如何编写 C#应用程序，学习 C#的语法，并将 C#和样例命令行、Windows 应用程序结合起来使用。这些示例将说明 C#如何快速轻松地启动和运行，并附带介绍 VCE 和 VS 开发环境以及本书将要使用的基本窗口和工具。

第 3 章介绍 C#语言的更多基础知识，分析变量的含义以及如何操纵它们。第 4 章将用流程控制(循环和分支)改进应用程序的结构，第 5 章介绍一些高级变量类型，如数组。第 6 章开始以函数形式封装代码，这样就更易于执行重复的操作，使代码更容易让人理解。

从第 7 章开始将运用 C#语言的基础知识，调试应用程序。这包括在运行应用程序时输出跟踪信息，使用 VS 查找错误，在强大的调试环境中找出解决问题的办法。

第 8 章将学习面向对象编程(Object-Oriented Programming，OOP)。首先了解这个术语的含义，回答“什么是对象？”。OOP 初看起来是较难的问题。我们将用一整章的篇幅来介绍它，解释对象的强大之处。直到本章的最后才会使用 C#代码。

第 9 章将理论应用于实践，开始在 C#应用程序中使用 OOP 时，一切都会发生变化，而这才体现出 C#的真正威力。第 10 章首先介绍如何定义类和接口，然后探讨类成员(包括字段、属性和方法)，在这一章的最后将开始创建一个扑克牌游戏应用程序，这个应用程序将在几章中开发完成，它非常有助于理解 OOP。

学习了 OOP 在 C#中的工作原理后，第 11 章将介绍几种常见的 OOP 场景，包括处理对象集合、比较和转换对象。第 12 章讨论.NET 2.0 中 C#的一个非常有用的特性——泛型，利用它可以创建非常灵活的类。第 13 章通过一些其他技术和事件(它在 Windows 编程中非常重要)结束 C#语言和 OOP 的讨论。最后，第 14 章介绍 C# 3.0 和 4 中引入的新特性。

0.3.2　Windows 编程(第 15～17 章)

第 15 章开始介绍 Windows 编程的概念，理解在 VCE 和 VS 中如何实现 Windows 编程。这一章也是从基础知识开始介绍，并在本章和第 16 章中逐渐介绍较复杂的内容。第 16 章学习如何在应用程序中使用.NET Framework 提供的各种控件。我们将简要论述.NET 如何以图形化方式建立 Windows 应用程序，以最少的时间和精力创建高级应用程序。

第 17 章讨论应用程序的部署，包括建立安装程序，以便用户快速安装和运行应用程序。

0.3.3　Web 编程(第 18～20 章)

这个部分的结构与 Windows 编程部分类似。首先，第 18 章描述了构成最简单 Web 应用程序的控件，如何把它们组合在一起，让它们使用 ASP.NET 执行任务。接着介绍了更高级的技术、ASP.NET AJAX、各种控件、Web 上下文下的状态管理以及 Web 标准的遵循。

第 19 章将走入 Web 服务的精彩世界，它可以编程访问 Internet 上的信息和功能，可以把复杂数据和功能以独立于平台的方式嵌入 Web 和 Windows 应用程序。这一章讨论如何使用和创建 Web 服务，以及.NET 提供的其他工具，如安全性。

最后，第 20 章探讨 Web 应用程序和服务的部署，尤其是可以通过单击按钮把应用程序发布到 Web 上的 VS 和 VWD 特性。

0.3.4 数据访问(第 21～24 章)

第 21 章介绍了应用程序如何将数据保存到磁盘以及如何检索磁盘上的数据(作为简单的文本文件或者更复杂的数据表示方式)。这一章还将讨论如何压缩数据，如何操纵旧数据(例如，用逗号分隔的值(CSV)文件)，如何监视和处理文件系统的变化。

第 22 章学习数据交换的事实标准 XML。之前的章节接触过 XML 几次，而这一章将了解 XML 的基本规则，论述 XML 的所有功能。

本部分其余章节介绍 LINQ(这是内置于.NET Framework 最新版本中的查询语言)。第 23 章简要介绍 LINQ，第 24 章使用 LINQ 访问数据库和其他数据。

0.3.5 其他技术(第 25～27 章)

本书最后一部分将讨论.NET Framework 最新版本中出现的几项新技术。第 25 章介绍 Windows Presentation Foundation(WPF)，了解它给 Windows 和 Web 开发带来哪些重大的变化。第 26 章介绍 Windows Communication Foundation(WCF)，它把 Web 服务的概念扩展和改进为一种企业级通信技术。本书最后一章是第 27 章，介绍了 Windows Workflow Foundation(WF)，它允许在应用程序中执行工作流功能，因此可以定义一些操作，这些操作由外部的交互操作控制，按特定顺序执行，这对许多类型的应用程序都很有帮助。

0.4 使用本书的要求

本书中 C#和.NET Framework 的代码和描述都适用于.NET 4。除了 Framework 之外，不需要其他东西就可以理解本书的这个方面，但许多示例都需要使用开发工具。本书将 Visual C# 2010 Express 作为主要开发工具，一些章节则使用了 Visual Web Developer 2010 Express。另外，一些功能只能在 Visual Studio 2010 中使用，这会在相应的地方明确指出。

0.5 源代码

在读者学习本书中的示例时，可以手工输入所有代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 http://www.wrox.com/或 www.tupwk.com.cn/downpage 上下载。登录到站点 http://www.wrox.com/，使用 Search 工具或使用书名列表就可以找到本书。接着单击本书细目页面上的 Download Code 链接，就可以获得所有源代码。

注释：

由于许多图书的标题都很类似，所以按 ISBN 搜索是最简单的，本书英文版的 ISBN 是 978-0-470-50226-6。

在下载了代码后，只需用自己喜欢的解压缩软件对它进行解压缩即可。另外，也可以进入 http://www.wrox.com/dynamic/books/download.aspx 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有代码。

0.6　勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果您在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

请给 wkservice@vip.163.com 发电子邮件，我们就会检查您的反馈信息，如果是正确的，我们将在本书的后续版本中采用。在本书编辑过程中，我们接受了热心读者白爽针对第 4 版中文译著提出的一些修改意见，在此特向白爽表示衷心感谢。

要在网站上找到本书英文版的勘误表，可以登录 http://www.wrox.com，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看到 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/booklist.shtml。

0.7　P2P.WROX.COM

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于您张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，它可以给您传送感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 http://p2p.wrox.com 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读使用协议，并单击 Agree 按钮。

(3) 填写加入该论坛所需要的信息和自己希望提供的其他信息，单击 Submit 按钮。

(4) 您会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

注释：

不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须加入该论坛。

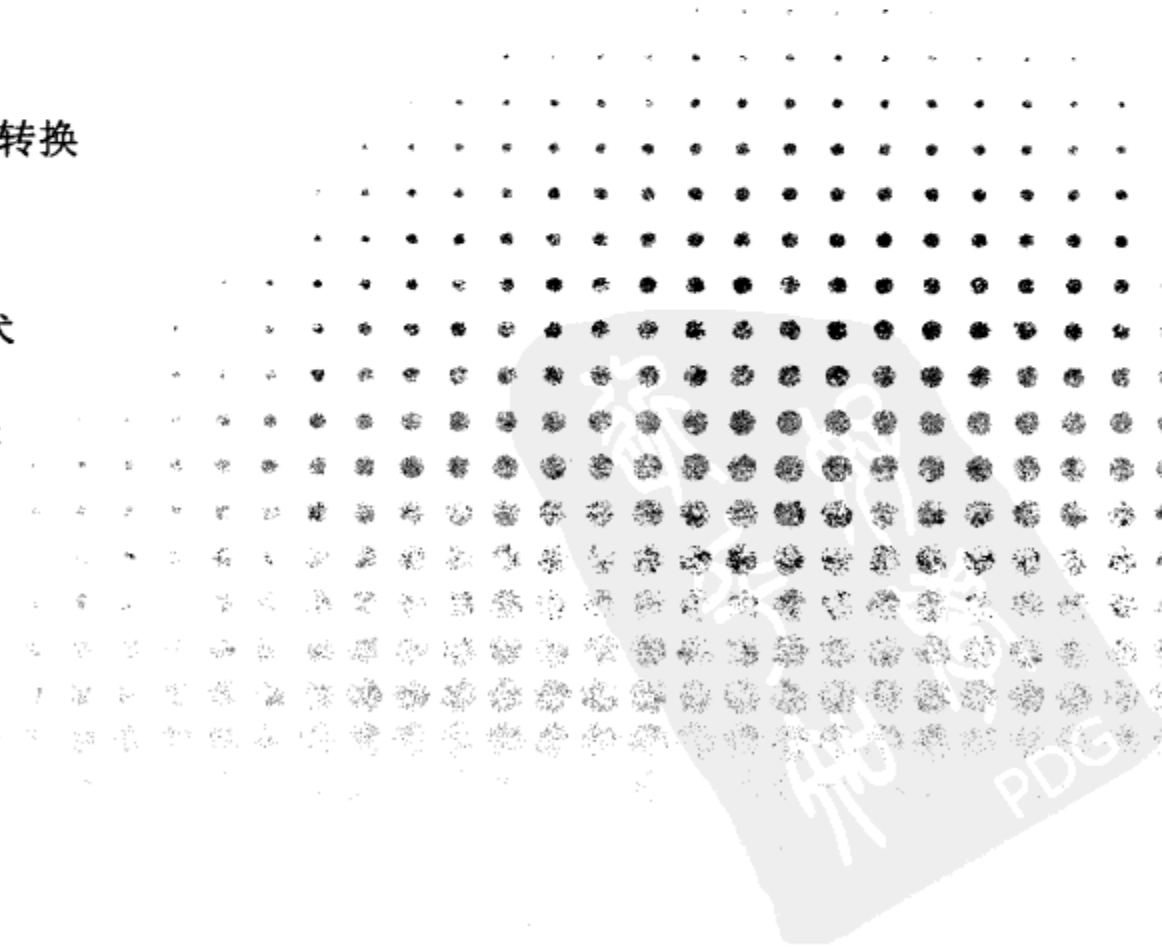
加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

关于使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作情况以及 P2P 和 Wrox 图书的许多常见问题。要阅读 FAQ，可以在任意 P2P 页面上单击 FAQ 链接。

目　录

第 I 部分

C# 语 言



第 1 章

C# 简 介

本章内容：

- .NET Framework 的功能及其包含的内容
- .NET 应用程序的工作原理
- C#的概念及其与.NET Framework 的关系
- 用 C#创建.NET 应用程序的工具

本书第 I 部分将介绍使用 C# 语言所需的基础知识。第 1 章将概述 C#和.NET Framework，包括这两项技术的含义、作用及相互关系。

首先讨论.NET Framework。这种技术包含的许多概念初看起来都不是很容易掌握的。也就是说，我们必须在很短的篇幅里介绍许多新概念，但是，快速浏览这些基础知识对于理解如何利用 C#进行编程是非常重要的，本书后面将详细论述这里提到的许多论题。

之后，本章将讨论 C#本身，包括它的起源以及与C++的类似之处。最后介绍本书使用的主要工具：Visual Studio 2010 (VS)和 Visual C# 2010 Express(VCE)。

1.1 .NET Framework 的含义

.NET Framework(现在是版本4)是Microsoft为开发应用程序而创建的一个具有革命意义的平台。这句话最有趣的地方在于它的广义性，但这是有原因的。首先，注意这句话没有说“在 Windows 操作系统上开发应用程序”。尽管.NET Framework 的 Microsoft 版本运行在 Windows 操作系统上，但以后将推出运行在其他操作系统上的版本，例如 Mono，它是.NET Framework 的开源版本(包含 C#编译器)，该版本可以运行在几个操作系统上，包括各种 Linux 版本和 Mac OS。另外，还可以在个人数字助手(PDA)类设备和一些智能电话上使用 Microsoft .NET Compact Framework(基本上是完整 .NET Framework 的一个子集)。使用.NET Framework 的一个重要原因是它可以作为集成各种操作系统的方式。

另外，上面给出的.NET Framework 定义并未限制应用程序的类型。这是因为本来就没有限制。可以使用.NET Framework 创建 Windows 应用程序、Web 应用程序、Web 服务和其他各种类型的应

用程序。另外注意，对于 Web 应用程序，按照定义，它们是多平台的应用程序，因为任何带 Web 浏览器的系统都可以访问它们。最近新增了 Silverlight，这种类别还包含运行在客户浏览器内部的应用程序，以及仅以 HTML 格式显示 Web 内容的应用程序。

.NET Framework 的设计方式确保它可以用于各种语言，包括本书介绍的C#语言，以及 C++、Visual Basic、JScript，甚至一些旧的语言，如 COBOL。为此，还推出了这些语言的.NET 版本，目前还在不断推出更多版本。所有这些语言都可以访问.NET Framework，它们彼此之间还可以通信。C#开发人员可以使用 Visual Basic 程序员编写的代码，反之亦然。

所有这些提供了意想不到的多样性，这也是.NET Framework 具有诱人前景的部分原因。

1.1.1　.NET Framework 的内容

.NET Framework 主要包含一个非常大的代码库，可以在客户语言(如 C#)中通过面向对象编程技术(OOP)来使用这些代码。这个库分为多个不同的模块，这样就可以根据希望得到的结果来选择使用其中的各个部分。例如，一个模块包含 Windows 应用程序的构件，另一个模块包含网络编程的代码块，还有一个模块包含 Web 开发的代码块。一些模块还分为更具体的子模块，例如，在 Web 开发模块中，有用于建立 Web 服务的子模块。

其目的是，不同操作系统可以根据自己的特性，支持其中的部分或全部模块。例如，PDA 支持所有的核心.NET 功能，但不需要某些更高级的模块。

部分.NET Framework 库定义了一些基本类型。类型是数据的一种表达方式，指定其中最基础的部分(如 32 位带符号的整数)，以便使用.NET Framework 在各种语言之间进行交互操作。这称为通用类型系统(Common Type System，CTS)。

除了提供这个库以外，.NET Framework 还包含.NET 公共语言运行库(Common Language Runtime，CLR)，它负责管理用.NET 库开发的所有应用程序的执行。

1.1.2　使用.NET Framework 编写应用程序

使用.NET Framework 编写应用程序，就是使用.NET 代码库编写代码(使用支持 Framework 的任何一种语言)。本书用 VS 和 VCE 进行开发，VS 是一种强大的集成开发环境，支持 C#(以及托管和非托管 C++、Visual Basic 和其他一些语言)。VCE 是 VS 的一个删节版本(免费)，仅支持 C#。这些环境的优点是便于把.NET 功能集成到代码中。我们创建的代码完全是 C#代码，但使用了.NET Framework，并在需要时利用了 VS 和 VCE 中的其他工具。

为了执行 C#代码，必须把它们转换为目标操作系统能够理解的语言，即本机代码(native code)。这种转换称为编译代码，由编译器执行。但在.NET Framework 下，此过程包括两个阶段。

1. CIL 和 JIT

在编译使用.NET Framework 库的代码时，不是立即创建专用于操作系统的本机代码，而是把代码编译为通用中间语言(Common Intermediate Language，CIL)代码，这些代码并非专门用于任何一种操作系统，也非专门用于 C#。其他.NET 语言，如 Visual Basic .NET 也可以在第一阶段编译为这种语言，开发 C#应用程序时，这个编译步骤由 VS 或 VCE 完成。

显然，要执行应用程序，必须完成更多工作，这是 Just-In-Time(JIT)编译器的任务，它把 CIL 编译为专用于 OS 和目标机器结构的本机代码。这样 OS 才能执行应用程序。这里编译器的名称

Just-In-Time 反映了 CIL 代码仅在需要时才编译的事实。

过去，常常需要把代码编译为几个应用程序，每个应用程序都用于特定的操作系统和 CPU 结构。这通常是一种优化形式(例如，为了让代码在 AMD 芯片组上运行得更快)，而且有时是非常重要的(例如，对于工作在 Win9x 和 WinNT/2000 环境下的应用程序)。现在就不必要了，因为顾名思义，JIT 编译器使用 CIL 代码，而 CIL 代码是独立于计算机、操作系统和 CPU 的。目前有几种 JIT 编译器，每种编译器都用于不同的结构，我们总能找到一个合适的编译器创建所需的本机代码。

这样，用户需要做的工作就比较少了。实际上，可以忽略与系统相关的细节，把注意力集中在代码的功能上就够了。

读者可以遇到过 Microsoft Intermediate Language(MSIL)或 IL，MSIL 是 CIL 原来的名称，许多开发人员仍沿用这个术语。

2. 程序集

在编译应用程序时，所创建的 CIL 代码存储在一个程序集中。程序集包括可执行的应用程序文件(这些文件可以直接在 Windows 上运行，不需要其他程序，其扩展名是.exe)和其他应用程序使用的库(其扩展名是.dll)。

除了包含 CIL 外，程序集还包含元信息(即程序集中包含的数据的信息，也称为元数据)和可选的资源(CIL 使用的其他数据，例如，声音文件和图片)。元信息允许程序集是完全自描述的。不需要其他信息就可以使用程序集，也就是说，我们不会遇到下述情形：不能把需要的数据添加到系统注册表中，而这种情形在使用其他平台进行开发时常常出现。

因此，部署应用程序就非常简单了，只需把文件复制到远程计算机上的目录下即可。因为不需要目标系统上的其他信息，所以只需从该目录中运行可执行文件即可(假定安装了.NET CLR)。

当然，不必把运行应用程序需要的所有信息都安装到一个地方。可以编写一些代码来执行多个应用程序所要求的任务。此时，通常把这些可重用的代码放在所有应用程序都可以访问的地方。在.NET Framework 中，这个地方是全局程序集缓存(Global Assembly Cache，GAC)，把代码放在这个缓存中是很简单的，只需把包含代码的程序集放在包含该缓存的目录中即可。

3. 托管代码

在将代码编译为 CIL，再用 JIT 编译器将它编译为本机代码后，CLR 的任务还没有全部完成，还需要管理正在执行的用.NET Framework 编写的代码(这个执行代码的阶段通常称为运行时(runtime))。即 CLR 管理着应用程序，其方式是管理内存、处理安全性以及允许进行跨语言调试等。相反，不受 CLR 控制运行的应用程序属于非托管类型，某些语言如 C++可以用于编写这类应用程序，例如，访问操作系统的低级功能。但是在 C#中，只能编写在托管环境下运行的代码。我们将使用 CLR 的托管功能，让.NET 自己与操作系统进行交互。

4. 垃圾回收

托管代码最重要的一个功能是垃圾回收(garbage collection)。这种.NET 方法可确保应用程序不再使用某些内存时，就会完全释放这些内存。在.NET 推出以前，这项工作主要由程序员负责，代码中

的几个简单错误会把大块内存分配到错误的地方，使这些内存神秘失踪。这通常意味着计算机的速度逐渐减慢，最终导致系统崩溃。

.NET 垃圾回收会定期检查计算机内存，从中删除不再需要的内容。它不设置时间帧，可能一秒钟内会进行上千次的检查，也可能几秒钟才检查一次，或者随时进行检查，但一定会进行检查。

这里要给程序员一些提示。因为在不可预知的时间执行这项工作，所以在设计应用程序时，必须留意这一点。需要许多内存才能运行的代码应自己执行这样的检查，而不是坐等垃圾回收，但这不像听起来那样难。

5. 把它们组合在一起

在继续学习之前，先总结一下上述创建.NET 应用程序所经历的步骤：

(1) 使用某种.NET 兼容语言(如 C#)编写应用程序代码，如图 1-1 所示。

(2) 把代码编译为 CIL，存储在程序集中，如图 1-2 所示。

图　1-1

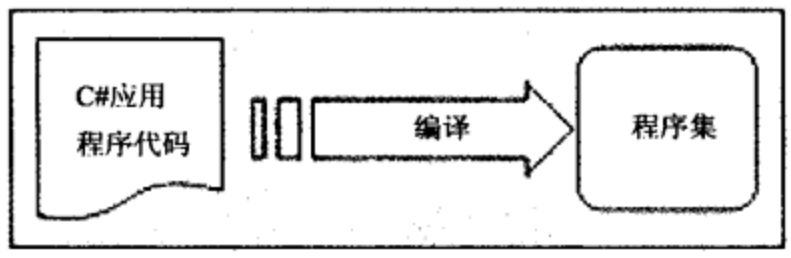


图　1-2

(3) 在执行代码时(如果这是一个可执行文件，就自动运行，或者在其他代码使用它时运行)，首先必须使用 JIT 编译器将代码编译为本机代码，如图 1-3 所示。

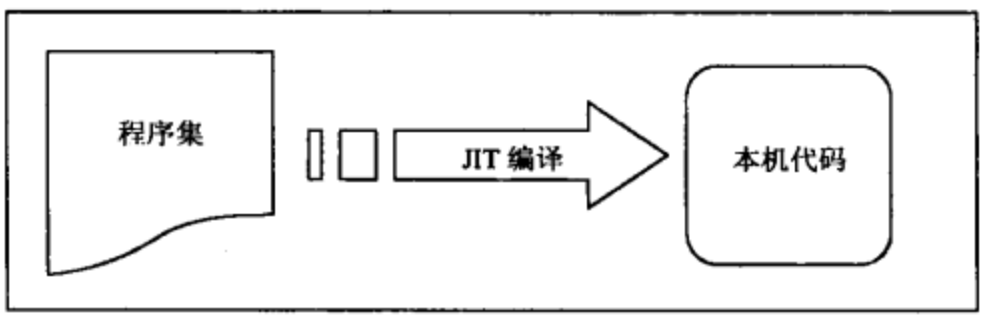


图　1-3

(4) 在托管的 CLR 环境下运行本机代码，以及其他应用程序或进程，如图 1-4 所示。

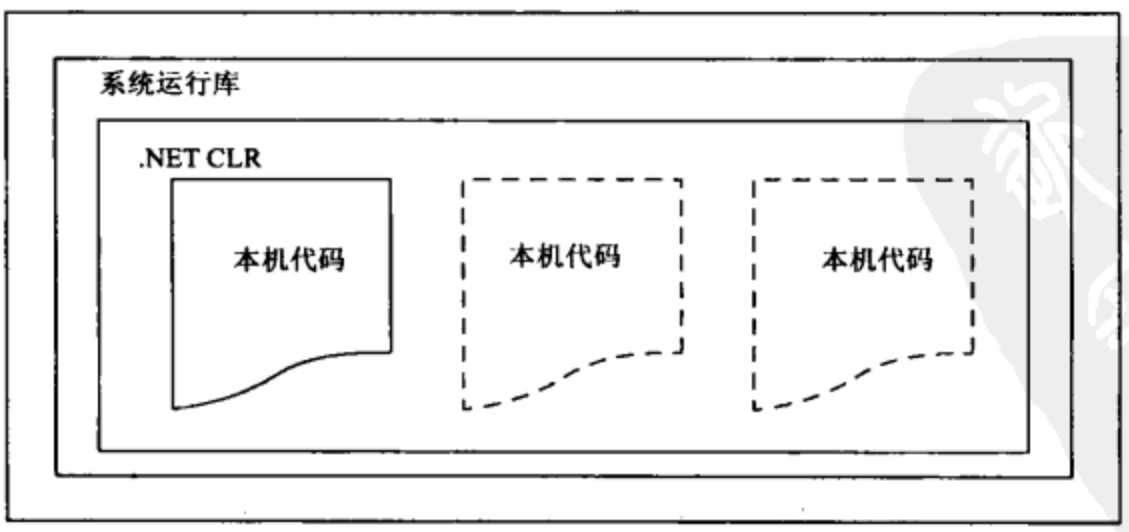


图　1-4

6. 链接

在上述过程中还有一点要注意。在第(2)步中编译为 CIL 的 C#代码不一定包含在单独文件中，可以把应用程序代码放在多个源代码文件中，再把它们编译到一个程序集中。这个过程称为链接(linking)，是非常有用的。原因是处理几个较小的文件比处理一个大文件要简单得多。可以把逻辑上相关的代码分解到一个文件中，以便单独进行处理，这也更易于在需要时找到特定的代码块，让开发小组把编程工作分解为一些可管理的块，让每个人编写一小块代码，而不会破坏已编写好的代码部分或其他人正在处理的部分。

1.2　C#的含义

如上所述，C#是可用于创建要运行在.NET CLR 上的应用程序的语言之一，它从 C 和 C++语言演化而来，是 Microsoft 专门为使用.NET 平台而创建的。因为 C#是近期发展起来的，所以吸取了以往的教训，考虑了其他语言的许多优点，并解决了它们的问题。

使用 C#开发应用程序比使用 C++简单，因为其语法比较简单。但是，C#是一种强大的语言，在 C++中能完成的任务几乎都能利用 C#完成。如前所述，C#中与 C++高级功能等价的功能(例如直接访问和处理系统内存)，只能在标记为“不安全(unsafe)”的代码中使用。这个高级编程技术存在潜在威胁(正如它的名称所暗示的)，因为它可能覆盖系统中重要的内存块，导致严重后果。因此，本书不讨论这个问题。

C#代码常比 C++略长一些。这是因为 C#是一种类型安全的语言(与 C++不同)。在外行人看来，这表示一旦为某个数据指定了类型，就不能转换为另一个不相关的类型。所以，在类型之间转换时，必须遵守严格的规则。执行相同的任务时，用 C#编写的代码通常比用 C++编写的代码长。但 C#代码更健壮，调试起来也比较简单，.NET 始终可以随时跟踪数据的类型。在 C#中，不能完成诸如“把 4 字节的内存放在这个数据中，使之有 10 个字节长，并把它解释为 X”等的任务，但这并不是一件坏事。

C#只是用于.NET 开发的一种语言，但它是最好的一种语言。C#的优点是，它是唯一彻头彻尾为.NET Framework 设计的语言，是在移植到其他操作系统上的.NET 版本中使用的主要语言。要使诸如 VB.NET 的语言尽可能类似于其以前的语言，且仍遵循 CLR，就不能完全支持.NET 代码库的某些功能，至少需要不常见的语法。但 C#能使用.NET Framework 代码库提供的每种功能。.NET 的最新版本还对 C#语言进行了几处改进，满足了开发人员的要求，使之更强大。

1.2.1　用 C#能编写什么样的应用程序

如前所述，.NET Framework 没有限制应用程序的类型。C#使用的是.NET Framework，所以也没有限制应用程序的类型。这里仅讨论几种常见的应用程序类型。

- **Windows 应用程序**　这些应用程序(如 Microsoft Office)具有我们很熟悉的 Windows 外观和操作方式，使用.NET Framework 的 Windows Forms 模块就可以简便地生成这种应用程序。Windows Forms 模块是一个控件库，其中的控件(例如，按钮、工具栏、菜单等)可以用于建立 Windows 用户界面(UI)。另外，还可以使用 Windows Presentation Foundation (WPF)建立 Windows 应用程序，WPF 提供了更大的灵活性和更卓越的功能。

- **Web 应用程序**　它们是一些 Web 页面，可以通过任何 Web 浏览器查看。.NET Framework 包括一个动态生成 Web 内容的强大系统，允许进行个性化和实现安全性等。这个系统叫作 Active Server Pages .NET(ASP.NET)，我们可以使用 C#通过 Web Forms 创建 ASP.NET 应用程序。还可以使用 Silverlight 编写在浏览器内部运行的应用程序。
- **Web 服务**　这是创建各种分布式应用程序的激动人心的新方式，使用 Web 服务可以通过 Internet 虚拟交换数据。无论使用什么语言创建 Web 服务，也无论 Web 服务驻留在什么系统上，都使用一样简单的语法。对于更高级的功能，还可以创建 Windows Communication Foundation(WCF)服务。

这些类型也需要某种形式的数据库访问，这可以通过.NET Framework 的 Active Data Objects .NET(ADO.NET)部分、ADO.NET Entity Framework 或 C#的 LINQ(Language Integrated Query)功能来实现。也可以使用许多其他资源，例如，创建联网组件、输出图形、执行复杂数学任务的工具。

1.2.2　本书中的 C#

本书第 I 部分介绍 C# 语言的语法和用法，但不过分强调.NET Framework。这是必需的，因为我们不能没有一点儿 C# 编程基础就使用.NET Framework。首先介绍一些比较简单的内容，把较复杂的面向对象编程(Object-Oriented Programming，OOP)论题放在基础知识的后面论述。假定读者没有一点儿编程的知识，这些是首要的规则。

学习了基础知识后，本书还将介绍如何开发更复杂、更有用的应用程序。本书的第 II 部分将讨论 Windows Forms 编程，第III部分将研究 Web 应用程序和 Web 服务编程，第Ⅳ部分将讲述数据访问(对数据库、文件系统和 XML 数据的访问)，第 V 部分将介绍其他一些有趣的.NET 论题。

1.3　Visual Studio 2010

本书使用 Visual Studio 2010(VS)或 Visual C# 2010 Express(VCE)开发工具进行所有的 C#编程，包括简单的命令行应用程序，乃至比较复杂的项目类型。VS 不是开发 C#应用程序所必需的开发工具或集成开发环境(IDE)，但使用它可以使任务更简单一些。可以在基本的文本编辑器(如常见的记事本)中处理 C#源代码文件，再使用命令行应用程序(是.NET Framework 的一部分)把代码编译到程序集中。但是，为什么不使用功能完备的 IDE 呢？

下面列出的是一些使 VS 成为.NET 开发首选工具的功能。

- VS 可以自动执行编译源代码的步骤，同时可以完全控制重写它们时应使用的任何选项。
- VS 文本编辑器为 VS 支持的语言(包括 C#)量身定制，这样就可以智能检测错误，在输入代码时给出合适的推荐代码，这个功能称为 IntelliSense。
- VS 包括 Windows Forms、Web Forms 及其他应用程序的设计器，允许 UI 元素的简单拖放设计。
- 在 C#中，许多类型的项目都可以用已有的“样板”代码来创建，不需要从头开始。各种代码文件通常已经准备好了，减少了从头开始一个项目所用的时间。对于“Starter Kit”项目类型来说尤其如此，该项目类型可以以功能全面的应用程序为基础进行开发。一些 Starter Kit 项目类型包含在 VS 安装程序中，还可以联机使用更多的该项目类型。

- VS 包括几个可自动执行常见任务的向导，其中的很多任务可以在已有的文件中添加合适的代码，在某些情况下，您甚至不需要考虑语法的正确性。
- VS 包含许多强大的工具，可以显示项目中的元素并允许在其中导航，这些元素可以是 C#源代码文件，也可以是其他资源，例如位图图像或声音文件。
- 除了在 VS 中编写应用程序比较简单外，还可以创建部署项目，以便为客户提供代码，并使客户方便地完成安装。
- 在开发项目时，VS 允许使用高级调试技巧，例如，能在代码中一次调试一条指令，并监视应用程序的状态。

C#还有许多功能，希望读者能掌握它们！

1.3.1 Visual Studio 2010 Express 产品

除了 Visual Studio 2010 之外，Microsoft 还提供了几个更简单的开发工具，称为 Visual Studio 2010 Express 产品。可以在 http://www.microsoft.com /express 上免费获得它们。

其中两个产品是 Visual C# 2010 Express 和 Visual Web Developer 2010 Express，它们都可以创建所需的几乎所有 C#应用程序。在功能上它们都是 VS 的删节版本，但外观和操作方式是一样的。尽管它们提供了 VS 的许多功能，但缺少一些重要的功能；不过我们仍可以在学习本书的过程中使用它们。

本书尽可能使用 VCE 开发 C#应用程序，仅在需要某些功能时才使用 VS。当然，如果有 VS，就无需使用 Express 产品。

1.3.2 解决方案

在使用 VS 或 VCE 开发应用程序时，可以通过创建解决方案来完成。在 VS 和 VCE 术语中，解决方案不仅仅是一个应用程序，它还包含项目，可以是 Windows Forms 项目和 Web Form 项目等。但是，解决方案可以包含多个项目，这样，即使相关的代码最终在硬盘上的多个位置编译为多个程序集，也可以把它们组合到一个地方。

这是非常有用的，因为它可以处理“共享”代码(这些代码放在 GAC 中)，同时，应用程序也使用这段共享代码。在使用唯一的开发环境时，调试代码是非常容易的，因为可以在多个代码块中单步调试指令。

1.4 小结

本章简要介绍了.NET Framework，并讨论了如何轻松创建各种强大的应用程序。还探讨了把用 C#等语言编写的代码转换为可运行的应用程序所需要做的工作，以及使用在.NET 公共语言运行库下运行的托管代码有什么优点。

本章还阐述了C#的实质，以及它与.NET Framework 的关系，描述了进行 C#开发时所使用的工具——Visual Studio 2010 和 Visual C# 2010 Express。

第 2 章将介绍如何运行一些 C#代码，介绍基础知识，并集中讨论 C#语言本身，而不是过多地讨论 IDE 的工作原理。

1.5　本章要点

主　　题	重 要 概 念
.NET Framework 基础	.NET Framework 是 Microsoft 最新的开发平台，目前的版本是 4。它包括一个公共类型系统(CTS)和一个公共语言运行库(CLR)。.NET Framework 应用程序使用面向对象的编程(OOP)的方法编写，通常包含托管代码。托管代码的内存管理由.NET 运行库处理，其中包括垃圾回收
.NET Framework 应用程序	用.NET Framework 编写的应用程序首先编译为 CIL。在执行应用程序时，JIT 把 CIL 编译为本机代码。应用程序编译后，把不同的部分链接到包含 CIL 的程序集中
C#基础	C#是包含在.NET Framework 中的一种语言，它是以前的语言(如 C++)的一种演变，可以用于编写任意应用程序，包括网站和 Windows 应用程序
集成开发环境(IDE)	可以在 Visual Studio 2010 中用 C#编写任意类型的.NET 应用程序，还可以在免费的、但功能稍弱的 Express 产品系列(包括 Visual C# Developer Express)中用 C#创建.NET 应用程序。这两种 IDE 都使用解决方案，解决方案可以包含多个项目

第 2 章

编写C#程序

本章内容：

- Visual Studio 2010 和 Visual C# 2010 Express Edition 的基础知识
- 如何编写简单的控制台应用程序
- 如何编写 Windows Form 应用程序

第 1 章用一定的篇幅讨论了 C#是什么，它是如何适应.NET Framework 的，现在就该编写一些代码了。本书主要使用 Visual Studio 2010(VS)和 Visual C# 2010 Express(VCE)，所以首先介绍这些开发环境的一些基础知识。

VS 是一个庞大复杂的产品，可能会使初学者望而生畏，但使用它创建简单的应用程序是非常容易的。在本章开始使用 VS 时，不需要了解许多知识，就可以编写 C#代码。本书的后面将介绍 VS 能执行的更复杂的操作，现在仅介绍基础知识。

从 VCE 入手要简单得多。在本书的前面部分，所有示例都是在这个 IDE 中编写。但是如果选择使用 VS，所有工作都或多或少地以相同的方式完成。因此，本章介绍这两个 IDE，先介绍 VS。

介绍完 IDE 后，将把两个简单的应用程序组合在一起。现在不要过多地考虑代码，只要应用程序可以运行即可。在这些早期的示例中熟悉了应用程序的创建过程，不久之后就会适应这个过程了。

下面要创建的第一个应用程序是一个简单的控制台应用程序。控制台应用程序没有使用图形化的 Windows 环境，所以不需要考虑按钮、菜单、用鼠标指针进行的交互等，而是在命令行窗口中运行应用程序，用更简单的方式与其交互。

第二个应用程序是一个 Windows Forms 应用程序，其外观和操作方式对 Windows 用户来说会非常熟悉，而且该应用程序创建起来并不费力。但所需代码的语法比较复杂，尽管在许多情况下，并不需要考虑细节。

本书接下来的两个部分也使用这两种应用程序类型，但开始时略微强调一下控制台应用程序。在学习 C#语言时，不需要了解 Windows 应用程序的其他灵活性能。控制台应用程序的简单性可以让我们集中精力学习语法，而无需考虑应用程序的外观和操作方式。

2.1　开发环境

本节讨论 VS 和 VCE 开发环境，先介绍 VS。这些环境是类似的，无论使用哪个 IDE，都应了解这两个环境。

2.1.1　Visual Studio 2010

在第一次加载 VS 时，会立即显示一系列窗口以及一组菜单和工具栏图标，其中大多数窗口是空的。本书将使用大多数窗口，读者很快就会熟悉它们。

如果是第一次运行VS，则屏幕上会显示一个首选项列表，如果用户使用过这个开发环境的旧版本，则可以在这里做出选择，这些选择会影响到很多方面，例如，窗口的布局、控制台窗口运行的方式等。所以应选择 Visual C# Development Settings，否则步骤就不像下面描述的这样了。注意，可用选项会随着安装 VS 时选择的选项而变化，但只要选择安装 C#，这个选项就是可用的。

如果不是第一次运行 VS，但选择了另一个选项，不必惊慌。为了把设置重置为 Visual C# Development Settings，只需导入它们即可。为此，单击 Tools 菜单上的 Import and Export Settings 选项，再选中 Reset All Settings 选项，如图 2-1 所示。

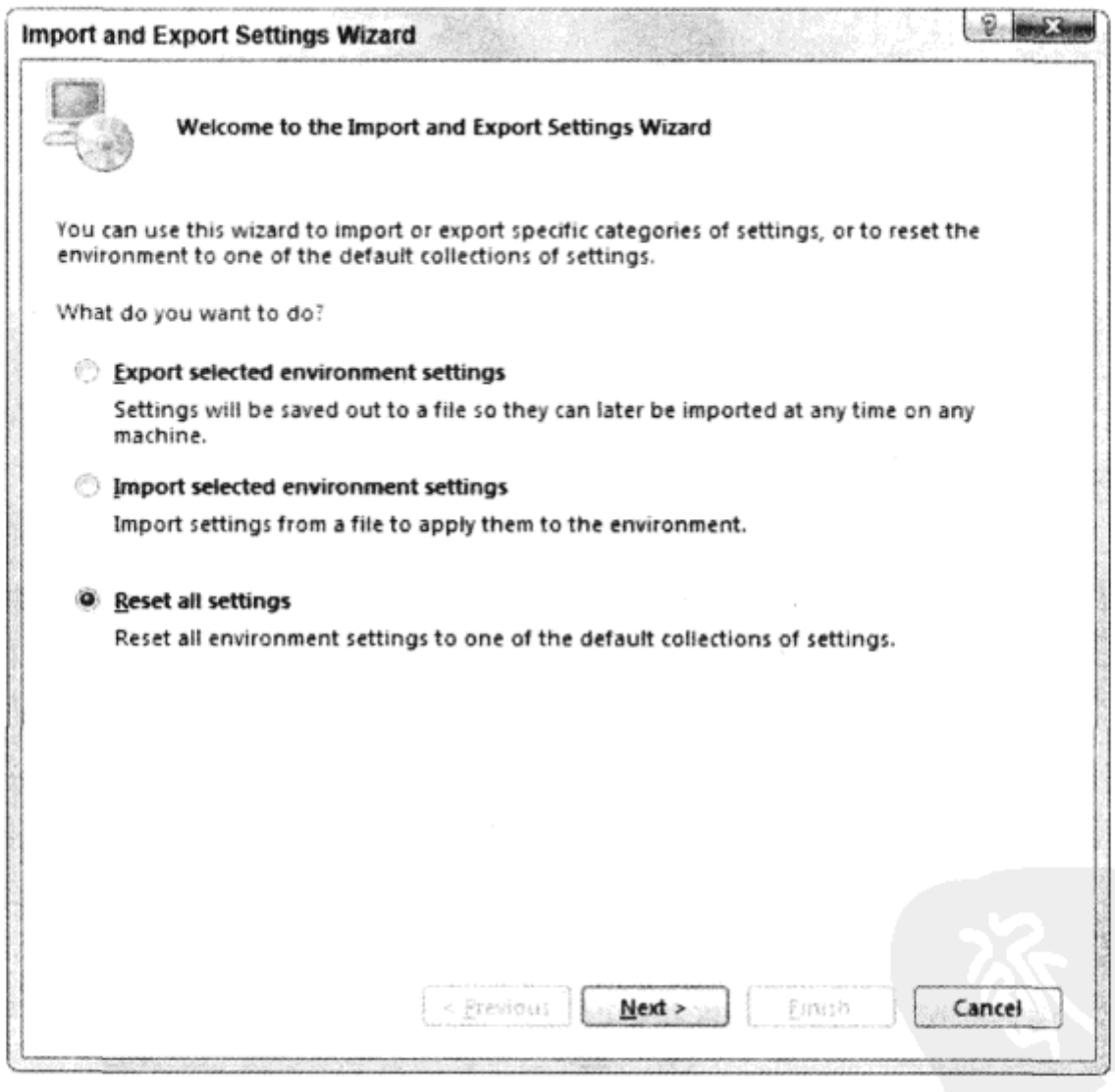


图　2-1

单击 Next 按钮，选择是否要在继续之前保存已有的设置。如果对设置进行了定制，就保存设置，否则就选择 No 按钮，再次单击 Next 按钮。在下一个对话框中，选择 Visual C# Development Settings 选项，如图 2-2 所示。可用的选项可能会变化。

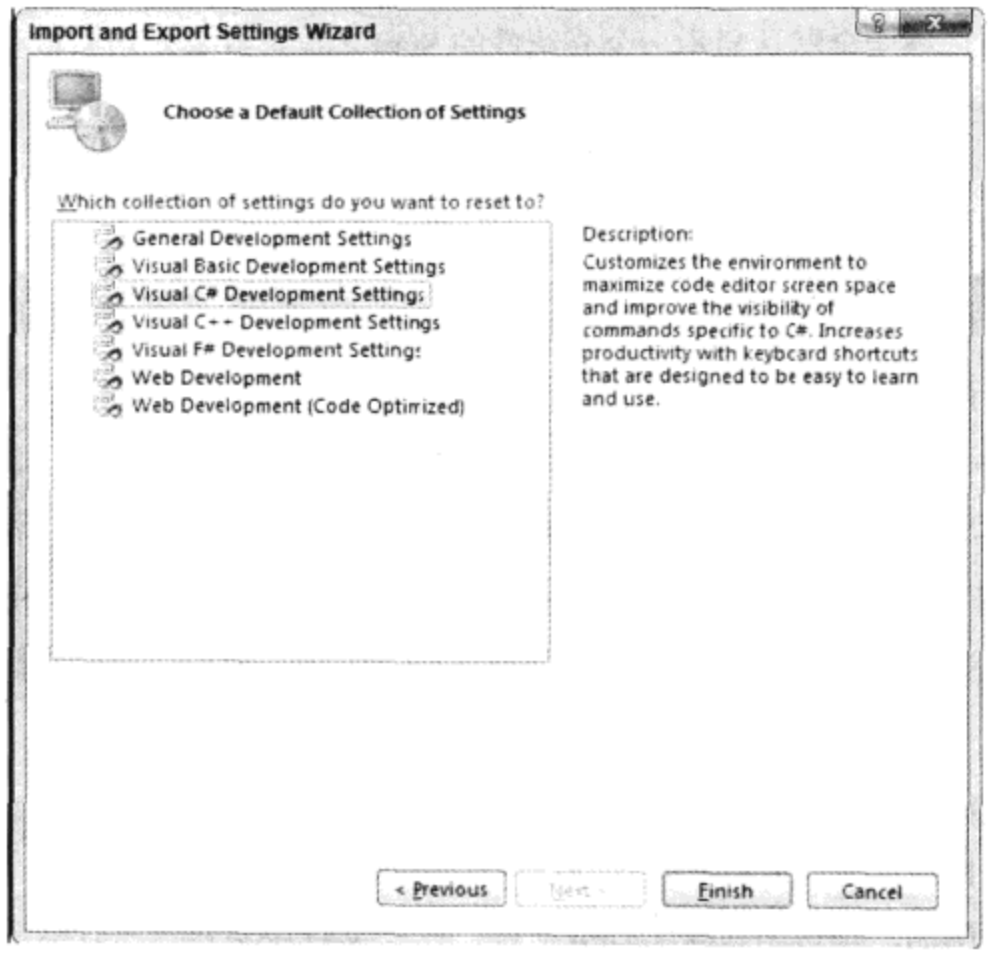


图　2-2

最后单击 Finish 按钮，应用设置。

VS 环境布局是完全可定制的，但默认设置很适合我们。在 C# Developer Settings 设置下，其布局如图 2-3 所示。

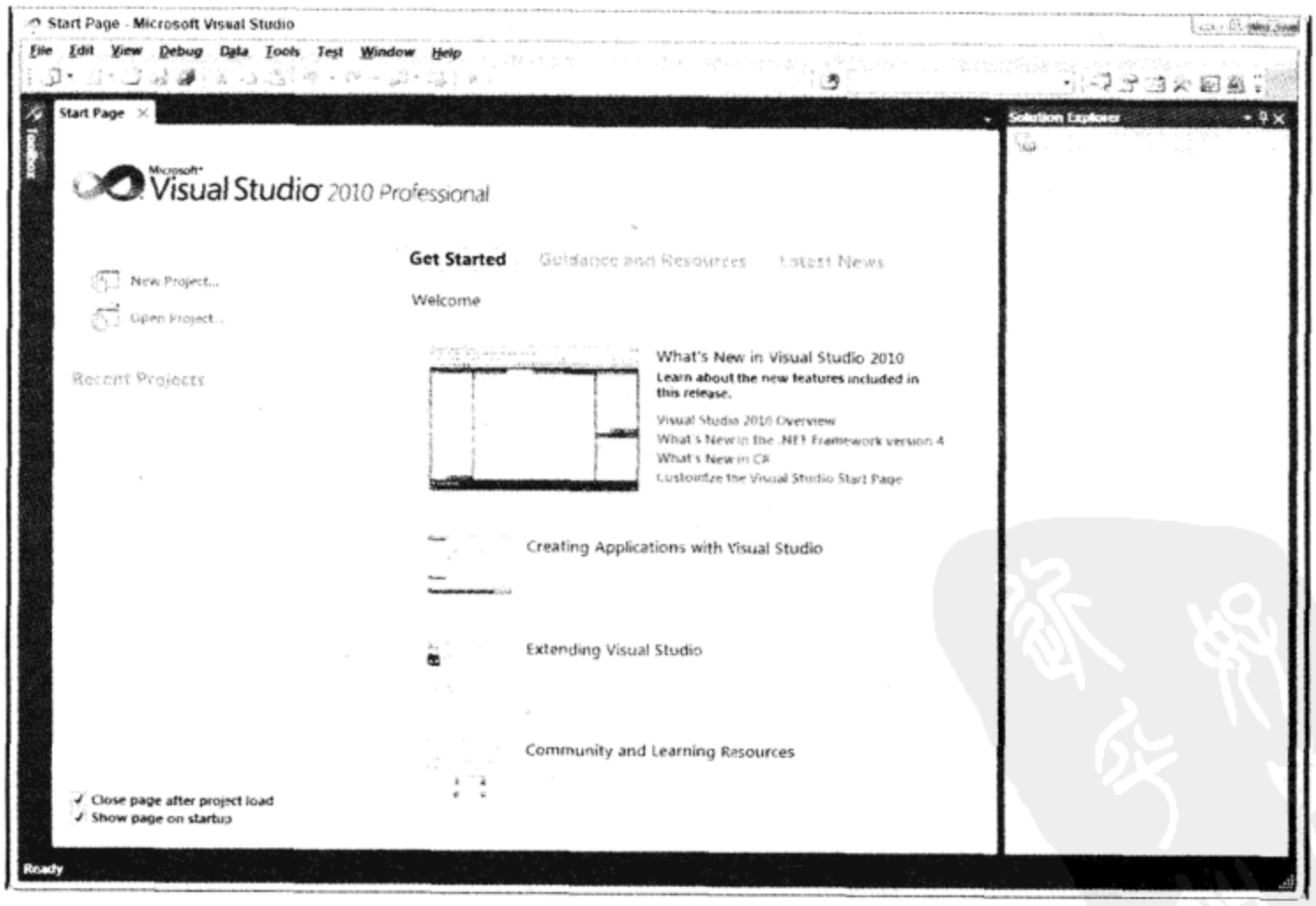


图　2-3

在 VS 启动时，选项卡会默认显示一个介绍性的 Start Page，并显示所有的代码。这个窗口可以包含许多文档，每个文档都有一个选项卡，单击文件名，就可以在文件之间切换。这个窗口也具有其他功能：它可以显示设计用于项目、纯文本文件和 HTML 的图形用户界面以及各种内置于 VS 的工具。本书将陆续介绍它们。

在主窗口的上面，有工具栏和 VS 菜单。这里有几个不同的工具栏，其功能包括保存和加载文件，生成和运行项目，以及调试控件等。在需要使用这些工具栏时将会讨论它们。

下面简要描述 VS 的最常用功能：

- 把鼠标指针放在 Toolbox 上，就会显示 Toolbox 工具栏，它们提供了 Windows 应用程序的用户界面构件等条目。另一个选项卡 Server Explorer 也在这里显示(通过 View Server Explorer 菜单项也可以选择它)，它包含其他许多功能，例如，访问数据源、服务器设置和服务等。
- Solution Explorer 窗口显示当前加载的解决方案的信息。如上一章所述，解决方案是包含一个或多个项目及其配置的Visual Studio 术语。Solution Explorer 窗口显示了解决方案中项目的各种视图，例如，项目中包含了什么文件，这些文件中又包含了什么内容。
- Solution Explorer 窗口之下可以显示 Properties 窗口，该窗口没有显示在图 2-3 中。稍后会看到这个窗口，因为它只在处理项目时才出现(也可以使用View | Properties Window 菜单项切换它)。这个窗口提供了更详细的项目内容视图，允许另外配置单独元素。例如，使用这个窗口可以改变 Windows 窗体中按钮的外观。
- 另一个非常重要的窗口也未出现在图 2-3 中：Error List 窗口。这个窗口可以使用 View | Error List 菜单项打开，它显示了错误、警告和其他与项目有关的信息。这个窗口会持续不断地更新，但其中一些信息只有在编译项目时才出现。

这似乎要理解很多东西，但不必担心，过不了多久就习惯了。下面首先建立第一个示例项目，它将使用上面介绍的许多 VS 元素。

VS 还可以显示许多其他窗口，它们都包含许多信息，有许多功能。其中的一些窗口与上面提及的窗口在相同的位置上，使用选项卡可以切换它们。本书的后面会介绍其中的许多窗口，在详细介绍 VS 环境时，还会发现更多的窗口。

2.1.2 Visual C# 2010 Express Edition

使用 VCE，您不必考虑设置的改变。显然，这个产品并不适用于 VB 编程，所以无需考虑相应的设置。第一次启动 VCE 时，会显示一个非常类似于 VS 的屏幕，如图 2-4 所示。

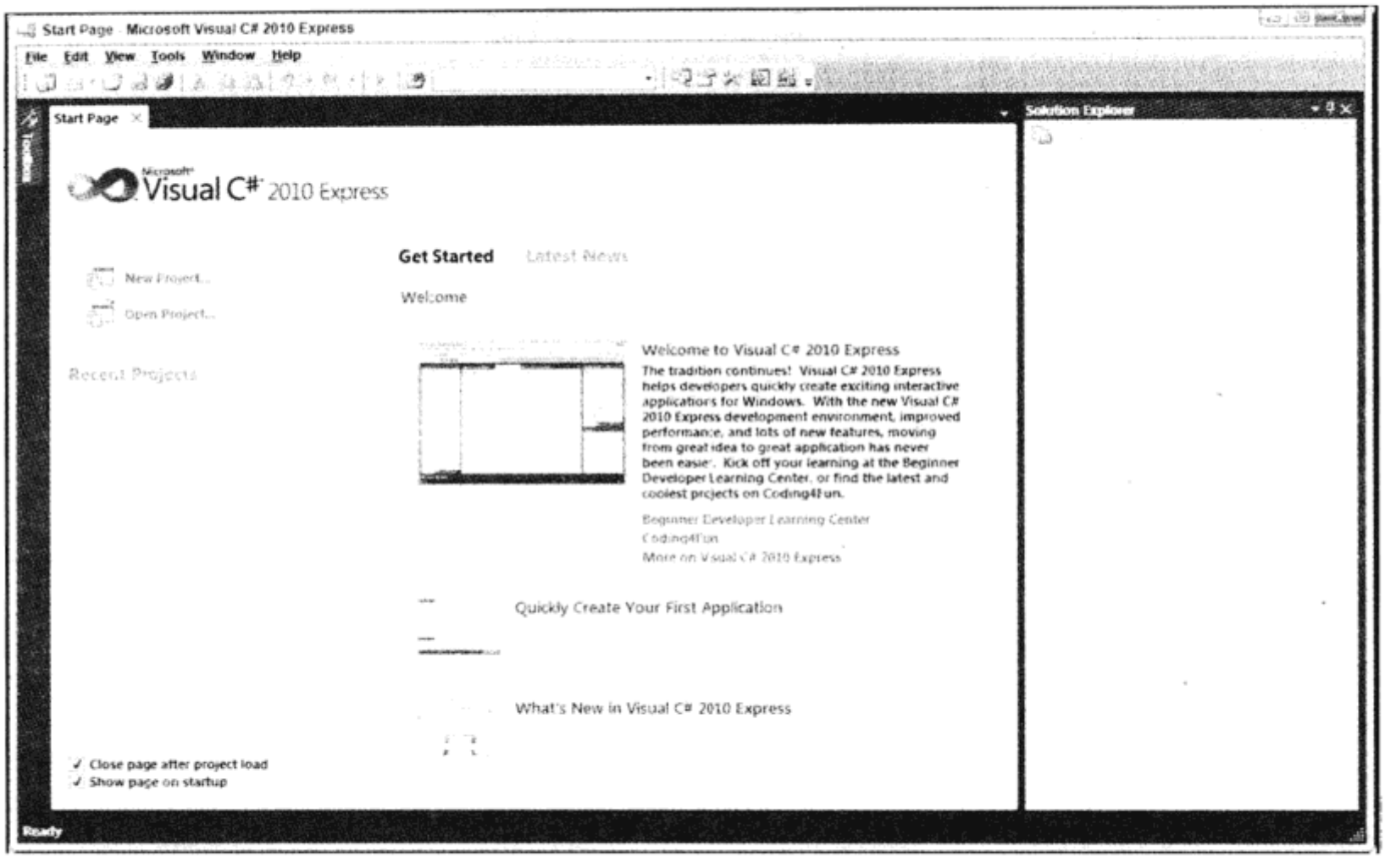


图　2-4

2.2　控制台应用程序

本书将频繁使用控制台应用程序，特别是开始时要使用这类应用程序，所以下面创建一个简单的控制台应用程序。这个示例包含了用于 VS 和 VCE 的指令。

试一试：创建一个简单的控制台应用程序

(1) 在 VS 中选择 File | New | Project...菜单项，或者在 VCE 中选择 File | New Project...，创建一个新的控制台应用程序项目，如图 2-5 和 2-6 所示。

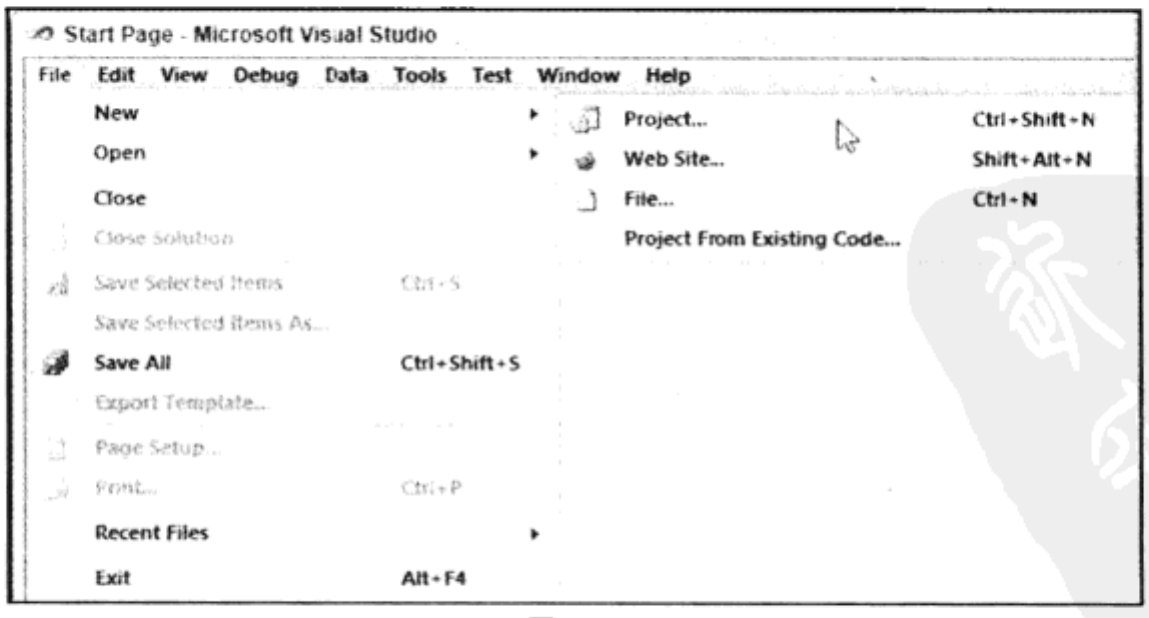


图　2-5

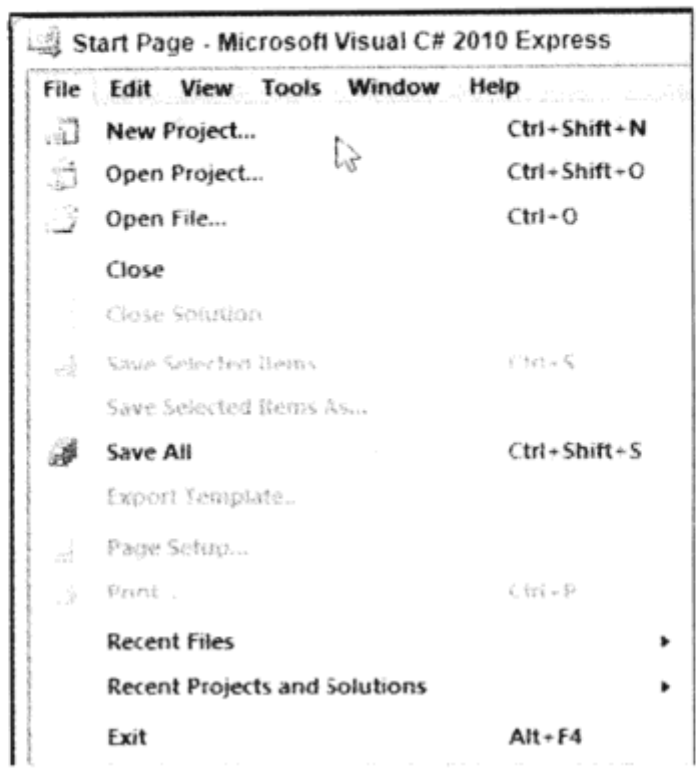


图　2-6

(2) 在 VS 显示窗口的 Installed Templates 窗格中选择 Visual C# 节点，在中间窗格中选择 Console Application 项目类型，如图 2-7 所示。在 VCE 中，只需在 Templates 窗格中选择 Console Application 项目类型，如图 2-8 所示。在 VS 中把 Location 文本框改为 C:\BegVCSharp\Chapter02(如果该目录不存在，会自动创建)。在 VS 和 VCE 中，Name 文本框中的默认文本(ConsoleApplication1)和其他设置不变，如图 2-7 和 2-8 所示。

图　2-7

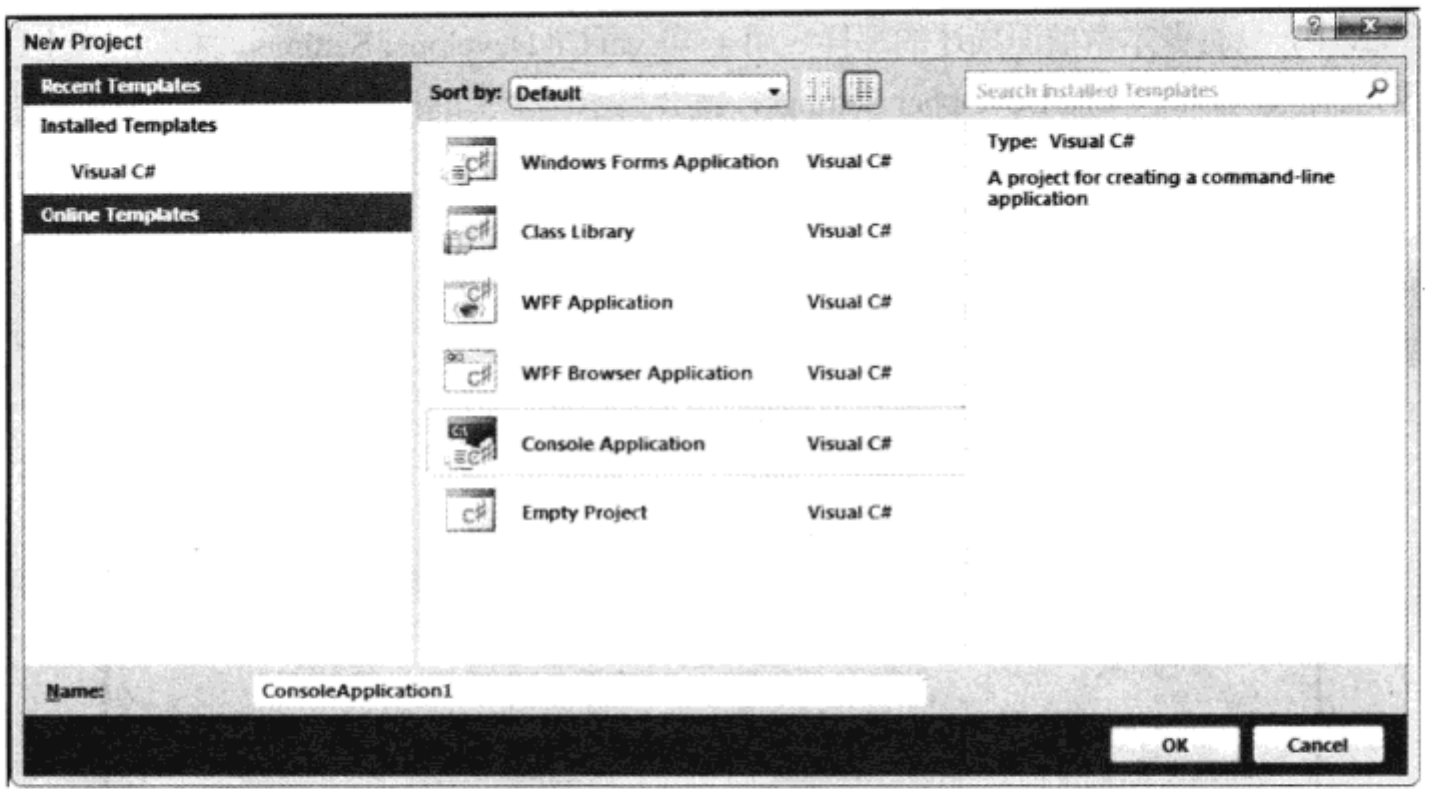


图　2-8

(3) 单击 OK 按钮。

(4) 如果使用的是 VCE，在初始化项目后，单击工具栏上的 Save All 按钮或选择 File 菜单中的 Save All 选项，将 Location 字段设置为 C:\BegVCSharp\Chapter02，单击 Save 按钮。

(5) 初始化项目后，在主窗口显示的文件里添加如下代码行：


```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Output text to the screen.
            Console.WriteLine("The first app in Beginning C# Programming!");
            Console.ReadKey();
        }
    }
}
```

代码段 ConsoleApplication1\Program.cs

(6) 选择 Debug | Start Debugging 菜单项。稍后就应看到如图 2-9 所示的结果。

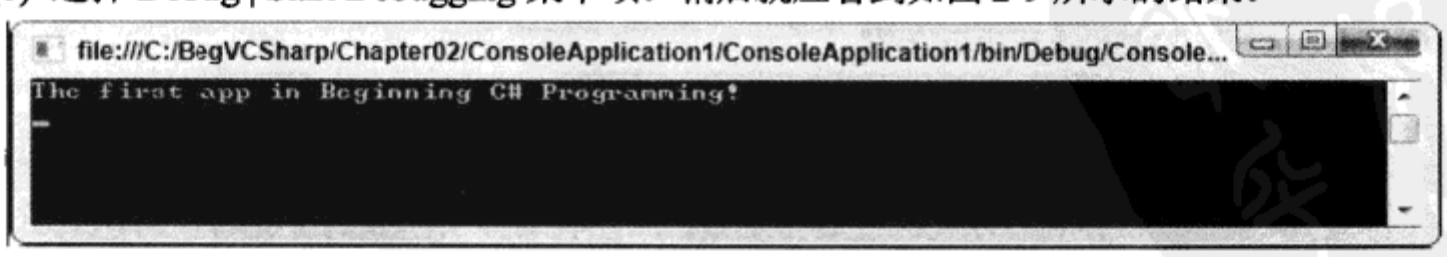


图　2-9

(7) 按下任意键，退出应用程序(首先需要单击控制台窗口，以激活它)。

在 VS 中，只有像本章前面描述的那样应用了 Visual C# Developer Settings，才会出现上述显示。例如，若应用了 Visual Basic Developer Settings，就会显示一个空的控制台窗口，应用程序的输出结果显示在 Immediate 窗口中。在这种情况下，Console.ReadKey()代码也会失败，显示一个错误。如果遇到这个问题，本书中所有示例的最佳解决方案是应用 Visual C# Developer Settings，这样读者看到的结果才会与书中显示的相同。如果问题没有解决，可以打开 Tools | Options 对话框，取消对 Debugging | Redirect all Output Window text to the Immediate Window 选项的选择，如图 2-10 所示。

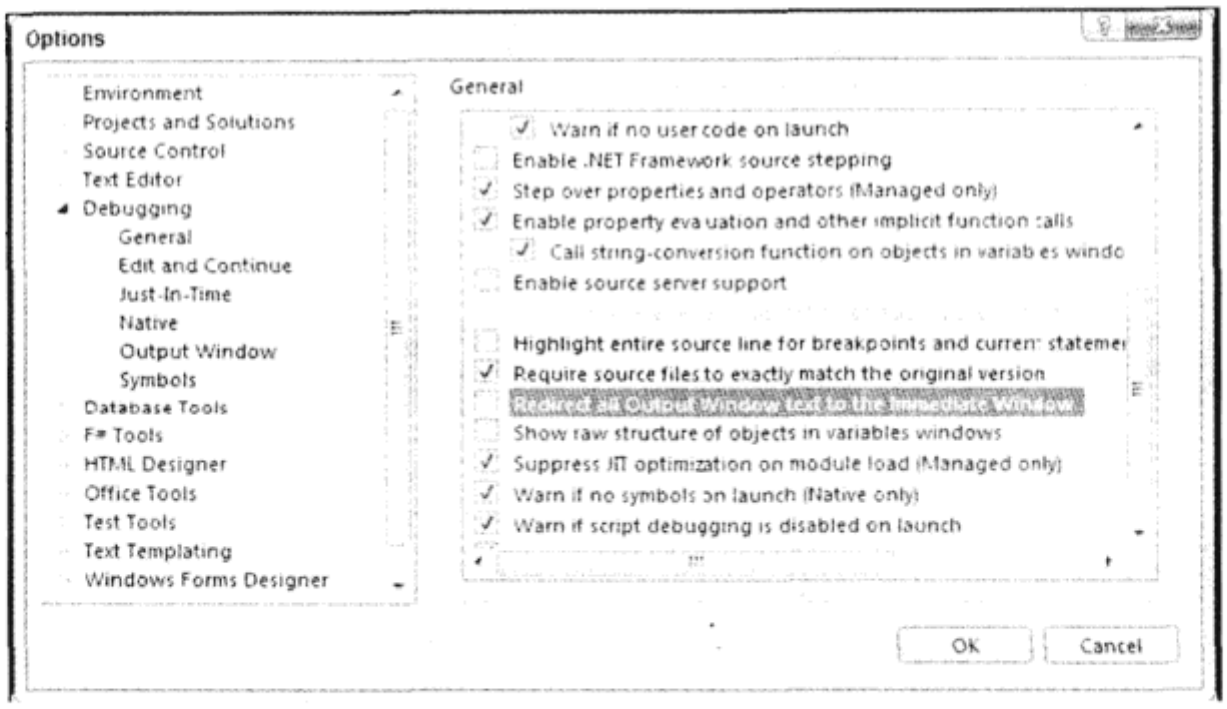


图　2-10

示例的说明

现在不仔细研究这个项目中使用的代码，而关心如何使用开发工具来启动和运行代码。显然，VS 和 VCE 做了许多工作，简化了编译和执行代码的过程。执行这些简单的步骤还有多种方式。例如，创建一个新项目可以像前面那样使用 File | New Project...菜单项，也可以按下 Ctrl+Shift+N 组合键，还可以单击工具栏上相应的图标。

同样，也可以用多种方式编译和执行代码。上面使用的方法是选择 Debug | Start Debugging 菜单项，也可以按下快捷键(F5)，或者使用工具栏中的图标。使用 Debug | Start Without Debugging 菜单项(也可以按下 Ctrl+F5 组合键)还可以以非调试模式运行代码，使用 Build | Build Solution 或 F6 可以编译项目但不运行它(打开或关闭调试功能)。注意，执行项目但不调试，或者生成项目可以使用工具栏中的图标进行，只是这些图标在默认情况下没有显示出来。编译好代码后，在 Windows Explorer 中运行生成的.exe 文件，就可以执行代码。也可以在命令提示窗口中执行，为此，应打开一个命令提示窗口，把目录改为 C:\BegVCSharp\Chapter02\ConsoleApplication1\Console Application1\bin\Debug\，键入 ConsoleApplication1，并按下回车键。

在以后的示例中，我们仅说明“创建一个新的控制台项目”或“执行代码”，用户可以选择自己喜欢的方式执行这些步骤。除非特别声明，否则所有的代码都应在启用调试的情况下运行。另外，本书中的“启动”、“执行”和“运行”等术语都可以互换使用，示例后面的讨论总是假定已经退出了示例中的应用程序。

控制台应用程序会在执行完毕后立即终止，如果直接通过 IDE 运行它们，就无法看到运行的结果。为了解决上例中的这个问题，使用

```
Console.ReadKey();
```

告诉代码在结束前等待按键。后面的示例将多次使用这种技术。前面创建了一个项目，现在详细讨论开发环境中的各个组成部分。

2.2.1　Solution Explorer

首先要讨论的窗口是屏幕右上角的 Solution Explorer，它在 VS 和 VCE 中是相同的(除非特别说明，否则本章分析的所有窗口在 VS 和 VCE 中都是相同的)。这个窗口默认设置为自动隐藏，在该窗口可见时，单击其图钉图标可以把它停靠在屏幕的一条边上。Solution Explorer 窗口与另一个有用的窗口 Class View 在相同的位置上，使用 View | Class View 菜单项就可以显示 Class View 窗口。图 2-11 显示了展开所有节点的这两个窗口(在窗口停靠时，单击窗口底部的选项卡，就可以切换它们)。

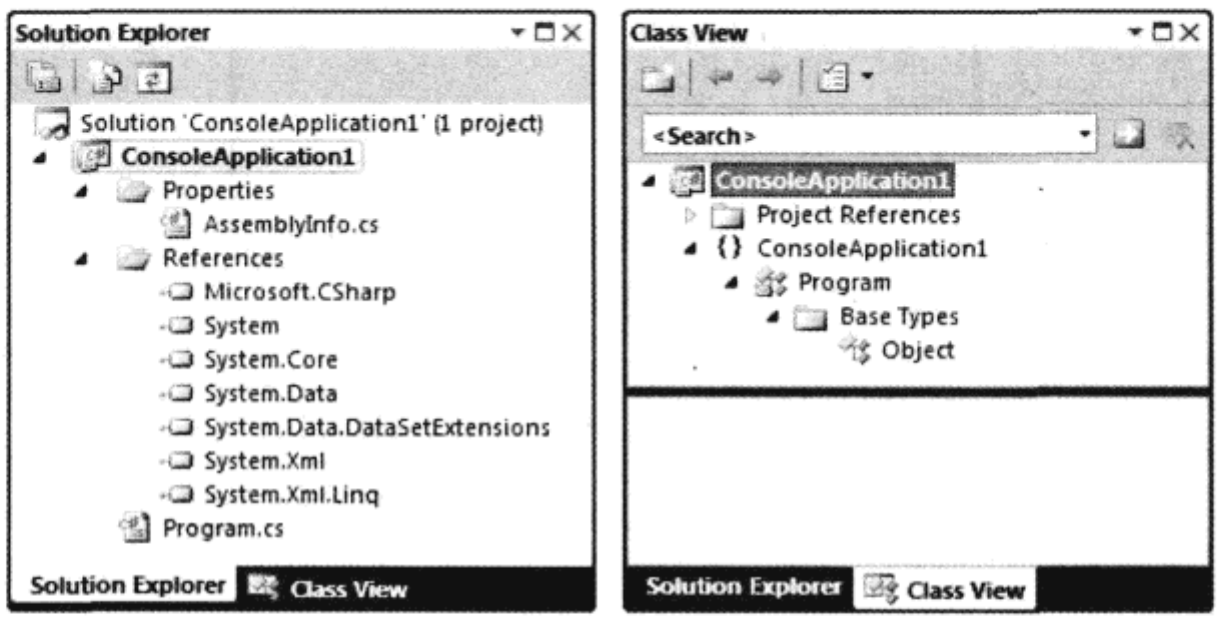


图　2-11

在 VCE 中，只有打开 Expert Settings，才能使用 Class View 窗口。通过 Tools | Settings | Expert Settings 菜单项可以打开 Expert Settings。

Solution Explorer 视图显示了组成 ConsoleApplication1 项目的文件，包括我们在其中添加代码的文件 Program.cs、另一个代码文件 AssemblyInfo.cs 和几项引用。

所有 C#文件都使用.cs 文件扩展名。

此时不需要考虑 AssemblyInfo.cs 文件，它包含项目中目前我们不需要关心的其他信息。

使用这个窗口可以改变主窗口中显示的代码，方法是双击.cs 文件，或右击这些文件并选择 View Code，或选中它们，单击窗口顶部的工具栏按钮。还可以对这些文件执行其他操作，例如，重新命

名它们，或从项目中删除它们等。在该窗口中还可以显示其他类型的文件，例如，项目资源(资源是项目使用的文件，这些文件可能不是 C#文件，如位图图像和声音文件等)。可以通过同一界面处理它们。

References 项包含项目中使用的一个.NET 库列表，这个列表在后面介绍，因为标准引用很适于初学者使用。视图 Class View 显示了项目的另一种视图，可以用于查看刚才创建的代码结构。本书后面将介绍代码结构，现在选择显示 Solution Explorer 视图。单击这些窗口中的文件或其他图标，Properties 窗口的内容就会发生相应变化，如图 2-12 所示。

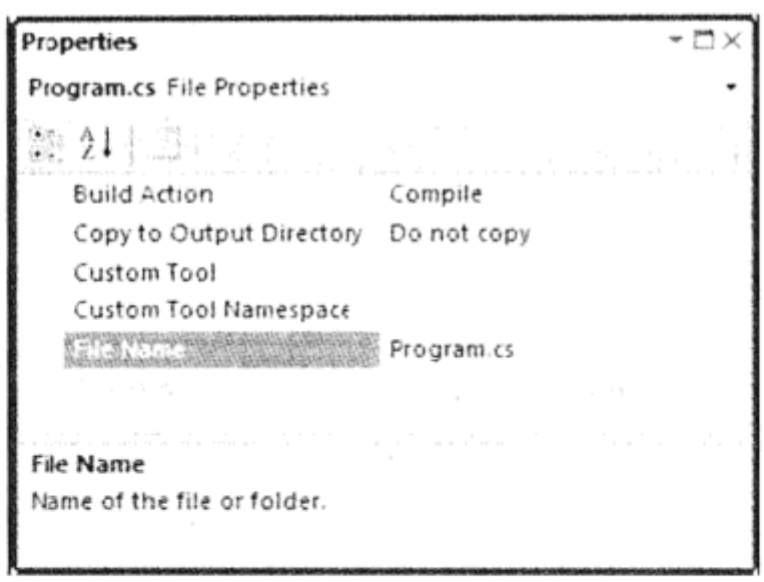


图　2-12

2.2.2　Properties 窗口

使用 View | Properties Window 菜单项就可以打开 Properties 窗口。这个窗口显示了在上述窗口中所选的项的其他信息。例如，选择项目中的 Program.cs 文件，就会显示如图 2-12 所示的视图。这个窗口还显示了其他选中项的信息，例如，用户界面组件(参见本章的“Windows Forms 应用程序”一节)。

通常在 Properties 窗口中对项目的改变会直接影响代码，添加代码行，或改变文件中的内容。对于一些项目来说，通过这个窗口来操作与手动修改代码所花的时间是相同的。

2.2.3　Error List 窗口

当前 Error List 窗口(View | Error List)没有显示什么有趣的信息，这是因为应用程序没有错误。但是这的确是一个非常有用的窗口。下面进行测试，从上一节添加的一行代码中删除分号。过一会儿，就会看到如图 2-13 所示的结果。

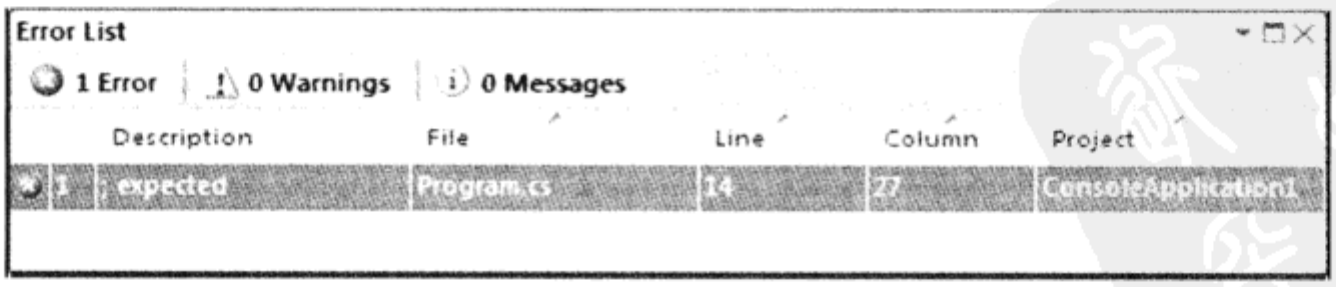


图　2-13

这次项目不会编译。

第 3 章介绍 C#语法后，你就会明白分号在大多数代码行的末尾，它是必不可少的。

这个窗口有助于根除代码中的错误，因为它会跟踪我们的工作，编译项目。如果双击该窗口中显示的错误，光标就会跳到源代码中出现错误的地方(如果包含错误的源文件没有打开，将被打开)，这样就可以快速更正错误。代码中有错误的一行会出现红色的波浪线，便于我们快速扫描源代码，找出错误。

注意错误的位置用一个行号来指定。在默认情况下，行号不会显示在 VS 文本编辑器中，但其实有必要显示它。为此，需要单击 Tools | Options 菜单项，选中 Options 对话框中的 Line numbers 复选框。该复选框位于 Text Editor | C# | General 类别中，如图 2-14 所示。

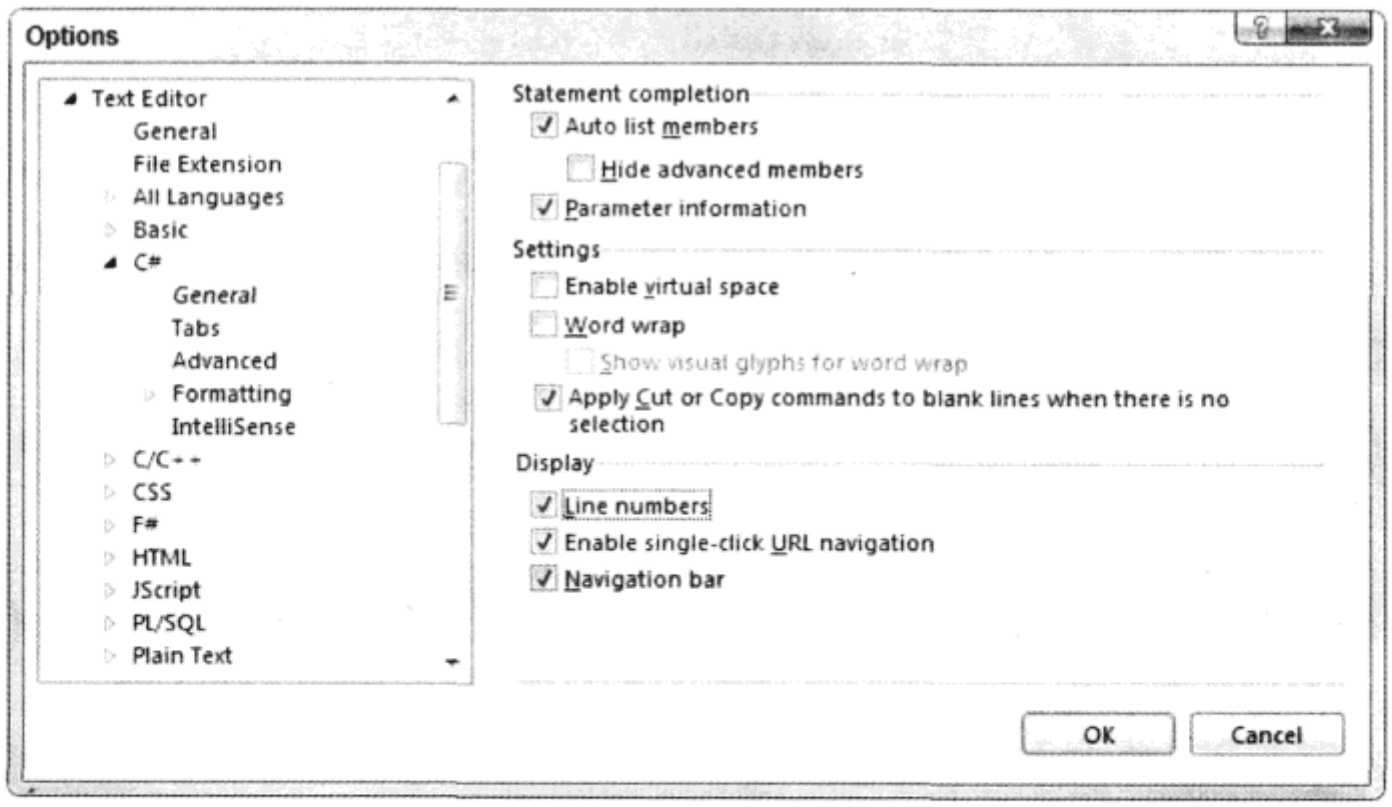


图　2-14

在 VCE 中，只有选择 Show All Settings 才能使这个选项可用，选项列表与图 2-14 略有不同。

这个对话框中包含许多有用的选项，本书将使用其中的几个。

2.3　Windows Forms 应用程序

通常，如果将代码当作 Windows 应用程序的一部分运行来描述代码，要比通过控制台窗口或命令提示符简单一些。下面用用户界面构件来组合一个用户界面。

下面的示例介绍建立用户界面的基础知识，说明如何启动和运行 Windows 应用程序，但并不详

细讨论应用程序实际完成的工作。本书的后面会详细研究 Windows 应用程序。

试一试：创建一个简单的 Windows 应用程序

(1) 在以前的位置(C:\BegVCSharp\Chapter02，如果使用的是 VCE，就在创建该项目后把它保存在这个位置)创建一个类型为 Windows Forms Application(VS 或 VCE)的新项目，其默认名称是 WindowsFormsApplication1。如果使用的是 VS，而第一个项目仍处于打开状态，则应选择 Create New Solution 选项来启动一个新解决方案，这些设置如图 2-15 和图 2-16 所示。

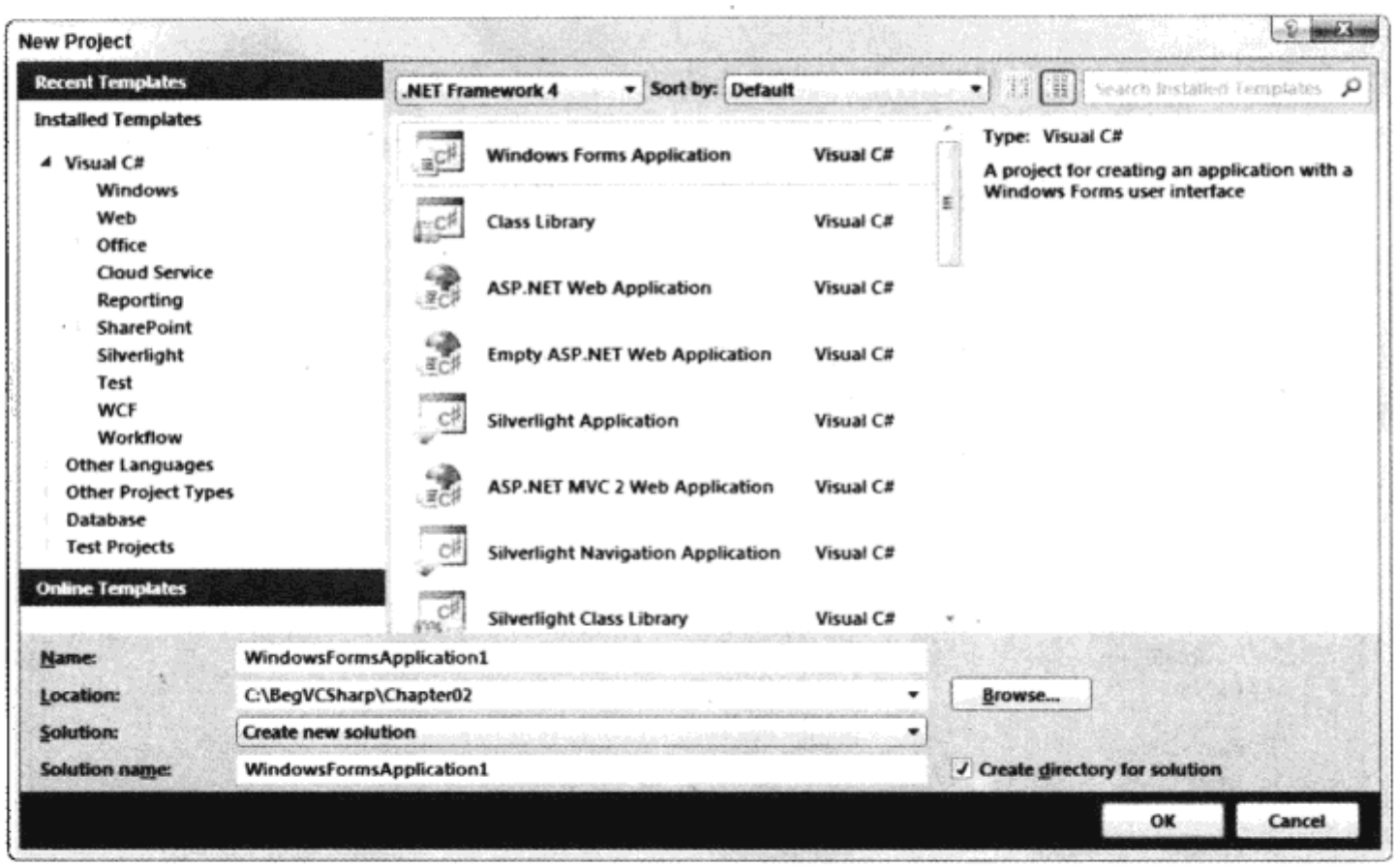


图　2-15

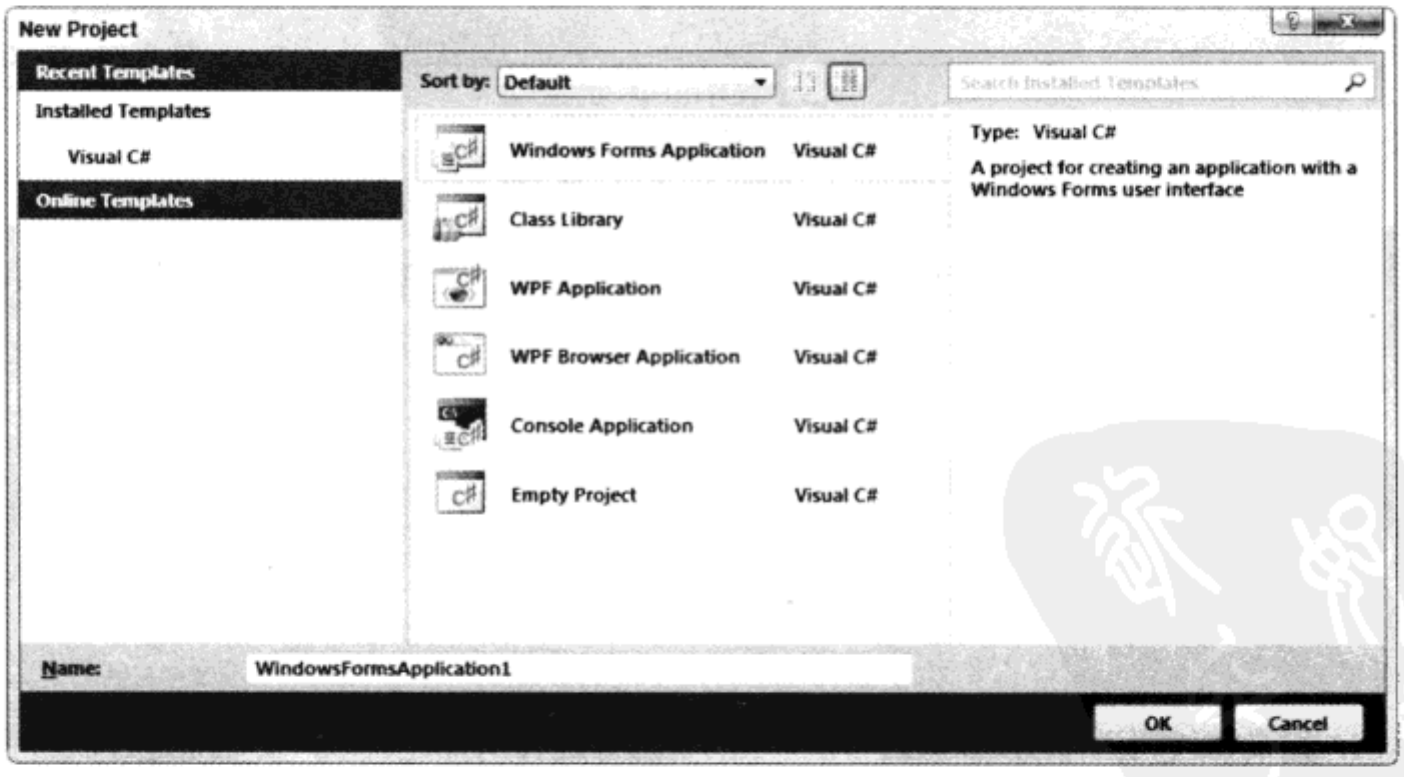


图　2-16

(2) 单击 OK 按钮，创建项目后，应该会看到一个空白的 Windows 窗体。把鼠标指针移到屏幕

左边的 Toolbox 栏上，然后移到 All Windows Forms 选项卡上的 Button 选项，在该选项上双击，就会在应用程序的主窗体(Form1)上添加一个按钮。

(3) 双击刚才添加到窗体中的按钮。

(4) 现在应显示 Form1.cs 中的 C#代码。进行如下修改(为了简短起见，这里只显示了文件中的部分代码)：

可从 wrox.com 下载源代码

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("The first Windows app in the book!");
}
```

代码段 WindowsFormsApplication1\Form1.cs

(5) 运行应用程序。

(6) 单击显示出来的按钮，打开一个消息对话框，如图 2-17 所示。

图 2-17

(7) 单击 OK。像每个标准 Windows 应用程序那样，单击右上角的 X 图标，退出应用程序。

示例的说明

IDE 又一次自动完成了许多工作，使我们能不费吹灰之力就可以完成一个实用的 Windows 应用程序的创建。刚才创建的应用程序与其他窗口的行为方式相同——可以移动、重新设置其大小、最小化等。我们不必编写任何代码，它就可以工作。我们添加的按钮也是这样。双击按钮，IDE 就知道我们想添加一些代码，当运行应用程序时，用户单击该按钮，就执行我们已经编写好的代码。只要提供了这段代码，就可以得到按钮单击的所有功能。

当然，Windows 应用程序不仅限于带有按钮的普通窗体。如果看看从中选择 Button 选项的工具栏，就会看到一整套用户界面构件，其中一些用户可能很熟悉。本书在其他地方将使用其中的大多数用户界面构件，它们使用起来都非常简单，可以节省许多时间和精力。

应用程序的代码在 Form1.cs 中，看起来并不比上一节提供的代码复杂多少，Solution Explorer 窗口中其他文件的代码也不太复杂。开发环境生成的代码在默认情况下是隐藏的，它们都与窗体上控件的布局有关。这就是为什么可以在主窗口的设计视图中查看代码的原因，而设计视图正是这些布局代码的可视化转换。按钮就是可以使用的一种控件，同样，也可以使用 Toolbox 上 Windows Forms 部分中的其他 UI 构件。

下面把按钮作为一个控件示例，详细解释一下。使用主窗口中的选项卡，切换回窗体的设计视图，单击按钮，选择它。此时，屏幕右下角的 Properties 窗口就会显示按钮控件的属性(控件的属性非常类似于上例中文件的属性)，确保应用程序当前并未运行，向下滚动到 Text 属性，它目前设置为 button1，把其值改为 Click Me，如图 2-18 所示。

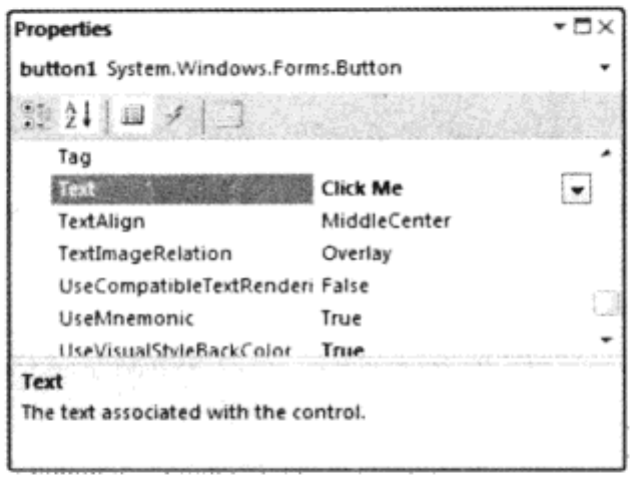


图　2-18

在 Form1 中，按钮上的文本应反映这个变化。

这个按钮有许多属性，从按钮颜色和大小的简单格式，到某些模糊设置(如数据绑定设置，它可以建立与数据库的联系)，应有尽有。如上例所述，改变属性通常会直接改变代码，这也不例外。但如果切换回 Form1.cs 的代码视图，就会发现代码没有变化。

要查看修改过的代码，需要查看上面提及的隐藏代码。为了查看包含代码的文件，需要在 Solution Explorer 窗口中扩展 Form1.cs 节点。打开 Form1.Designer.cs 节点，双击这个文件，就可以查看其中的内容。

匆匆一瞥可能注意不到代码中哪些地方反映了按钮属性的改变，这是因为 C#代码中处理窗体上控件的布局和格式化的部分被隐藏了(毕竟，如果有了结果的图形显示，就几乎不需要查看代码了)。

VS 和 VCE 使用代码突出显示系统来显示这部分 C#代码，如图 2-19 所示。

Form1.Designer.cs*

WindowsFormsApplication1.Form1　　components

```
1   namespace WindowsFormsApplication1
2   {
3       partial class Form1
4       {
5           /// <summary>
6           /// Required designer variable.
7           /// </summary>
8           private System.ComponentModel.IContainer components;
9
10          /// <summary>
11          /// Clean up any resources being used.
12          /// </summary>
13          /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
14          protected override void Dispose(bool disposing)
15          {
16              if (disposing && (components != null))
17              {
18                  components.Dispose();
19              }
20              base.Dispose(disposing);
21          }
22
23          Windows Form Designer generated code
57
58          private System.Windows.Forms.Button button1;
59      }
60  }
```

100 %

图　2-19

注意在代码的左边(如果已显示行号，就在行号的旁边)，有一些灰色的线条，其上有+和 - 号的方框。这些方框可用于展开和折叠代码区域。靠近文件的底部，有一个带+号的方框，它对应代码主体中的“Windows Form Designer generated code”。这个标签的基本含义是“这里有一些 VS 生成的代码，用户不需知道”。但如果愿意，也可以进行查看，并通过修改按钮属性来了解所做的事情。只要单击带有+号的方框，代码就会显示出来，在某处应该会看到如下代码：

```
this.button1.Text = "Click Me";
```

不必过多地考虑这里使用的语法，我们可以看到在 Properties 窗口中键入的文本直接显示在代码中。

在编写代码时，这种突出显示代码的方式带来了极大的方便。我们可以展开和折叠其他许多区域(而不仅仅是正常情况下隐藏的代码段)。查看一本书的目录，有助于快速了解该书的内容；查看一组折叠的代码段，则非常便于浏览大量的 C#代码。

2.4　小结

本章介绍了本书后面所使用的一些工具，快速浏览了 Visual Studio 2010 和 Visual C# 2010 Express 开发环境，并使用它们建立了两种类型的应用程序。其中比较简单的是控制台应用程序，它足以满足我们的大多数需要，便于我们集中精力学习 C#编程的基础知识。Windows 应用程序比较复杂，但其可视化程度比较高，对于习惯了 Windows 环境的人来说，使用起来也比较直观。

知道如何创建简单的应用程序，就可以真正开始学习 C#了。本书后面的章节将介绍 C#的基本语法和程序结构，之后讨论更高级的面向对象方法。学习了这些内容后，就可以开始了解如何使用 C#访问.NET Framework 的功能了。

在后面的章节中，除非特别说明，否则指令都是针对 VCE 的，但如本章所述，这些指令很容易修改，以用于 VS，读者可以使用自己喜欢的任何 IDE。

2.5　本章要点

主　题	重 要 概 念
Visual Studio 2010 设置	本书需要在第一次运行 VS 时选择 C# Development Settings 选项，或者重置它们
控制台应用程序	控制台应用程序是简单的命令行应用程序，本书主要用它演示技术。在 VS 或 VCE 中创建新项目时，使用 Console Application 模板就会创建新的控制台应用程序。要在调试模式下运行项目，可以使用 Debug | Start Debugging 菜单项或者按下 F5
IDE 窗口	项目内容显示在 Solution Explorer 窗口中。选中项的属性显示在 Properties 窗口中。错误显示在 Error List 窗口中
Windows 窗体应用程序	Windows 窗体应用程序具备标准桌面应用程序的外观和操作方式，包括最大化、最小化和关闭应用程序等大家熟悉的图标。它们是在 New Project 对话框中用 Windows Forms 模板创建的


PDG

第 3 章

变量和表达式

本章内容：

- C#的基本语法
- 变量及其用法
- 表达式及其用法

要想高效地使用 C#，就一定要理解创建计算机程序时真正需要做些什么。计算机程序最基本的描述也许是一系列处理数据的操作，即使对于最复杂的示例，例如 Microsoft Office 套装软件之类的大型多功能 Windows 应用程序，这个论述也正确。应用程序的用户虽然看不到它们，但这些操作总是在后台进行。

为了进一步解释它，考虑一下计算机的显示单元。我们常常比较熟悉屏幕上的内容，很难不把它想像为“移动的图片”。但实际上，我们看到的仅是一些数据的显示结果，其最初的形式是存储在计算机内存中的 0 和 1 数据流。因此我们在屏幕上进行的任何操作，无论是移动鼠标指针，单击图标，或在字处理器上输入文本，都会改变内存中的数据。

当然，还可以利用一些较简单的情形来说明这一点。如果使用计算器应用程序，就要提供数字，对这些数字执行操作，就像用纸和笔计算数字一样，但使用程序会快得多。

如果计算机程序是对数据执行操作，则说明我们需要以某种方式来存储数据，需要某些方法来处理它们。这两种功能是由变量和表达式提供的，本章将探究它们的含义。

在开始之前，应该首先了解一下 C#编程的基本语法，因为我们需要一个环境来学习使用 C#语言中的变量和表达式。

3.1 C#的基本语法

C#代码的外观和操作方式与 C++和 Java 非常类似。初看起来，其语法可能比较混乱，不像书面英语和其他语言那么直观。但实际上，在 C#编程中，使用的样式是比较清晰的，不用花太多力气就可以编写出便于阅读的代码。

与其他语言的编译器不同，C#编译器不考虑代码中的空格、回车符或 tab 字符(这些字符统称为空白字符)。这样格式化代码时就有很大的自由度，但遵循某些规则将有助于阅读代码。

C#代码由一系列语句组成，每个语句都用一个分号来结束。因为空白被忽略，所以一行可以有多个语句，但从可读性的角度来看，通常在分号的后面加上回车符，这样就不会在一行上放置多个语句了。但语句放在多个行上是可以的(也比较常见)。

C#是一种块结构的语言，所有的语句都是代码块的一部分。这些块用花括号来界定("{"和"}")，代码块可以包含任意多行语句，或者根本不包含语句。注意花括号字符不需要附带分号。

所以，简单的 C#代码块如下所示：

```
{
   <code line 1, statement 1>;
   <code line 2, statement 2>
      <code line 3, statement 2>;
}
```

其中<code line *x*, statement *y*>部分并非真正的 C#代码，而是用这个文本作为 C#语句的占位符。在这段代码中，第 2、3 行代码是同一个语句的一部分，因为在第 2 行的末尾没有分号。

下面的简单示例还使用了缩进格式，提高了 C#代码的可读性。这是一个标准做法，实际上在默认情况下 VS 会自动缩进代码。一般情况下，每个代码块都有自己的缩进级别，即它向右缩进了多少。代码块可以互相嵌套(即块中可以包含其他块)，而被嵌套的块要缩进得多一些。

```
{
   <code line 1>;
   {
      <code line 2>;
      <code line 3>;
   }
   <code line 4>;
}
```

前面代码行的续行通常也要缩进得多一些，如上面第一个示例中的第 3 行代码所示。

在能通过 Tools | Options 访问的 VCE 或 VS Options 对话框中，显示了 VCE 用于格式化代码的规则。在 Text Editor | C# | Formatting 节点的子类别下，包含了其中的很多规则。此处的大多数设置都反映了还没有讲述的 C#部分，但如果以后要修改设置，以更适合自己的个性化样式，就可以回过头来看看这些设置。在本书中，为了简洁起见，所有代码段都使用默认设置来格式化。

当然，这种样式并不是强制的。但如果不使用它，读者在阅读本书时会很快陷入迷茫之中。

在 C#代码中，另一个常见的语句是注释。注释并非严格意义上的 C#代码，但代码最好有注释。注释就是自解释，即给代码添加描述性文本(用英语、法语、德语、外蒙古语等)，编译器会忽略这些内容。在开始处理冗长的代码段时，注释可用于为正在进行的工作添加提示，例如"这行代码要求用户输入一个数字"，或"这段代码由 Bob 编写"。

C#添加注释的方式有两种。可以在注释的开头和结尾放置标记，也可以使用一个标记，其含义是"这行代码的其余部分是注释"。在 C#编译器忽略回车符的规则中，后者是一个例外，但这是一

种特殊情况。

要使用第一种方式标记注释，可以在注释开头加上/*字符，在末尾加上*/字符。这些注释符号可以在单独一行上，也可以在不同的行上，注释符号之间的所有内容都是注释。注释中唯一不能输入的是*/，因为它会被看作注释结束标记。所以下面的语句是正确的：

```
/* This is a comment */

/* And so...

              ... is this! */
```

但以下语句会产生错误：

```
/* Comments often end with "*/" characters */
```

注释结束符号后的内容("*/"后面的字符)会被当作 C#代码，因此产生错误。

另一种添加注释的方法是用//开始一个注释，在其后可以编写任何内容，只要这些内容在一行上即可。下面的语句是正确的：

```
// This is a different sort of comment.
```

但下面的语句会失败，因为第二行代码会解释为 C#代码：

```
//So is this,
  but this bit isn't.
```

这类注释可用于语句的说明，因为它们都放在一行上：

```
<A statement>;        // Explanation of statement
```

前面讲过，有两种给 C#代码添加注释的方法。但在 C#中，还有第三类注释，严格地说，这是//语法的扩展。它们都是单行注释，用三个/符号来开头，而不是两个。

```
/// A special comment
```

正常情况下，编译器会忽略它们，就像其他注释一样，但可以配置 VS，在编译项目时，提取这些注释后面的文本，创建一个特殊格式的文本文件，该文件可用于创建文档说明书。为了创建文档说明书，注释必须遵循 XML 文档的规则。本书不讨论这个主题，如果读者有时间，这是一个很值得学习的内容。

特别要注意的一点是，C#代码是区分大小写的。与其他语言不同，必须使用正确的大小写形式输入代码，因为简单地用大写字母代替小写字母会中断项目的编译。看看下面这行代码，它在第 2 章的第一个示例中使用：

```
Console.WriteLine("The first app in Beginning C# Programming!");
```

C#编译器能理解这行代码，因为 Console.WriteLine()命令的大小写形式是正确的。但是，下面的语句都不能工作：

```
console.WriteLine("The first app in Beginning C# Programming!");
CONSOLE.WRITELINE("The first app in Beginning C# Programming!");
Console.Writeline("The first app in Beginning C# Programming!");
```

这里使用的大小写形式是错误的，所以 C#编译器不知道我们要做什么。幸好，VCE 在代码的键入方面提供了许多帮助，在大多数情况下，它都知道(程序也知道)我们要做什么。在键入代码的过程中，VS 会推荐用户可能要使用的命令，并尽可能纠正大小写问题。

3.2　C#控制台应用程序的基本结构

下面看看第 2 章的控制台应用程序示例(ConsoleApplication1)，并研究一下它的结构。其代码如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         // Output text to the screen.
         Console.WriteLine("The first app in Beginning C# Programming!");
         Console.ReadKey();
      }
   }
}
```

可以立即看出，上一节讨论的所有语法元素这里都有。其中有分号、花括号、注释和适当的缩进。目前看来，这段代码中最重要的部分如下所示：

```
static void Main(string[] args)
{
   // Output text to the screen.
   Console.WriteLine("The first app in Beginning C# Programming!");
   Console.ReadKey();
}
```

在运行控制台应用程序时，就会运行这段代码，更确切地讲，是运行花括号中的代码块。如前所述，注释行不做任何事情，包含它们只为了简明而已。其他两行代码在控制台窗口中输出一些文本，并等待一个响应。但目前我们还不需要关心它的具体机制。

这里要注意一下如何实现第 2 章介绍的代码突出显示功能，这对于 Windows 应用程序来说比较重要，因为它是一个非常有用的特性。要实现该功能，需要使用#region 和#endregion 关键字，来定义可以展开和折叠的代码区域的开头和结尾。例如，可以修改为 ConsoleApplication1 生成的代码，如下所示：

```
#region Using directives

using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
#endregion
```

这样就可以把这些代码行折叠为一行，以后要查看其细节时，可以再次展开它。这里包含的 using 语句和其下的 namespace 语句将在本章后面予以解释。

以#开头的任意关键字实际上都是一个预处理指令，严格地说并不是 C#关键字。除了这里描述的#region 和#endregion 关键字之外，其他关键字都相当复杂，用法也比较专业。所以，这是一个读者通读全书后才能探究的主题。

现在不必考虑示例中的其他代码，因为本书前几章仅解释 C#的基本语法，至于应用程序进行 Console.WriteLine()调用的具体方式，则不在我们的考虑之列。以后会阐述此附加代码的重要性。

3.3　变量

如前所述，变量关系到数据的存储。实际上，可以把计算机内存中的变量看作架子上的盒子。在这些盒子中，可以放入一些东西，再把它们取出来，或者只是看看盒子里是否有东西。变量也是这样，数据可放在变量中，可以从变量中取出数据或查看它们。

尽管计算机中的所有数据事实上都是相同的东西(一组 0 和 1)，但变量有不同的内涵，称为类型。下面再使用盒子来类比，盒子有不同的形状和尺寸，某些东西只能放在特定的盒子中。建立这个类型系统的原因是，不同类型的数据需要用不同的方法来处理。变量限定为不同的类型，可以避免混淆。例如，组成数字图片的 0 和 1 序列与组成声音文件的 0 和 1 序列，其处理方式是不同的。

要使用变量，需要声明它们。即给变量指定名称和类型。声明变量后，就可以把它们用作存储单元，存储所声明的数据类型的数据。

声明变量的 C#语法是指定类型和变量名，如下所示：

```
<type> <name>;
```

如果使用未声明的变量，代码将无法编译，但此时编译器会告诉我们出现了什么问题，所以这不是一个灾难性错误。另外，使用未赋值的变量也会产生一个错误，编译器会检测出这个错误。

可以使用的变量类型是无限多的。其原因是可以自己定义类型，存储各种复杂数据。尽管如此，总有一些数据类型是每个人都要使用的，例如，存储数值的变量。因此，我们应了解一些简单的预定义类型。

3.3.1　简单类型

简单类型就是组成应用程序中基本构件的类型，例如，数值和布尔值(true 或 false)。简单类型与复杂类型不同，没有子类型或特性。大多数简单类型都是存储数值的，初看起来有点奇怪，肯定只需要一种类型来存储数值吗？

数值类型过多的原因是在计算机内存中，把数字作为一系列的 0 和 1 来存储的机制。对于整数值，用一定的位(单个数字，可以是 0 或 1)来存储，用二进制格式来表示。以 N 位来存储的变量可以表示任何介于 0 到 (2^N-1)之间的数。大于这个值的数因为太大，所以不能存储在这个变量中。

例如，有一个变量存储了 2 位，在整数和表示该整数的位之间的映射应如下所示：

```
0 = 00
1 = 01
2 = 10
3 = 11
```

如果要存储更多数字，就需要更多的位(例如，3 位可以存储 0~7 的数)。

这样得到的结论是要存储每个可以想像得到的数，就需要非常多的位，这并不适合PC。即使可以用足够多的位来表示每一个数，变量使用这些位来存储它，其效率也非常低下，例如，只需要存储从 0~10 之间的数(因为存储器被浪费了)。其实 4 位就足够了，可以用相同的内存空间存储这个范围内的更多数值。

相反，许多不同的整数类型可以用于存储不同范围的数值，占用不同的内存空间(至多64 位)，这些类型如表 3-1 所示。

表 3-1

类　型	别　名	允 许 的 值
sbyte	System.SByte	在 －128~127 之间的整数
byte	System.Byte	在 0~255 之间的整数
short	System.Int16	在－32 768~32 767 之间的整数
ushort	System.UInt16	在 0 ~65 535 之间的整数
int	System.Int32	在－2 147 483 648~ 2 147 483 647 之间的整数
uint	System.UInt32	在 0~4 294 967 295 之间的整数
long	System.Int64	在－9 223 372 036 854 775 808~9 223 372 036 854 775 807 之间的整数
ulong	System.UInt64	在 0~18 446 744 073 709 551 615 之间的整数

这些类型中的每一种都利用了.NET Framework 中定义的标准类型。如第 1 章所述，使用标准类型可以在语言之间交互操作。在 C#中这些类型的名称是 Framework 中定义的类型的别名，表 3-1 列出了这些类型在.NET Framework 库中的名称。

一些变量名称前面的“u”是 unsigned 的缩写，表示不能在这些类型的变量中存储负数，参见该表中的“允许的值”一列。

当然，还需要存储浮点数，它们不是整数。可以使用的浮点数变量类型有 3 种：float、double 和 decimal。前两种可以用+/－$m\times2^e$的形式存储浮点数，m 和 e 的值因类型而异。Decimal 使用另一种形式：+/－$m\times10^e$。这 3 种类型、其 m 和 e 的值，以及它们在实数中的上下限如表 3-2 所示。

表　3-2

类　型	别　名	m 的最小值	m 的最大值	e 的最小值	e 的最大值	近似的最小值	近似的最大值
float	System.Single	0	2^{24}	-149	104	1.5×10^{-45}	3.4×10^{38}
double	System.Double	0	2^{53}	-1075	970	5.0×10^{-324}	1.7×10^{308}
decimal	System.Decimal	0	2^{96}	-26	0	1.0×10^{-28}	7.9×10^{28}

除了数值类型外，另外还有 3 种简单类型，如表 3-3 所示。

表　3-3

类　型	别　名	允 许 的 值
char	System.Char	一个 Unicode 字符，存储 0~65 535 之间的整数
bool	System.Boolean	布尔值：true 或 false
string	System.String	一组字符

注意组成 string 的字符数没有上限，因为它可以使用可变大小的内存。

布尔类型 bool 是 C#中最常用的一种变量类型，类似的类型在其他语言的代码中非常丰富。当编写应用程序的逻辑流程时，一个可以是 true 或 false 的变量有非常重要的分支作用。例如，考虑一下有多少问题可以用 true 或 false(或 yes 和 no)来回答。执行变量值之间的比较或检查输入的有效性就是后面使用布尔变量的两个编程示例。

介绍了这些类型后，下面用一个简短示例来声明和使用它们。在下面的示例中，要使用一些简单的代码来声明两个变量，给它们赋值，再输出这些值。

试一试：使用简单类型的变量

(1) 在目录 C:\BegVCSharp\Chapter03 下创建一个新的控制台应用程序 Ch03Ex01。

(2) 在 Program.cs 中添加如下代码：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
    int myInteger;
    string myString;
    myInteger = 17;
    myString = "\"myInteger\" is";
    Console.WriteLine("{0} {1}.", myString, myInteger);
    Console.ReadKey();
}
```

代码段 Ch03Ex01\Program.cs

(3) 运行代码，结果如图 3-1 所示。

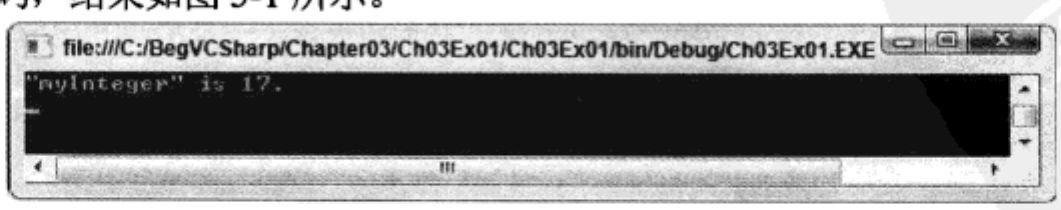


图　3-1

示例的说明

我们添加的代码完成了以下 3 项任务：

- 声明两个变量
- 给这两个变量赋值
- 将两个变量的值输出到控制台上

变量声明使用下述代码：

```
int myInteger;
string myString;
```

第一行声明一个类型为 int 的变量 myInteger，第二行声明一个类型为 string 的变量 myString。

变量的命名是有限制的，不能使用任意的字符序列。“变量的命名”将介绍变量的命名规则。

接下来的两行代码为变量赋值：

```
myInteger = 17;
myString = "\"myInteger\" is";
```

使用=赋值运算符(在本章的“表达式”一节中将详细介绍)给变量分配两个固定的值(在代码中称为字面值)。把整数值 17 赋给 myInteger，把字符串"myInteger"(包括引号)赋给 myString。以这种方式给字符串赋予字面值时，必须用双引号把字符串括起来。因此，如果字符串本身包含双引号，就会出现错误，必须用一些表示这些字符的其他字符(即转义序列)来替代它们。本例使用序列\"来转义双引号：

```
myString = "\"myInteger\" is";
```

如果不使用这些转义序列，而输入如下代码：

```
myString = ""myInteger" is";
```

就会出现编译错误。

注意给字符串赋予字面值时，必须小心换行 —— C#编译器会拒绝分布在多行上的字符串字面值。如果要添加一个换行符，可以在字符串中使用换行符的转义序列，即\n。例如，赋值语句：

```
myString = "This string has a\nline break.";
```

会在控制台视图中显示两行代码，如下所示：

```
This string has a
line break.
```

所有的转义序列都包含一个反斜杠符号，后跟一个字符组合(详见后面的内容)，因为反斜杠符号的这种用途，它本身也有一个转义序列，即两个连续的反斜杠\\。

下面继续解释代码，还有一行没有说明：

```
Console.WriteLine("{0} {1}.", myString, myInteger);
```

它看起来类似于第一个示例中把文本写到控制台上的简单方法，但本例指定了变量。这里不打算详细讨论这行代码。这是本书第 I 部分用于给控制台窗口输出文本的一种技巧，知道这一点就足够了。在括号中，有如下两项：

- 一个字符串
- 一个用逗号分隔的变量列表，这些变量的值将插入到输出字符串中

输出字符串是"{0} {1}."，它们并没有包含有用的文本。可以看出，这并不是我们运行代码时希望看到的结果，其原因是：字符串实际上是插入变量内容的一个模板，字符串中的每对花括号都是一个占位符，包含列表中每个变量的内容。每个占位符(或格式字符串)用包含在花括号中的一个整数来表示。整数从 0 开始，每次递增 1，占位符的总数应等于列表中指定的变量数，该列表用逗号分隔开，跟在字符串后。把文本输出到控制台时，每个占位符就会用每个变量的值来替代。在上面的示例中，{0}用第一个变量的值 myString 替换，{1}用 myInteger 的内容来替换。

在后面的示例中，就使用这种给控制台输出文本的方式显示代码的输出结果。最后一行代码在前面的示例中也出现过，用于在程序结束前等待用户输入内容：

```
Console.ReadKey();
```

这里不详细探讨这行代码，但后面的示例会常常用到它。现在只需要知道，它暂停代码的执行，等待用户按下一个键。

3.3.2　变量的命名

如上一节所述，不能把任意序列的字符作为变量名。这并不像第一次听起来那样需要担心什么，因为这种命名系统仍是非常灵活的。

基本的变量命名规则如下：

- 变量名的第一个字符必须是字母、下划线(_)或@。
- 其后的字符可以是字母、下划线或数字。

另外，有一些关键字对于 C#编译器而言有特定的含义，例如前面出现的 using 和 namespace 关键字。如果错误地使用其中一个关键字，编译器会产生一个错误，我们马上就会知道出错了，所以不必担心。

例如，下面的变量名是正确的：

```
myBigVar
VAR1
_test
```

下列变量名有误：

```
99BottlesOfBeer
namespace
It's-All-Over
```

记住，C#区分大小写，所以必须小心，不要忘了在声明变量时使用正确的大小写。在程序后面引用它们时，即使只有一个字母的大小写形式出错，都不能编译成功。其进一步的结果是得到多个变量，其名称仅有大小写的区别，例如，下面的变量都是不同的：

```
myVariable
MyVariable
MYVARIABLE
```

命名约定

变量名是比较常用的，所以有必要用一定的篇幅讨论几种要用到的变量名称。在开始前，要记住这是有争议的。多年以来，出现了不同的系统，一些开发人员拼命维护他们的个人系统。

最近，最流行的系统是所谓的 Hungarian 表示法。这个系统在所有的变量名上加上一个小写形式的前缀，表示其类型。例如，如果变量的类型是 int，就在其名称前加上 i(或 n)，如 iAge。使用这个系统，很容易看出各个变量是什么类型的。

更现代的语言如 C#则很难实现这个系统。与前面介绍的所有类型一样，可以用一两个字母前缀表示变量的类型。但由于可以创建自己的类型，而且在.NET Framework 中有上百种更复杂的类型，所以这种系统很快就失效了。在多人合作完成的项目中，不同的人很容易遇到易混淆的不同前缀，它们可能导致灾难性的后果。

开发人员现在认识到，最好根据变量的作用来命名它们。如果出现问题，就很容易确定变量的类型。在 VS 和 VCE 中，只需把鼠标指针在变量名上停留足够长的时间，就会弹出一个方框，指出该变量的类型。

目前，在.NET Framework 名称空间中有两种命名约定，称为 PascalCase 和 camelCase。在名称中使用的大小写表示它们的用途。它们都应用到由多个单词组成的名称中，并指定名称中的每个单词除了第一个字母大写外，其余字母都是小写。在 camelCase 中，还有一个规则，即第一个单词以小写字母开头。

下面是 camelCase 变量名：

```
age
firstName
timeOfDeath
```

下面是 PascalCase 变量名：

```
Age
LastName
WinterOfDiscontent
```

Microsoft 建议：对于简单的变量，使用 camelCase 规则，而对于比较高级的命名则使用 PascalCase。最后，注意许多以前的命名系统常常使用下划线字符作为变量名中各个单词之间的分隔符，如 yet_another_variable。但这种用法现在已经淘汰了。

3.3.3　字面值

在前面的示例中，有两个字面值的示例：整数和字符串。其他变量类型也有相关的字面值，如表 3-4 所示。其中有许多涉及到后缀，即在字面值的后面添加一些字符，指定想要的类型。一些字面值有多种类型，在编译时由编译器根据它们的上下文确定其类型。

表 3-4

类　型	类　别	后　缀	示例/允许的值
bool	布尔	无	true 或 false
int, uint, long, ulong	整数	无	100
uint, ulong	整数	u 或 U	100U
long, ulong	整数	l 或 L	100L
ulong	整数	ul、uL、Ul、UL、lu、lU、Lu 或 LU	100UL
float	实数	f 或 F	1.5F
double	实数	无、 d 或 D	1.5
decimal	实数	m 或 M	1.5M
char	字符	无	'a'或转义序列
string	字符串	无	"a...a"，可以包含转义序列

字符串的字面值

本章前面介绍了几个可以在字符串的字面值中使用的转义序列，表 3-5 是这些转义序列的完整列表，以便以后引用。

表 3-5

转 义 序 列	产生的字符	字符的 Unicode 值
\'	单引号	0x0027
\"	双引号	0x0022
\\	反斜杠	0x005C
\0	空	0x0000
\a	警告(产生蜂鸣)	0x0007
\b	退格	0x0008
\f	换页	0x000C
\n	换行	0x000A
\r	回车	0x000D
\t	水平制表符	0x0009
\v	垂直制表符	0x000B

表 3-5 中的“Unicode 值”列是字符在 Unicode 字符集中的 16 进制值。与上面一样，使用 Unicode 转义序列可以指定 Unicode 字符，该转义序列包括标准的\字符，后跟一个 u 和一个 4 位十六进制值(例如，表 3-5 中 x 后面的 4 位数字)。

下面的字符串是等价的：

```
"Karli\'s string."
"Karli\u0027s string."
```

显然，Unicode 转义序列还有更多用途。

也可以逐字地指定字符串，即两个双引号之间的所有字符都包含在字符串中，包括行末字符和需要转义的字符。唯一例外是双引号字符的转义，它们必须指定，以免结束字符串。为此，可以在该字符串之前加一个@字符：

```
@"Verbatim string literal."
```

可以采用一般方式指定这个字符串，但需要使用下面这种方法：

```
@"A short list:
item 1
item 2"
```

逐字指定的字符串在文件名中非常有用，因为文件名中大量使用了反斜杠字符。如果使用一般的字符串，就必须在字符串中使用两个反斜杠，例如：

```
"C:\\Temp\\MyDir\\MyFile.doc"
```

而有了逐字指定的字符串字面值，这段代码就更便于阅读。下面的字符串与上面的等价：

```
@"C:\Temp\MyDir\MyFile.doc"
```

从本书的后面可以看出，字符串是引用类型，而本章中的其他类型都是值类型。所以，字符串也可以被赋予 null 值，即字符串变量不引用字符串。

3.3.4　变量的声明和赋值

快速回顾一下，前面使用变量的类型和名称来声明它们，例如：

```
int age;
```

然后用＝赋值运算符给变量赋值：

```
age = 25;
```

变量在使用前，必须初始化。上面的赋值语句可以用作初始化语句。

这里还可以做两件事，用户可以在 C#代码中看到。第一是同时声明多个类型相同的变量，方法是在类型的后面用逗号分隔变量名，如下所示：

```
int xSize, ySize;
```

其中 xSize 和 ySize 都声明为整数类型。

第二个技巧是在声明变量的同时为它们赋值，即把两行代码合并在一起：

```
int age = 25;
```

可以同时使用这两个技巧：

```
int xSize = 4, ySize = 5;
```

xSize 和 ySize 被赋予不同的值。注意下面的代码：

```
int xSize, ySize = 5;
```

其结果是 ySize 被初始化，而 xSize 仅进行了声明，在使用前仍需要初始化。

3.4　表达式

前面介绍了如何声明和初始化变量，下面该处理它们了。C#包含许多进行这类处理的运算符。把变量和字面值(在使用运算符时，它们都称为操作数)与运算符组合起来，就可以创建表达式，它是计算的基本构件。

运算符范围广泛，有简单的，也有非常复杂的，其中一些可能只在数学应用程序中使用。简单的操作包括所有的基本数学操作，例如+运算符是把两个操作数加在一起，而复杂的操作包括通过变量内容的二进制表示来处理它们。还有专门用于处理布尔值的逻辑运算符，以及赋值运算符，如=运算符。

本章主要介绍数学和赋值运算符，而逻辑运算符将在第 4 章中介绍，主要论述控制程序流程的布尔逻辑。

运算符大致分为如下 3 类。

- 一元运算符，处理一个操作数
- 二元运算符，处理两个操作数
- 三元运算符，处理三个操作数

大多数运算符都是二元运算符，只有几个一元运算符和一个三元运算符，即条件运算符(条件运算符是一个逻辑运算符，详见第 4 章)。下面先介绍数学运算符，它包括一元运算符和二元运算符。

3.4.1　数学运算符

有 5 个简单的数学运算符，其中两个有二元和一元两种形式。表 3-6 列出了这些运算符，并用一个简短示例来说明它们的用法，以及使用简单的数值类型(整数和浮点数)时它们的结果。

表　3-6

运 算 符	类　　别	示例表达式	结　　果
+	二元	var1 = var2 + var3;	var1 的值是 var2 与 var3 的和
-	二元	var1 = var2 - var3;	var1 的值是从 var2 减去 var3 所得的值
*	二元	var1 = var2 * var3;	var1 的值是 var2 与 var3 的乘积
/	二元	var1 = var2 / var3;	var1 是 var2 除以 var3 所得的值
%	二元	var1 = var2 % var3;	var1 是 var2 除以 var3 所得的余数
+	一元	var1 = +var2;	var1 的值等于 var2 的值
-	一元	var1 = - var2;	var1 的值等于 var2 的值乘以 - 1

+(一元)运算符有点古怪，因为它对结果没有影响。它不会把值变成正的：如果 var2 是-1，则+var2 仍是-1。但是，这是一个普遍认可的运算符，所以也把它包含进来。这个运算符最有用的方面是，可以定制它的操作，本书在后面探讨运算符的重载时会介绍它。

上面的示例都使用简单的数值类型，因为使用其他简单类型，结果可能不太清晰。例如把两个布尔值加在一起，会得到什么结果？此时，如果对 bool 变量使用+(或其他数学运算符)，编译器会报错。char 变量的相加也会有点让人摸不着头脑。记住，char 变量实际上存储的是数字，所以把两个 char 变量加在一起也会得到一个数字(其类型为 int)。这是一个隐式转换的示例，稍后将详细介绍这个主题和显式转换，因为它也可以应用到 var1、var2 和 var3 都是混合类型的情况。

二元运算符+在用于字符串类型变量时也是有意义的。此时，表 3-7 的表项应如下所示。

表 3-7

运 算 符	类　别	示例表达式	结　果
+	二元	var1 = var2 + var3;	var1 的值是存储在 var2 和 var3 中的两个字符串的连接值

但其他数学运算符不能用于处理字符串。

这里应介绍的另外两个运算符是递增和递减运算符，它们都是一元运算符，可以以两种方式加以使用：放在操作数的前面或后面。简单表达式的结果如表 3-8 所示。

表 3-8

运 算 符	类　别	示例表达式	结　果
++	一元	var1 = ++var2;	var1 的值是 var2 + 1，var2 递增 1
--	一元	var1 = --var2;	var1 的值是 var2 - 1，var2 递减 1
++	一元	var1 = var2++;	var1 的值是 var2，var2 递增 1
--	一元	var1 = var2--;	var1 的值是 var2，var2 递减 1

这些运算符改变存储在操作数中的值。

- ++总是使操作数加 1
- --总是使操作数减 1

var1 中存储的结果有区别，其原因是运算符的位置决定了它什么时候发挥作用。把运算符放在操作数的前面，则操作数是在进行任何其他计算前受到运算符的影响，而把运算符放在操作数的后面，则操作数是在完成表达式的计算后受到运算符的影响。

这有益于另一个示例，考虑以下代码：

```
int var1, var2 = 5, var3 = 6;
var1 = var2++ * --var3;
```

要把什么值赋予 var1？在计算表达式前，var3 前面的运算符 --会起作用，把它的值从 6 改为 5。

可以忽略 var2 后面的++运算符，因为它是在计算完成后才发挥作用，所以 var1 的结果是 5 与 5 的乘积，即 25。

在许多情况下，这些简单的一元运算符使用起来非常方便，它们实际上是下述表达式的简写形式：

```
var1 = var1 + 1;
```

这类表达式有许多用途，特别适合于在循环中使用，这将在第 4 章讲述。下面的示例说明如何使用数学运算符，并介绍另外两个有用的概念。代码提示用户键入一个字符串和两个数字，然后显示计算结果。

试一试：用数学运算符处理变量

(1) 在目录 C:\BegVCSharp\Chapter03 下创建一个新控制台应用程序 Ch03Ex02。

(2) 在 Program.cs 中添加如下代码：


```
static void Main(string[] args)
{
   double firstNumber, secondNumber;
   string userName;
   Console.WriteLine("Enter your name:");
   userName = Console.ReadLine();
   Console.WriteLine("Welcome {0}!", userName);
   Console.WriteLine("Now give me a number:");
   firstNumber = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("Now give me another number:");
   secondNumber = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
             secondNumber, firstNumber + secondNumber);
   Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
             secondNumber, firstNumber, firstNumber - secondNumber);
   Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
             secondNumber, firstNumber * secondNumber);
   Console.WriteLine("The result of dividing {0} by {1} is {2}.",
              firstNumber, secondNumber, firstNumber / secondNumber);
   Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
              firstNumber, secondNumber, firstNumber % secondNumber);
   Console.ReadKey();
}
```

代码段 Ch03Ex02\Program.cs

(3) 执行代码，结果如图 3-2 所示。

图　3-2

(4) 输入名称，按下回车键，如图 3-3 所示。

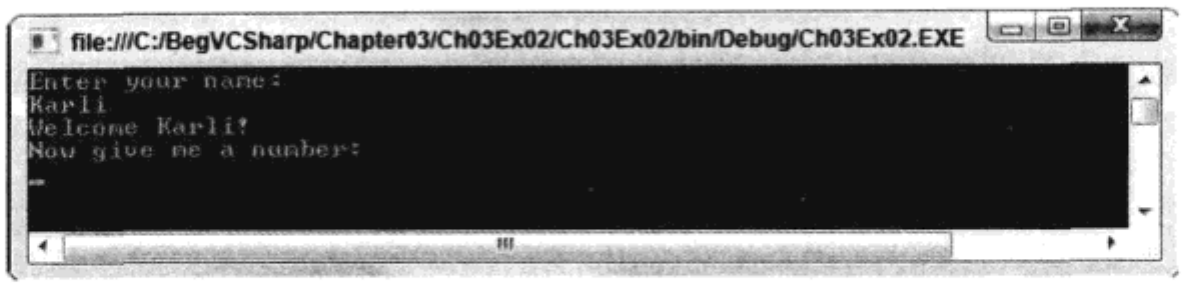


图　3-3

(5) 输入一个数字，按下回车键，再输入另一个数字，按下回车键，如图 3-4 所示。

图　3-4

示例的说明

除了演示数学运算符外，这段代码还引入了两个重要概念，在以后的示例中将多次用到这些概念。

- 用户输入
- 类型转换

用户输入使用与前面 Console.WriteLine()命令类似的语法。但这里使用 Console.ReadLine()。这个命令提示用户输入信息，并把它们存储在 string 变量中。

```
string userName;
 Console.WriteLine("Enter your name:");
 userName = Console.ReadLine();
 Console.WriteLine("Welcome {0}!", userName);
```

这段代码直接将已赋值变量 userName 的内容写到屏幕上。

这个示例还读取了两个数字，下面略微展开讨论一下。因为 Console.ReadLine()命令生成一个字符串，而我们希望得到一个数字，所以这就引入了类型转换的问题。第 5 章将详细讨论类型转换，下面先看看本例使用的代码。

首先，声明要存储数字的变量：

```
double firstNumber, secondNumber;
```

接着，给出提示，对 Console.ReadLine()得到的字符串使用命令 Convert.ToDouble()，把字符串转换为 double 类型，把这个数值赋给前面声明的变量 firstNumber：

```
 Console.WriteLine("Now give me a number:");
```

```
firstNumber = Convert.ToDouble(Console.ReadLine());
```

这个语法是相当简单的，其他的许多转换也用类似的方式进行。

其余的代码按同样的方式获取第二个数：

```
Console.WriteLine("Now give me another number:");
secondNumber = Convert.ToDouble(Console.ReadLine());
```

然后输出两个数字的加、减、乘、除的结果，并使用余数运算符(%)显示除操作的余数。

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
          secondNumber, firstNumber + secondNumber);
Console.WriteLine("The result of subtracting {0} from {1} is {2}.",
          secondNumber, firstNumber, firstNumber - secondNumber);
Console.WriteLine("The product of {0} and {1} is {2}.", firstNumber,
          secondNumber, firstNumber * secondNumber);
Console.WriteLine("The result of dividing {0} by {1} is {2}.",
          firstNumber, secondNumber, firstNumber / secondNumber);
Console.WriteLine("The remainder after dividing {0} by {1} is {2}.",
          firstNumber, secondNumber, firstNumber % secondNumber);
```

注意，我们提供了表达式 firstNumber + secondNumber 等，作为 Console.WriteLine()语句的一个参数，而没有使用中间变量：

```
Console.WriteLine("The sum of {0} and {1} is {2}.", firstNumber,
          secondNumber, firstNumber + secondNumber);
```

这种语法可以提高代码的可读性，减少需要编写的代码量。

3.4.2　赋值运算符

我们迄今一直在使用简单的=赋值运算符，其实还有其他赋值运算符，而且它们都很有用。除了＝运算符外，其他赋值运算符都以类似的方式工作。与＝一样，它们都是根据运算符和右边的操作数，把一个值赋给左边的变量。

表 3-9 列出了这些运算符及其说明。

表　3-9

运 算 符	类　别	示例表达式	结　果
=	二元	var1 = var2;	var1 被赋予 var2 的值
+=	二元	var1 += var2;	var1 被赋予 var1 与 var2 的和
-=	二元	var1 -= var2;	var1 被赋予 var1 与 var2 的差
*=	二元	var1 *= var2;	var1 被赋予 var1 与 var2 的乘积
/=	二元	var1 /= var2;	var1 被赋予 var1 与 var2 相除所得的结果
%=	二元	var1 %= var2;	var1 被赋予 var1 与 var2 相除所得的余数

可以看出，这些运算符把 var1 也包括在计算过程中，下面的代码：

```
var1 += var2;
```

与下面的代码结果相同。

```
var1 = var1 + var2;
```

+=运算符也可以用于字符串，与+运算符一样。

使用这些运算符，特别是在使用长变量名时，可以使代码更便于阅读。

3.4.3　运算符的优先级

在计算表达式时，会按顺序处理每个运算符。但这并不意味着必须从左至右地运用这些运算符。例如，有下面的代码：

```
var1 = var2 + var3;
```

其中+运算符就是在=运算符之前进行计算的。在其他一些情况下，运算符的优先级并没有这么明显，例如：

```
var1 = var2 + var3 * var4;
```

其中*运算符先计算，其后是+运算符，最后是=运算符，这是标准的数学运算顺序，其结果与我们在纸上进行算术运算的结果相同。

像这样的计算，可以使用括号控制运算符的优先级，例如：

```
var1 = (var2 + var3) * var4;
```

先计算括号中的内容，即+运算符在*运算符之前计算。

对于前面介绍的运算符，其优先级如表 3-10 所示，优先级相同的运算符(如*和/)按照从左至右的顺序计算。

表 3-10

优 先 级	运 算 符
优先级由高到低	++, --(用作前缀); +, -(一元)
	*, /, %
	+, -
	=, *=, /=, %=, +=, -=
	++, --(用作后缀)

如上所述，括号可用于重写优先级顺序。另外，++和--用作后缀运算符时，在概念上其优先级最低，如上表所示。它们不对赋值表达式的结果产生影响，所以可以认为它们的优先级比所有其他运算符都高。但是，它们会在计算表达式后改变操作数的值，所以很容易认可它们在上表中的优先级。

3.4.4　名称空间

在继续学习前，应花一定的时间了解一个比较重要的主题——名称空间。它们是.NET 中提供应用程序代码容器的方式，这样就可以唯一地标识代码及其内容。名称空间也用作.NET Framework 中给项分类的一种方式。大多数项都是类型定义，例如，本章描述的简单类型(System.Int32 等)。

默认情况下，C#代码包含在全局名称空间中。这意味着对于包含在这段代码中的项，只要按照名称进行引用，就可以由全局名称空间中的其他代码访问它们。可以使用 namespace 关键字为花括号中的代码块显式定义名称空间。如果在该名称空间代码的外部使用名称空间中的名称，就必须写出该名称空间中的限定名称。

限定名称包括它所有的分层信息。这基本上意味着，如果一个名称空间中的代码需要使用在另一个名称空间中定义的名称，就必须包括对该名称空间的引用。限定名称在不同的命名空间级别之间使用句点字符(.)。如下所示：

```
namespace LevelOne
{
   // code in LevelOne namespace

   // name "NameOne" defined
}

// code in global namespace
```

这段代码定义了一个名称空间 LevelOne，以及该名称空间中的一个名称 NameOne(注意这里没有列出其他代码，是为了使我们的讨论更具普遍性，并在定义名称空间的地方添加了一个注释)。在名称空间 LevelOne 中编写的代码可以使用 NameOne 来引用该名称，不需要任何分类信息。但全局名称空间中的代码必须使用分类名称 LevelOne.NameOne 来引用这个名称。

根据约定，名称空间通常采用 PascalCase 命名方式。

在名称空间中，使用关键字 namespace 还可以定义嵌套的名称空间。嵌套的名称空间通过其层次结构来引用，并使用句点区分层次结构的层次。这最好通过一个示例来加以说明。考虑下面的名称空间：

```
namespace LevelOne
{
   // code in LevelOne namespace

   namespace LevelTwo
   {
      // code in LevelOne.LevelTwo namespace

      // name "NameTwo" defined
   }
}

// code in global namespace
```

在全局名称空间中，NameTwo 必须引用为 LevelOne.LevelTwo.NameTwo；在 LevelOne 名称空间中，可以引用为 LevelTwo.NameTwo；在 LevelOne.LevelTwo 名称空间中，可以引用为 NameTwo。

非常重要的一点是，名称是由名称空间唯一定义的。可以在 LevelOne 和 LevelTwo 名称空间中定义名称 NameThree：

```
namespace LevelOne
{
   // name "NameThree" defined

   namespace LevelTwo
   {
      // name "NameThree" defined
   }
}
```

这定义了两个不同的名称 LevelOne.NameThree 和 LevelOne.LevelTwo.NameThree，可以独立使用它们，互不干扰。

创建了名称空间后，即可使用 using 语句简化对它们所含名称的访问。实际上，using 语句的意思是“我们需要这个名称空间中的名称，所以不要每次总是要求对它们分类”。例如，在下面的代码中，LevelOne 名称空间中的代码可以访问 LevelOne.LevelTwo 名称空间中的名称，而无需分类：

```
namespace LevelOne
{
   using LevelTwo;

   namespace LevelTwo
   {
      // name "NameTwo" defined
   }
}
```

LevelOne 命名空间中的代码现在可以直接使用 NameTwo 引用 LevelTwo.NameTwo。

有时，与上面的 NameThree 示例一样，不同名称空间中的相同名称会产生冲突，使系统崩溃(此时，代码无法编译，编译器会告诉我们名称有冲突)。此时，可以使用 using 语句为名称空间提供一个别名。

```
namespace LevelOne
{
   using LT = LevelTwo;

   // name "NameThree" defined

   namespace LevelTwo
   {
      // name "NameThree" defined
   }
}
```

LevelOne 名称空间中的代码可以把 LevelOne.NameThree 引用为 NameThree，把 LevelOne.LevelTwo.NameThree 引用为 LT.NameThree。

using 语句可以应用到包含它们的名称空间，以及该名称空间中包含的嵌套命名空间。在上面的代码中，全局名称空间不能使用 LT.NameThree。但如果 using 语句声明如下：

```
using LT = LevelOne.LevelTwo;

namespace LevelOne
{
   // name "NameThree" defined

   namespace LevelTwo
   {
      // name "NameThree" defined
   }
}
```

全局名称空间中的代码和 LevelOne 名称空间中的代码就可以使用 LT.NameThree。

需要注意特别重要的一点：using 语句本身不能访问另一个名称空间中的名称。除非名称空间中的代码以某种方式链接到项目上，或者代码是在该项目的源文件中定义的，或在链接到该项目的其他代码中定义的，否则就不能访问其中包含的名称。另外，如果包含名称空间的代码链接到项目上，无论是否使用 using，都可以访问其中包含的名称。using 语句便于我们访问这些名称，减少代码量，以及提高可读性。

回过头来看看本章开头的 ConsoleApplication1 中的代码，下面的代码被应用到名称空间上：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   ...
}
```

以 using 关键字开头的 4 行代码声明在这段 C#代码中使用 System、System.Collections. Generic、System.Linq 和 System.Text 名称空间，它们可以在该文件的所有名称空间中访问，无需分类。System 名称空间是.NET Framework 应用程序的根命名空间，包含控制台应用程序需要的所有基本功能。其他 3 个名称空间常用于控制台应用程序，所以该程序包含了这 4 行代码。

最后，为应用程序代码本身声明一个名称空间 ConsoleApplication1。

3.5　小结

本章介绍了创建有效 C#应用程序的大量基础知识，讲述了 C#的基本语法，分析了在创建控制台应用程序项目时 VS 和 VCE 生成的基本控制台应用程序代码。

本章重点讲述变量的用法。我们描述了变量的含义，阐述了如何创建变量，如何给它们赋值，如何处理它们以及它们包含的值。同时，介绍了一些基本的用户交互，描述了如何把文本输出到控制台应用程序上，如何读取用户的输入。这涉及到一些非常基本的类型转换。类型转换是一个复杂

的主题，将在第 5 章详细论述。

本章还介绍了如何将运算符和操作数组合为表达式，并说明了这些运算符的执行方式，以及它们的执行顺序。

最后介绍了名称空间。随着本书内容的深入，命名空间会显得越来越重要。这里仅以比较抽象的方式介绍了这个主题，完整的论述请见本书后面的内容。

到目前为止，所有的编程工作都是逐行完成的。第 4 章将学习如何使用循环技术和条件分支控制程序执行的流程，以便提高代码的效率。

3.6　练习

(1) 在下面的代码中，如何从名称空间 fabulous 的代码中引用名称 great?

```
namespace fabulous
{
   // code in fabulous namespace
}

namespace super
{
   namespace smashing
   {
      // great name defined
   }
}
```

(2) 下面哪些变量名不合法?

- myVariableIsGood
- 99Flake
- _floor
- time2GetJiggyWidIt
- wrox.com

(3) 字符串 supercalifragilisticexpialidocious 是因为太长了而不能放在 string 变量中吗？为什么？

(4) 考虑运算符的优先级，列出下述表达式的计算步骤。

```
resultVar += var1 * var2 + var3 % var4 / var5;
```

(5) 编写一个控制台应用程序，要求用户输入 4 个 int 值，并显示它们的乘积。提示：可以使用 Convert.ToDouble()命令，把用户在控制台上输入的数转换为 double；从 string 转换为 int 的命令是 Convert.ToInt32()。

附录 A 给出了练习答案。

3.7　本章要点

主　题	重 要 概 念
C#基本语法	C#是一种区分大小写的语言，每行代码都以分号结束。如果代码行太长或者表示嵌套的块，可以缩进代码行，以方便阅读。使用//或/*...*/语法可以包含不编译的注释。代码块可以隐藏到区域中，也是为了方便阅读
变量	变量是有名称和类型的数据块。.NET Framework 定义了大量的简单类型，例如数字和字符串(文本)类型，以供使用。变量只有经过声明和初始化后，才能使用。可以把字面值赋予变量，以初始化它们，变量还可以在单个步骤中声明和初始化
表达式	表达式利用运算符和操作数来建立，其中运算符对操作数执行操作。运算符有 3 种：一元、二元和三元运算符，它们分别操作 1、2 和 3 个操作数。数学运算符对数值执行操作，赋值运算符把表达式的结果放在变量中。运算符有固定的优先级，优先级确定了运算符在表达式中的处理顺序
名称空间	.NET 应用程序中定义的所有名称，包括变量名，都包含在名称空间中。名称空间采用层次结构，我们通常需要根据包含名称的名称空间来限定名称，以便访问它们

第 4 章

流 程 控 制

本章内容：

- 布尔逻辑的含义及其用法
- 如何控制代码的分支
- 如何编写循环代码

我们迄今看到的 C#代码有一个共同点：程序的执行都是一行接一行、自上而下地进行，不遗漏任何代码。如果所有应用程序都这样执行，则我们能做的工作就很有限了。本章介绍控制程序流的两种方法。程序流程就是 C#代码的执行顺序。这两种方法是分支和循环。分支是有条件地执行代码。条件取决于计算的结果，例如，“只有 myVal 小于 10，才执行这行代码”。循环重复执行相同的语句(重复执行一定的次数，或者在满足测试条件后停止执行)。

这两种方法都要用到布尔逻辑。第 3 章介绍了 bool 类型，但并未讨论它。本章将在很多地方使用它，所以先讨论布尔逻辑，以便在流程控制环境下使用它。

4.1 布尔逻辑

第 3 章介绍的 bool 类型可以有两个值：true 或 false。这种类型常常用于记录某些操作的结果，以便操作这些结果。bool 类型可用于存储比较结果。

19 世纪中叶的英国数学家乔治·布尔为布尔逻辑奠定了基础。

考虑下述情形(如本章引言所述)：要根据变量 myVal 是否小于 10，来确定是否执行代码。为此，需要确定语句“myVal 小于 10”的真假，即需要了解比较的布尔结果。

布尔比较需要使用布尔比较运算符(也称为关系运算符)，如表 4-1 所示。这里 var1 都是 bool 类型的变量，var2 和 var3 则可以是不同类型。

表 4-1

运 算 符	类　别	示例表达式	结　果
==	二元	var1 = var2 = = var3;	如果 var2 等于 var3，var1 的值就是 true，否则为 false
!=	二元	var1 = var2 != var3;	如果 var2 不等于 var3，var1 的值就是 true，否则为 false
<	二元	var1 = var2 < var3;	如果 var2 小于 var3，var1 的值就是 true，否则为 false
>	二元	var1 = var2 > var3;	如果 var2 大于 var3，var1 的值就是 true，否则为 false
<=	二元	var1 = var2 <= var3;	如果 var2 小于等于 var3，var1 的值就是 true，否则为 false
>=	二元	var1 = var2 >= var3;	如果 var2 大于等于 var3，var1 的值就是 true，否则为 false

在代码中，可以对数值使用以下这些运算符：

```
bool isLessThan10;
isLessThan10 = myVal < 10;
```

如果 myVal 存储的值小于 10，这段代码就给 isLessThan10 赋予 true 值，否则赋予 false 值。
也可以对其他类型使用这些比较运算符，例如字符串：

```
bool isKarli;
isKarli = myString == "Karli";
```

如果 myString 存储的字符串是 Karli，isKarli 的值就为 true。
也可以对布尔值使用这些运算符：

```
bool isTrue;
isTrue = myBool == true;
```

但只能使用==和!=运算符。

一个常见的代码错误是，无意间假定由于 val1 < val2 是 false，所以 val1 > val2 为 true。如果 val1 == val2，则这两个语句都是 false。

在处理布尔值时，还有其他一些布尔运算符，如表 4-2 所示。

表 4-2

运算符	类 别	示例表达式	结 果
!	一元	var1 = ! var2;	如果 var2 是 false，var1 的值就是 true，否则为 false(逻辑非)
&	二元	var1 = var2 & var3;	如果 var2 和 var3 都是 true，var1 的值就是 true，否则为 false(逻辑与)
|	二元	var1 = var2 | var3;	如果 var2 或 var3 是 true(或两者都是)，var1 的值就是 true，否则为 false(逻辑或)
^	二元	var1 = var2 ^ var3;	如果 var2 或 var3 中有且仅有一个是 true，var1 的值就是 true，否则为 false (逻辑异或)

上面的代码也可以表述为：

```
bool isTrue;
isTrue = myBool & true;
```

&和 | 运算符也有两个类似的运算符，称为条件布尔运算符(见表 4-3)。

表 4-3

运算符	类 别	示例表达式	结 果
&&	二元	var1 = var2 && var3;	如果 var2 和 var3 都是 true，var1 的值就是 true，否则为 false (逻辑与)
||	二元	var1 = var2 || var3;	如果 var2 或 var3 是 true(或两者都是)，var1 的值就是 true，否则为 false (逻辑或)

这些运算符的结果与&和 | 完全相同，但得到结果的方式有一个重要区别：其性能比较好。两者都是检查第一个操作数的值(表 4-3 中的 var2)，再根据该操作数的值进行操作，可能根本就不处理第二个操作数(表 4-3 中的 var3)。

如果&&运算符的第一个操作数是 false，就不需要考虑第二个操作数的值了，因为无论第二个操作数的值是什么，其结果都是 false。同样，如果第一个操作数是 true，|| 运算符就返回 true，无需考虑第二个操作数的值。但上面的&和 | 运算符却不是这样。它们总是要计算两个操作数。

因为操作数的计算是有条件的，如果使用&&和||运算符来代替&和 |，性能会有一定提高。在大量使用这些运算符的应用程序中这表现得尤为明显。作为一个规则，尽可能使用&&和 || 运算符。这些运算符有时用于比较复杂的情形，例如，只有第一个操作数包含某个值时，才计算第二个操作数：

```
var1 = (var2 != 0) && (var3 / var2 >2);
```

如果 var2 是 0，则 var3 除以 var2 就会导致“除 0 错误”，或者把 var1 定义为无穷大(对于某些类型如 float 来说，可能出现后一种情形，也是可以检测到的)。

读者此时可能会问，为什么会有&和 | 运算符。原因是这两个运算符可以用于对数值执行操作。实际上，它们处理的是存储在变量中的一系列位，而不是变量的值。请参见稍后的“按位运算符”。

4.1.1　布尔赋值运算符

使用布尔赋值运算符可以把布尔比较与赋值组合起来，其方式与第 3 章中的数学赋值运算符(+=, *=等)相同。布尔值如表 4-4 所示。

表　4-4

运 算 符	类　别	示例表达式	结　果
&=	二元	var1 &= var2;	var1 的值是 var1 & var2 的结果
|=	二元	var1 |= var2;	var1 的值是 var1 | var2 的结果
^=	二元	var1 ^= var2;	var1 的值是 var1 ^ var2 的结果

这些运算符处理布尔值和数值的方式与&、| 和 ^ 相同。

&=和 |= 赋值运算符并不使用&&和 || 条件布尔运算符，即无论赋值运算符左边的值是什么，都处理所有的操作数。

在下面的示例中，用户键入一个整数，然后代码使用该整数执行各种布尔运算。

试一试：使用布尔运算符

(1) 在目录 C:\BegVCSharp\Chapter04 下创建一个新控制台应用程序 Ch04Ex01。

(2) 把以下代码添加到 Program.cs 中：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
    Console.WriteLine("Enter an integer:");
    int myInt = Convert.ToInt32(Console.ReadLine());
    bool isLessThan10 = myInt < 10;
    bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
    Console.WriteLine("Integer less than 10? {0}", isLessThan10);
    Console.WriteLine("Integer between 0 and 5? {0}", isBetween0And5);
    Console.WriteLine("Exactly one of the above is true? {0}",
        isLessThan10 ^ isBetween0And5);
    Console.ReadKey();
}
```

代码段 Ch04Ex01\Program.cs

(3) 运行应用程序，出现提示时，输入一个整数，结果如图 4-1 所示。

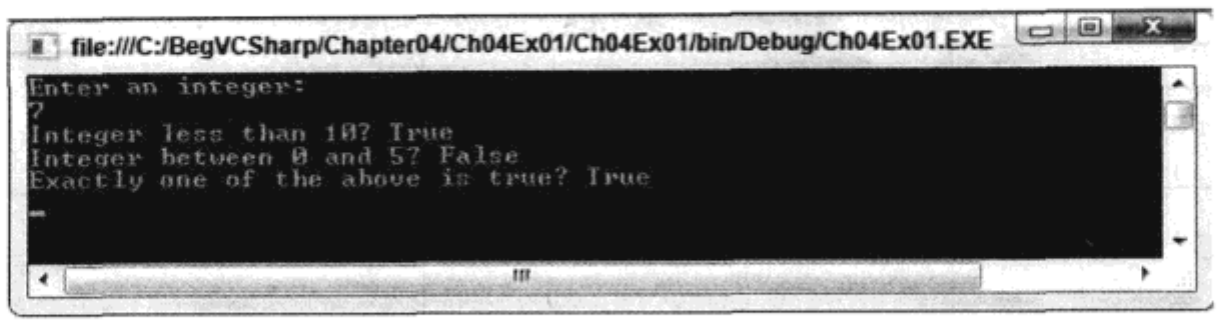


图 4-1

示例的说明

前两行代码使用前面介绍的技术，提示并接受一个整数值：

```
Console.WriteLine("Enter an integer:");
int myInt = Convert.ToInt32(Console.ReadLine());
```

使用 Convert.ToInt32()从字符串输入中得到一个整数。Convert.ToInt32()是另一个类型转换命令，与前面使用的 Convert.ToDouble()命令属于同一系列。

接着声明两个布尔变量 isLessThan10 和 isBetween0And5，并赋值，其中的逻辑匹配其名称中的描述：

```
bool isLessThan10 = myInt < 10;
bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
```

接着在下面的 3 行代码中使用这些变量，前两行代码输出它们的值，第 3 行对它们执行一个操作，并输出结果。在执行这段代码时，假定用户输入了 7，如图 4-1 所示。

第一个输出是操作 myInt < 10 的结果。如果 myInt 是 6，则它小于 10，因此结果为 true。如果 MyInt 的值是 10 或更大，就会得到 false。

第二个输出涉及较多计算：(0 <= myInt) && (myInt <= 5)，其中包含两个比较操作，用于确定 myInt 是否大于或等于 0，且小于或等于 5。接着对结果进行布尔 AND 操作。输入数字 6，则(0 <= myInt)返回 true，而(myInt <= 5)返回 false，最终结果就是(true) && (false)，即 false，如图 4-1 所示。

最后，对两个布尔变量 isLessThan10 和 isBetween0And5myInt 执行逻辑异或操作。如果一个变量的值是 true，另一个是 false，则代码返回 true，所以只有 myInt 是 6、7、8 或 9，才返回 true，本例输入的是 6，所以结果是 true。

4.1.2 按位运算符

前面介绍的&和 | 运算符还有一个作用：对数值执行操作。以这种方式使用时，它们处理的是变量中存储的一系列位，而不是变量值，因此它们称为按位运算符。

本节介绍它们和 C#语言定义的其他按位运算符。在大多数开发工作中，除了数学应用之外，这个功能都不太常用。因此本节没有示例。

下面先讨论&和 |。第一个操作数中的每个位都与第二个操作数中相同位置上的位进行比较，在得到的结果中，各个位置上的位如表 4-5 所示。

| 运算符与此类似，但得到的结果位是不同的，如表 4-6 所示。

表　4-5

操作数 1 的位	操作数 2 的位	&的结果位
1	1	1
1	0	0
0	1	0
0	0	0

表　4-6

操作数 1 的位	操作数 2 的位	| 的结果位
1	1	1
1	0	1
0	1	1
0	0	0

例如，考虑下面代码中的操作：

```
int result, op1, op2;
op1 = 4;
op2 = 5;
result = op1 & op2;
```

这里必须考虑 op1 和 op2 的二进制表示方式，它们分别是 100 和 101。比较这两个表达方式中相同位置上的二进制数字，得出结果，如下所示：

- 如果 op1 和 op2 最左边的位都是 1，result 最左边的位就是 1，否则为 0。
- 如果 op1 和 op2 次左边的位都是 1，result 次左边的位就是 1，否则为 0。
- 继续比较其他的位。

在这个示例中，op1 和 op2 最左边的位都是 1，所以 result 最左边的位就是 1。下一个位都是 0，第 3 个位置上的位分别是 1 和 0，则 result 第 2~3 个位都是 0。最后，结果的二进制值是 100，即结果是 4。以下是这个过程：

```
    1   0   0        4
&   1   0   1    &   5
-------------    -----
    1   0   0        4
```

如果使用 | 运算符，将进行相同的过程，但如果操作数中相同位置上的位有一个是 1，其结果位就是 1，如下所示：

```
    1  0  0        4
| 1  0  1    | 5
-------------   -----
    1  0  1        5
```

^运算符的用法与此相同。如果操作数中相同位置上的位有且仅有一个是1，其结果位就是1，如表4-7所示。

表 4-7

操作数1的位	操作数2的位	^的结果位
1	1	0
1	0	1
0	1	1
0	0	0

C#中还可以使用一元位运算符~，它将操作数中的位取反，其结果应是操作数中位为1的，在结果中就是0，反之亦然，如表4-8所示。

表 4-8

操作数的位	~的结果位
1	0
0	1

整数存储在.NET中的方式称为2的补位，即使用一元运算符～会使结果看起来有点古怪。假定int类型是一个32位的数字，则运算符～对所有32位进行操作，将有助于看出这种方式。例如，数字5的完整二进制表示为：

```
00000000000000000000000000000101
```

数字－5的完整二进制表示为：

```
11111111111111111111111111111011
```

实际上，按照2的补位系统，(－x)定义为(~x+1)。这个系统在把数字加在一起时非常有用。例如，把10和－5加起来(即从10中减去5)的二进制表示为：

```
   00000000000000000000000000001010
+  11111111111111111111111111111011
= 100000000000000000000000000000101
```

忽略最左端的1，就得到5的二进制表示。像~1==2这样的式子比较古怪，其原因是底层的结构强制生成了这个结果。

在某些情况下，本节介绍的这些位运算符是非常有用的，因为它们可以用变量中的各个位存储信息。例如，颜色可以使用 3 个位来指定红、绿、蓝。可以分别设置这些位，改变这 3 个位，进行以下一种配置，如表 4-9 所示。

表 4-9

位	十 进 制 数	含 义
000	0	黑色
100	4	红色
010	2	绿色
001	1	蓝色
101	5	洋红色
110	6	黄色
011	3	青色
111	7	白色

假定把这些值存储在一个类型为 int 的变量中。首先从黑色开始，即值为 0 的 int 变量，可以执行如下操作：

```
int myColor = 0;
bool containsRed;
myColor = myColor | 2;              // Add green bit, myColor now stores 010
myColor = myColor | 4;              // Add red bit, myColor now stores 110
containsRed = (myColor & 4) == 4;   // Check value of red bit
```

最后一行代码将值 true 赋予 containsRed，因为 myColor 的“红色位”是 1。这种技术在高效使用信息时非常有效，特别适合于同时检查多个位的值(对于 int 值，是 32 位)。但是，在单个变量中存储额外信息有更好的方式，即利用第 5 章讨论的高级变量类型。

除了这 4 个按位运算符外，本节还要介绍另外两个运算符，如表 4-10 所示。

表 4-10

运 算 符	类 别	示例表达式	结 果
>>	二元	var1 = var2 >> var3;	把 var2 的二进制值向右移动 var3 位，就得到 var1 的值
<<	二元	var1 = var2 << var3;	把 var2 的二进制值向左移动 var3 位，就得到 var1 的值

这些运算符通常称为位移运算符，最好用一个简单的示例加以说明：

```
int var1, var2 = 10, var3 = 2;
var1 = var2 << var3;
```

结果，var1 的值是 40。具体过程如下：10 的二进制值是 1010，把该数值向左移动两位，得到 101000，即十进制中的 40。实际上，是执行了乘法操作。每向左移动一位，该数都要乘以 2，所以向左移动两位，就给原来的操作数乘以 4。而每向右移动一位，则是给操作数除以 2，并丢弃非整余数：

```
int var1, var2 = 10;
var1 = var2 >> 1;
```

在这个示例中，var1 的值是 5，而下面的代码得到的值是 2：

```
int var1, var2 = 10;
var1 = var2 >> 2;
```

在大多数代码中，都不使用这些运算符，但应知道有这样的运算符存在。它们主要用于高度优化的代码，在这些代码中，不能使用其他数学操作。因此它们通常用于设备驱动程序或系统代码。

位移运算符也有赋值运算符，如表 4-11 所示。

表 4-11

运 算 符	类 别	示例表达式	结 果
>>=	一元	var1 >>= var2;	把 var1 的二进制值向右移动 var2 位，就得到 var1 的值
<<=	一元	var1 <<= var2;	把 var1 的二进制值向左移动 var2 位，就得到 var1 的值

4.1.3 运算符优先级的更新

现在要考虑更多的运算符，所以应更新第 3 章中的运算符优先级表，把它们包括在内，如表 4-12 所示。

表 4-12

优 先 级	运 算 符
优先级由高到低	++, -- (用作前缀); (), +, - (一元), !, ~
	*, /, %
	+, -
	<<, >>
	<, >, <=, >=
	==, !=
	&
	^
	|
	&&
	||
	=, *=, /=, %=, +=, - =, <<=, >>=, &=, ^=, |=
	++, --(用作后缀)

该表增加了好几个级别，但它明确定义了下述表达式该如何计算：

```
var1 = var2 <= 4 && var2 >= 2;
```

其中&&运算符在<= 和 >=运算符之后执行(在这行代码中，var2 是一个 int 值)。

这里要注意的是，添加括号可以使这样的表达式看起来更清晰。编译器知道用什么顺序执行运算符，但人们常常会忘记这个顺序(有时可能想改变这个顺序)。上述表达式也可以写为：

```
var1 = (var2 <= 4) && (var2 >= 2);
```

要解决这个问题，可以明确指定计算的顺序。

4.2　goto 语句

C#允许给代码行加上标签，这样就可以使用 goto 语句直接跳转到这些代码行上。该语句优缺点并存。主要的优点是：这是控制什么时候执行哪些代码的一种简单方式。主要的缺点是：过多地使用这个技巧将使代码晦涩难懂。

goto 语句的用法如下：

```
goto <labelName>;
```

标签用下述方式定义：

```
<labelName>:
```

例如，下面的代码：

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

其执行过程如下：

- myInteger 声明为 int 类型，并赋予值 5。
- goto 语句中断正常的执行过程，把控制权转到标有 myLabel:的代码行上。
- myInteger 的值写入控制台。

下面的第 3 行代码从未执行。

```
int myInteger = 5;
goto myLabel;
myInteger += 10;
myLabel:
Console.WriteLine("myInteger = {0}", myInteger);
```

实际上，如果在应用程序中加入这段代码，会发现编译代码时，Error List 窗口会显示一个警告，即 Unreachable code detected 和一个行号。在无法执行的代码行中，myInteger 下面还有绿色的波浪线。

goto 语句有它们的作用，但也可能使代码陷入混乱。尽量不要使用它(使用本章后面介绍的技巧，就可以避免使用它)。例如，因使用 goto 语句而非常难懂的代码如下所示：

```
start:
int myInteger = 5;
```

```
goto addVal;
writeResult:
Console.WriteLine("myInteger = {0}", myInteger);
goto start;
addVal:
myInteger += 10;
goto writeResult;
```

这是有效的代码，但非常难懂，读者可以自己试试，看看会发生什么情况。在此之前，应尝试理解这些代码会完成什么任务。后面再分析这个语句，因为本章的其他一些结构将使用该语句。

4.3 分支

分支是控制下一步要执行哪行代码的过程。要跳转到的代码行由某个条件语句来控制。这个条件语句使用布尔逻辑，对测试值和一个或多个可能的值进行比较。

本节介绍 C#中的 3 种分支技术：

- 三元运算符
- if 语句
- switch 语句

4.3.1 三元运算符

最简单的比较方式是使用第 3 章介绍的三元(或条件)运算符。一元运算符有一个操作数，二元运算符有两个操作数，所以三元运算符有 3 个操作数。其语法如下：

```
<test> ? <resultIfTrue> : <resultIfFalse>
```

其中，计算<test> 可得到一个布尔值，运算符的结果根据这个值来确定是<resultIfTrue>，还是<resultIfFalse>。

使用三元运算符可以测试 int 变量 myInteger 的值：

```
string resultString = (myInteger < 10) ? "Less than 10"
                                       : "Greater than or equal to 10";
```

三元运算符的结果是两个字符串中的一个，这两个字符串都可能赋给 resultString。把哪个字符串赋给 resultString，取决于 myInteger 的值与 10 的比较。如果 myInteger 的值小于 10，就把第一个字符串赋给 resultString；如果 myInteger 的值大于或等于 10，就把第二个字符串赋给 resultString。例如，如果 myInteger 的值是 4，则 resultString 的值就是字符串"Less than 10"。

这个运算符比较适用于这样的简单赋值语句，但不适用于根据比较结果执行大量代码的情形。此时应使用 if 语句。

4.3.2 if 语句

if 语句的功能比较多，是有效的决策方式。与?:语句不同的是，if 语句没有结果(所以不在赋值语句中使用它)，使用该语句是为了有条件地执行其他语句。

if 语句最简单的语法如下：

```
if (<test>)
   <code executed if <test> is true>;
```

先执行<test>(其计算结果必须是一个布尔值，这样代码才能编译)，如果<test>的计算结果是 true，就执行该语句之后的代码。在这段代码执行完毕后，或者因为<test>的计算结果是 false，而没有执行这段代码，将继续执行后面的代码行。

也可以将 else 语句和 if 语句合并使用，指定其他代码。如果<test>的计算结果是 false，就执行 else 语句：

```
if (<test>)
  <code executed if <test> is true>;
else
  <code executed if <test> is false>;
```

可以使用成对的花括号将这两段代码放在多个代码行上：

```
if (<test>)
{
  <code executed if <test> is true>;
}
else
{
  <code executed if <test> is false>;
}
```

例如，重新编写上一节使用三元运算符的代码：

```
string resultString = (myInteger < 10) ? "Less than 10"
                                       : "Greater than or equal to 10";
```

因为 if 语句的结果不能赋给一个变量，所以要单独将值赋给变量：

```
string resultString;
if (myInteger < 10)
  resultString = "Less than 10";
else
  resultString = "Greater than or equal to 10";
```

这样的代码尽管比较冗长，但与三元运算符相比，更便于阅读和理解，也更加灵活。

下面的示例演示了 if 语句的用法。

试一试：使用 if 语句

(1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新控制台应用程序 Ch04Ex02。

(2) 把下列代码添加到 Program.cs 中：


可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   string comparison;
   Console.WriteLine("Enter a number:");
   double var1 = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("Enter another number:");
   double var2 = Convert.ToDouble(Console.ReadLine());
```

```
   if (var1 < var2)
      comparison = "less than";
   else
   {
      if (var1 == var2)
         comparison = "equal to";
      else
          comparison = "greater than";
   }
   Console.WriteLine("The first number is {0} the second number.",
                     comparison);
   Console.ReadKey();
}
```

代码段 Ch04Ex02\Program.cs

(3) 执行代码，根据提示输入两个数字，如图 4-2 所示。

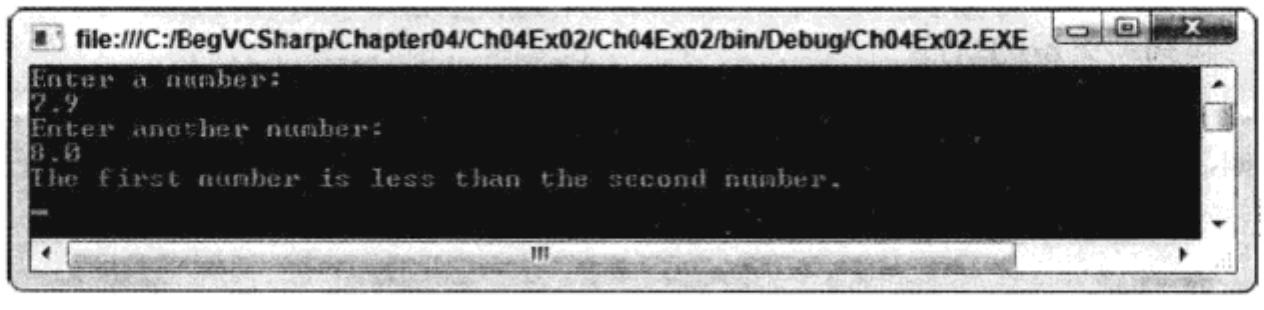


图 4-2

示例的说明

我们已经很熟悉了代码的第一部分，它从用户输入中得到两个 double 值：

```
string comparison;
Console.WriteLine("Enter a number:");
double var1 = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("Enter another number:");
double var2 = Convert.ToDouble(Console.ReadLine());
```

接着根据 var1 和 var2 的值，把一个字符串赋给 string 变量 comparison。首先看看 var1 是否小于 var2：

```
if (var1 < var2)
   comparison = "less than";
```

如果不是，则 var1 大于或等于 var2。在第一个比较操作的 else 部分，需要嵌套第二个比较：

```
else
 {
   if (var1 == var2)
      comparison = "equal to";
```

只有在 var1 大于 var2 时，才执行第二个比较操作中的 else 部分：

```
   else
       comparison = "greater than";
}
```

最后将比较操作的值写到控制台上：

```
Console.WriteLine("The first number is {0} the second number.",
                  comparison);
```

这里使用的嵌套只是进行这些比较的一种方式，还可以编写如下代码：

```
if (var1 < var2)
   comparison = "less than";
if (var1 == var2)
   comparison = "equal to";
if (var1 > var2)
    comparison = "greater than";
```

这个方式的缺点在于无论 var1 和 var2 的值是什么，都要执行 3 个比较操作。在第一种方式中，如果 var1 < var2 是 true，就只执行一个比较，否则就要执行两个比较操作(还执行了 var1 == var2 比较操作)，这样将使执行的代码行较少。其性能上的差异比较小，但在较重视速度的应用程序中，性能的差异就很明显了。

使用 if 语句判断更多的条件

在上面的示例中，有 3 个条件涉及到 var1 的值，包括了这个变量所有可能的值。有时要检查特定的值，例如，var1 是否等于 1、2、3 或 4 等。使用上面那样的代码会得到很多烦人的嵌套代码：

```
if (var1 == 1)
{
   // Do something.
}
else
{
   if (var1 == 2)
   {
      // Do something else.
   }
   else
   {
      if (var1 == 3 || var1 == 4)
      {
         // Do something else.
      }
      else
      {
         // Do something else.
      }
   }
}
```

常见错误：常会错误地将诸如 if(var1== 3 || var1 == 4)的条件写为 if (var1 == 3 || 4)。由于运算符有优先级，因此先执行==运算符，接着用||运算符处理布尔和数值操作数，就会出现错误。

在这些情况下，就要使用稍有不同的缩进模式，缩短 else 代码块(即在 else 块的后面使用一行代码，而不是一个代码块)，这样就得到 else if 语句结构。

```
if (var1 == 1)
{
   // Do something.
}
else if (var1 == 2)
{
   // Do something else.
}
else if (var1 == 3 || var1 == 4)
{
   // Do something else.
}
else
{
   // Do something else.
}
```

这些 else if 语句实际上是两个独立语句，它们的功能与上述代码相同。但更便于阅读。像这样进行多个比较的操作，应考虑使用另一种分支结构：switch 语句。

4.3.3 switch 语句

switch 语句非常类似于 if 语句，因为它也是根据测试的值来有条件地执行代码。但是，switch 语句可以一次将测试变量与多个值进行比较，而不是仅测试一个条件。这种测试仅限于离散的值，而不是像“大于 X”这样的子句，所以它的用法有点不同，但它仍是一种强大的技术。

switch 语句的基本结构如下：

```
switch (<testVar>)
{
   case <comparisonVal1>:
      <code to execute if <testVar> == <comparisonVal1> >
      break;
   case <comparisonVal2>:
      <code to execute if <testVar> == <comparisonVal2> >
      break;
   ...
   case <comparisonValN>:
      <code to execute if <testVar> == <comparisonValN> >
      break;
   default:
      <code to execute if <testVar> != comparisonVals>
      break;
}
```

<testVar>中的值与每个<comparisonValX>值(在 case 语句中指定)进行比较，如果有一个匹配，就执行为该匹配提供的语句。如果没有匹配，就执行 default 部分中的代码。

执行完每个部分中的代码后，还需有另一个语句 break。在执行完一个 case 块后，再执行第二个 case 语句是非法的。

在此，C#与 C++是有区别的，在 C++中，可以在运行完一个 case 语句后，运行另一个 case 语句。

这里的 break 语句将中断 switch 语句的执行，而执行该结构后面的语句。

在 C#代码中，还有一种方法可以防止程序流程从一个 case 语句转到下一个 case 语句。即使用 return 语句，中断当前函数的运行，而不是仅中断 switch 结构的执行(详见第 6 章)。也可以使用 goto 语句(如前所述)，因为 case 语句实际上是在 C#代码中定义的标签。例如：

```
switch (<testVar>)
{
   case <comparisonVal1>:
      <code to execute if <testVar> == <comparisonVal1> >
      goto case <comparisonVal2>;
   case <comparisonVal2>:
      <code to execute if <testVar> == <comparisonVal2> >
      break;
   ...
```

一个 case 语句处理完后，不能自由进入下一个 case 语句，但这个规则有一个例外。如果把多个 case 语句放在一起(堆叠它们)，其后加一个代码块，实际上是一次检查多个条件。如果满足这些条件中的任何一个，就会执行代码，例如：

```
switch (<testVar>)
{
   case <comparisonVal1>:
   case <comparisonVal2>:
      <code to execute if <testVar> == <comparisonVal1> or
                          <testVar> == <comparisonVal2> >
      break;
   ...
```

注意，这些条件也应用到 default 语句。default 语句不一定要放在比较操作列表的最后，还可以把它和 case 语句放在一起。用 break、goto 或 return 添加一个断点，可以确保在任何情况下，该结构都有一个有效的执行路径。

每个<comparisonValX>都必须是一个常数值。一种方法是提供字面值，例如：

```
switch (myInteger)
{
   case 1:
      <code to execute if myInteger == 1 >
      break;
   case -1:
      <code to execute if myInteger == -1 >
      break;
   default:
      <code to execute if myInteger != comparisons>
      break;
}
```

另一种方式是使用常量。常量与其他变量一样，但有一个重要的区别：它们包含的值是固定不变的。一旦给常量指定一个值，该常量在代码执行的过程中，其值一直不变。在这里使用常量是很方便的，因为它们通常更便于阅读，在比较时，看不到要比较的实际值。

声明常量需要指定变量类型和关键字 const，同时必须给它们赋值，例如：

```
const int intTwo = 2;
```

这行代码是有效的，但如果编写如下代码：

```
const int intTwo;
intTwo = 2;
```

就会产生一个编译错误。如果在最初的赋值之后，试图通过任何方式改变常量的值，也会出现编译错误。

在下面的示例中，将使用 switch 语句，根据用户为测试字符串输入的值，将不同的字符串写到控制台上。

试一试：使用 switch 语句

(1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新控制台应用程序 Ch04Ex03。

(2) 把下述代码添加到 Program.cs 中：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   const string myName = "karli";
   const string sexyName = "angelina";
   const string sillyName = "ploppy";
   string name;
   Console.WriteLine("What is your name?");
   name = Console.ReadLine();
   switch (name.ToLower ())
   {
      case myName:
         Console.WriteLine("You have the same name as me!");
         break;
      case sexyName:
         Console.WriteLine("My, what a sexy name you have!");
         break;
      case sillyName:
         Console.WriteLine("That's a very silly name.");
         break;
   }
   Console.WriteLine("Hello {0}!", name);
   Console.ReadKey();
}
```

代码段 Ch04Ex03\Program.cs

(3) 执行代码，输入一个姓名，结果如图 4-3 所示。

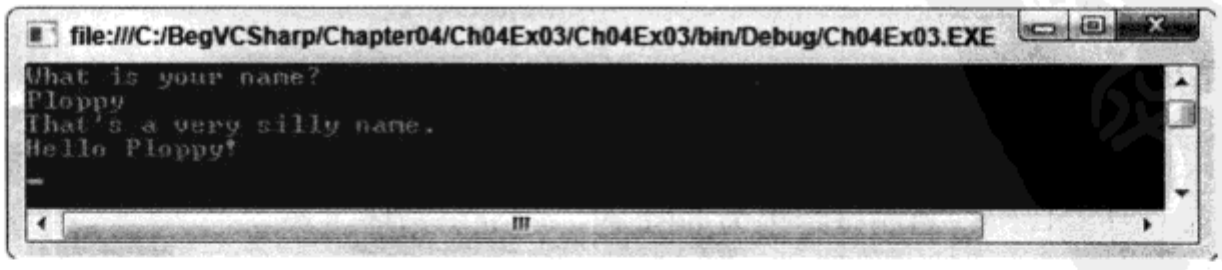


图 4-3

示例的说明

这段代码建立了 3 个常量字符串，接受用户输入的一个字符串，再根据输入的字符串把文本写到控制台上。这里，字符串是用户输入的姓名。

在比较输入的姓名(在变量 name 中)和常量值时，首先要用 name.ToLower()把输入的姓名转换为小写。name.ToLower()是一个标准命令，可用于处理所有的字符串变量，在不能确定用户输入的内容时，使用它是很方便的。使用这个技术，字符串 Karli、kArLi、karli 等就会与测试字符串 karli 匹配了。

switch 语句尝试将输入的字符串与定义的常量值进行匹配，如果成功，就会用一条个性化的消息问候用户。如果不匹配，则只简单地问候用户。

switch 语句对 case 语句的数量上没有限制，所以可以扩展这段代码，使之包含自己能想到的每个姓名，但这需要耗费一些时间。

4.4　循环

循环就是重复执行语句。这个技术使用起来非常方便，因为可以对操作重复任意多次(上千次，甚至百万次)，而无需每次都编写相同的代码。

例如，下面的代码计算一个银行账户在 10 年后的金额，假定支付每年的利息，且该账户没有其他款项的存取：

```
double balance = 1000;
double interestRate = 1.05; // 5% interest/year
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
balance *= interestRate;
```

将相同代码编写 10 次很费时间，如果把 10 年改为其他值，又会如何？那就必须把该代码行手工复制需要的次数，这是一件多么痛苦的事！幸运的是，完全不必这样做。使用一个循环就可以对指令执行需要的次数。

循环的另一个重要类型是一直循环到给定的条件满足为止。这些循环比上面描述的循环稍简单些(但也是很有效的)，所以首先从这类循环开始。

4.4.1　do 循环

do 循环以下述方式执行：执行标记为循环的代码，然后进行一个布尔测试，如果测试的结果为 true，就再次执行这段代码。当测试结果为 false 时，就退出循环。

do 循环的结构如下：

```
do
{
   <code to be looped>
} while (<Test>);
```

其中计算<Test>会得到一个布尔值。

while 语句之后必须使用分号。

例如，使用该结构可以把从 1~10 的数字输出到一列上：

```
int i = 1;
do
{
   Console.WriteLine("{0}", i++);
} while (i <= 10);
```

在把 i 的值写到屏幕上后，使用后缀形式的++运算符递增 i 的值，所以需要检查一下 i <= 10，把 10 也包含在输出到控制台的数字中。

下面的示例使用这个结构略微修改一下本节引言中的代码。该段代码计算了一个账户在 10 年后的余额。这次使用一个循环，根据起始的金额和固定利率，计算该账户的金额要花多长时间才能达到某个指定的数值。

试一试：使用 do 循环

(1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新的控制台应用程序 Ch04Ex04。

(2) 把下述代码添加到 Program.cs 中：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   double balance, interestRate, targetBalance;
   Console.WriteLine("What is your current balance?");
   balance = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("What is your current annual interest rate (in %)?");
   interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
   Console.WriteLine("What balance would you like to have?");
   targetBalance = Convert.ToDouble(Console.ReadLine());
   int totalYears = 0;
   do
   {
      balance *= interestRate;
      ++totalYears;
   }
   while (balance < targetBalance);
   Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                     totalYears, totalYears == 1 ? "" : "s", balance);
   Console.ReadKey();
}
```

代码段 Ch04Ex04\Program.cs

(3) 执行代码，输入一些值，示例结果如图 4-4 所示。

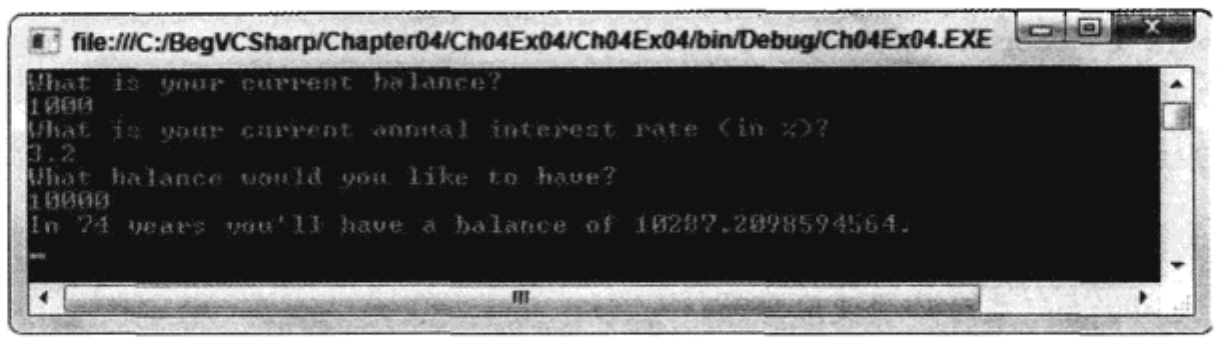


图 4-4

示例的说明

这段代码利用固定的利率，对年度计算余额的过程重复必要的次数，直到满足临界条件为止。在每次循环中，递增一个计数器变量，就可以确定需要多少年：

```
int totalYears = 0;
do
{
   balance *= interestRate;
   ++totalYears;
}
while (balance < targetBalance);
```

然后就可以将这个计数器变量用作输出结果的一部分：

```
Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                  totalYears, totalYears == 1 ? "" : "s", balance);
```

这可能是?: (三元)运算符最常见的用法了——用最少的代码有条件地格式化文本。如果 totalYears 不等于 1，就在 year 后面输出一个 s。

但这段代码并不完美，考虑一下目标余额少于当前余额的情况，则结果应如图 4-5 所示。

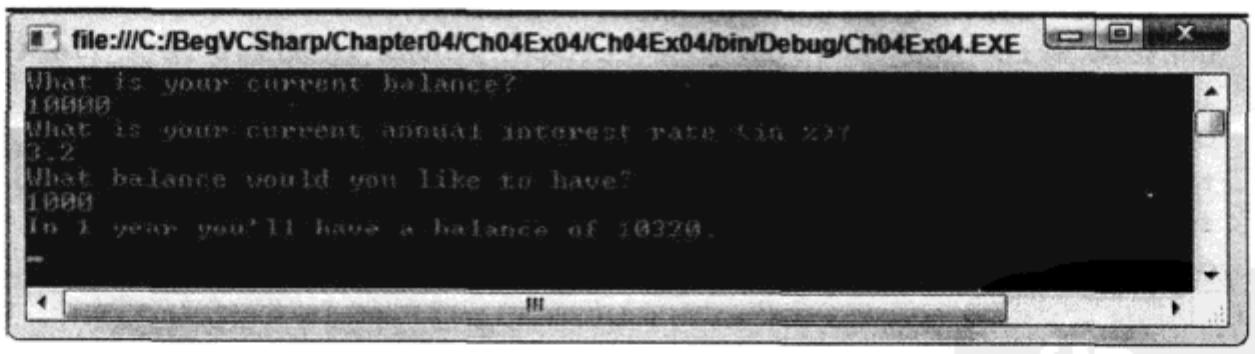


图 4-5

do 循环至少要执行一次。有时(像这种情况)这并不是很理想。当然，可以添加一个 if 语句。

```
int totalYears = 0;
if (balance < targetBalance)
{
  do
   {
```

```
        balance *= interestRate;
        ++totalYears;
      }
      while (balance < targetBalance);
    }
    Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                      totalYears, totalYears == 1? "" : "s", balance);
```

这显然增加了不必要的复杂性。更好的解决方案是使用 while 循环。

4.4.2 while 循环

while 循环非常类似于 do 循环，但有一个明显的区别：while 循环中的布尔测试是在循环开始时进行，而不是最后。如果测试结果为 false，就不会执行循环。程序会直接跳转到循环之后的代码。

按下述方式指定 while 循环：

```
while (<Test>)
{
   <code to be looped>
}
```

它使用的方式与 do 循环几乎完全相同，例如：

```
int i = 1;
while (i <= 10)
{
   Console.WriteLine("{0}", i++);
}
```

这段代码的执行结果与前面的 do 循环相同，它在一列中输出从 1~10 的数字。下面使用 while 循环修改上一个示例。

试一试：使用 while 循环

(1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新的控制台应用程序 Ch04Ex05。

(2) 修改代码，如下所示(开头使用 Ch04Ex04 中的代码，记住删除原来 do 循环最后的 while 语句)：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   double balance, interestRate, targetBalance;
   Console.WriteLine("What is your current balance?");
   balance = Convert.ToDouble(Console.ReadLine());
   Console.WriteLine("What is your current annual interest rate (in %)?");
   interestRate = 1 + Convert.ToDouble(Console.ReadLine()) / 100.0;
   Console.WriteLine("What balance would you like to have?");
   targetBalance = Convert.ToDouble(Console.ReadLine());
   int totalYears = 0;
   while (balance < targetBalance)
   {
      balance *= interestRate;
      ++totalYears;
   }
```

```
      Console.WriteLine("In {0} year{1} you'll have a balance of {2}.",
                        totalYears, totalYears == 1 ? "" : "s", balance);
      if (totalYears == 0)
         Console.WriteLine(
            "To be honest, you really didn't need to use this calculator.");
      Console.ReadKey();
   }
```

代码段 Ch04Ex05\Program.cs

(3) 再次执行代码，但这次使用少于起始余额的目标余额，如图 4-6 所示。

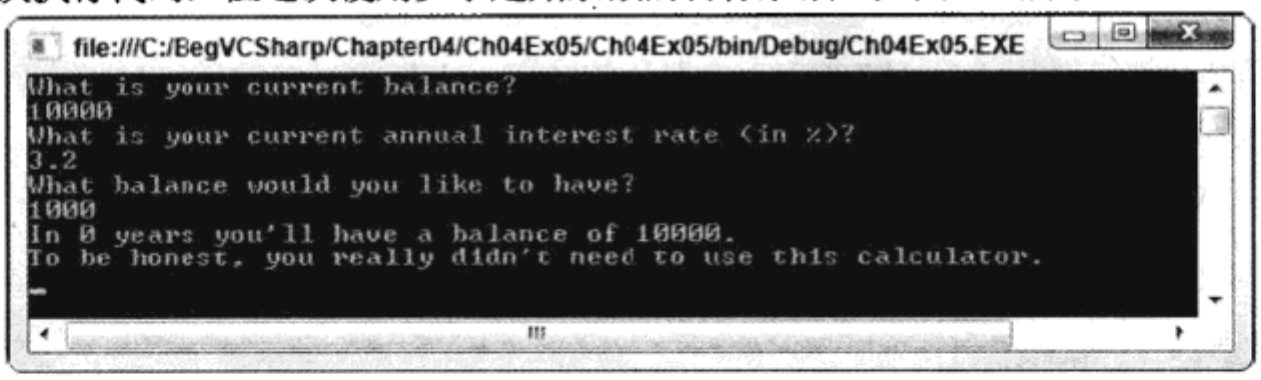


图 4-6

示例的说明

这段代码只是把 do 循环改为 while 循环，就解决了上一个示例中的问题。把布尔测试移到开头，就考虑了不需要执行循环的情况，可以直接跳转到输出结果上。

当然，这种情况还有一个解决方案。例如，可以检查用户输入，确保目标余额大于起始余额。此时，可以把用户输入部分放在循环中，如下所示：

```
      Console.WriteLine("What balance would you like to have?");
      do
      {
         targetBalance = Convert.ToDouble(Console.ReadLine());
         if (targetBalance <= balance)
            Console.WriteLine("You must enter an amount greater than " +
                              "your current balance!\nPlease enter another value.");
      }
      while (targetBalance <= balance);
```

这将拒绝接受无意义的值，得到如图 4-7 所示的结果。

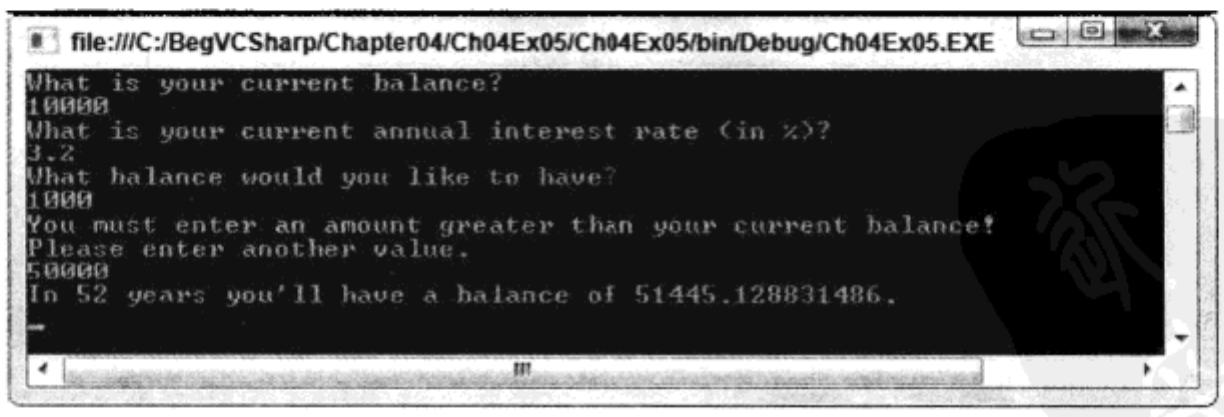


图 4-7

在设计应用程序时，用户输入的有效性检查是一个很重要的主题，本书将提供更多这方面的示例。

4.4.3 for 循环

本章介绍的最后一类循环是 for 循环。这类循环可以执行指定的次数，并维护它自己的计数器。要定义 for 循环，需要下列信息：

- 初始化计数器变量的一个起始值。
- 继续循环的条件，它应涉及到计数器变量。
- 在每次循环的最后，对计数器变量执行一个操作。

例如，如果要在循环中，使计数器从 1 递增到 10，递增量为 1，则起始值为 1，条件是计数器小于或等于 10，在每次循环的最后，要执行的操作是给计数器加 1。

这些信息必须放在 for 循环的结构中，如下所示：

```
for (<initialization>; <condition>; <operation>)
{
   <code to loop>
}
```

它的工作方式与下述 while 循环完全相同：

```
<initialization>
while (<condition>)
{
   <code to loop>
   <operation>
}
```

但 for 循环的格式使代码更易于阅读，因为其语法是在一个地方包括循环的全部规则，而不是把几个语句放在代码的不同地方。

前面使用 do 和 while 循环输出了从 1~10 的数字。下面看看如何使用 for 循环完成这个任务：

```
int i;
for (i = 1; i <= 10; ++i)
{
   Console.WriteLine("{0}", i);
}
```

计数器变量是一个整数 i，它的初始值是 1，在每次循环的最后递增 1。在每次循环过程中，把 i 的值写到控制台上。

注意，当 i 的值为 11 时，将执行循环后面的代码。这是因为在 i 等于 10 的循环末尾，i 会递增为 11。这是在测试条件 i <= 10 之前发生的，此时循环结束。与 while 循环一样，在第一次执行前，只在条件测定为 true 时才执行 for 循环，所以可能根本就不会执行循环中的代码。

最后要注意的是，可以把计数器变量声明为 for 语句的一部分，重新编写上述代码，如下所示：

```
for (int i = 1; i <= 10; ++i)
{
   Console.WriteLine("{0}", i);
}
```

但如果这么做，就不能在循环外部使用变量 i (参见第 6 章中的“变量作用域”一节)。

下面介绍一个使用 for 循环的示例。我们已经使用了几个循环，所以这个示例比较有趣：它将

显示一个 Mandelbrot 集合(使用纯文本字符，看起来不会那么吸引人)!

试一试：使用 for 循环

(1) 在目录 C:\BegVCSharp\Chapter04 中创建一个新的控制台应用程序 Ch04Ex06。

(2) 将以下代码添加到 Program.cs 中：


```
static void Main(string[] args)
{
   double realCoord, imagCoord;
   double realTemp, imagTemp, realTemp2, arg;
   int iterations;
   for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
   {
      for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
      {
         iterations = 0;
         realTemp = realCoord;
         imagTemp = imagCoord;
         arg = (realCoord * realCoord) + (imagCoord * imagCoord);
         while ((arg < 4) && (iterations < 40))
         {
            realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp)
                        - realCoord;
            imagTemp = (2 * realTemp * imagTemp) - imagCoord;
            realTemp = realTemp2;
            arg = (realTemp * realTemp) + (imagTemp * imagTemp);
            iterations += 1;
         }
         switch (iterations % 4)
         {
            case 0:
               Console.Write(".");
               break;
            case 1:
               Console.Write("o");
               break;
            case 2:
               Console.Write("O");
               break;
            case 3:
               Console.Write("@");
               break;
         }
      }
      Console.Write("\n");
   }
   Console.ReadKey();
}
```

代码段 Ch04Ex06\Program.cs

(3) 执行代码，结果如图 4-8 所示。

示例的说明

这里不打算详细说明如何计算 Mandelbrot 集合，而是解释为什么需要在这段代码中使用循环。如果你对数学不感兴趣，可以快速浏览下面两段，因为它们对代码的理解非常重要。

Mandelbrot 集合中的每个位置都对应于公式 N = x + y*i 中的一个复数。实数部分是 x，虚数部分是 y，i 是 -1 的平方根。图像中各个位置的 x 和 y 坐标对应于复数的 x 和 y 部分。

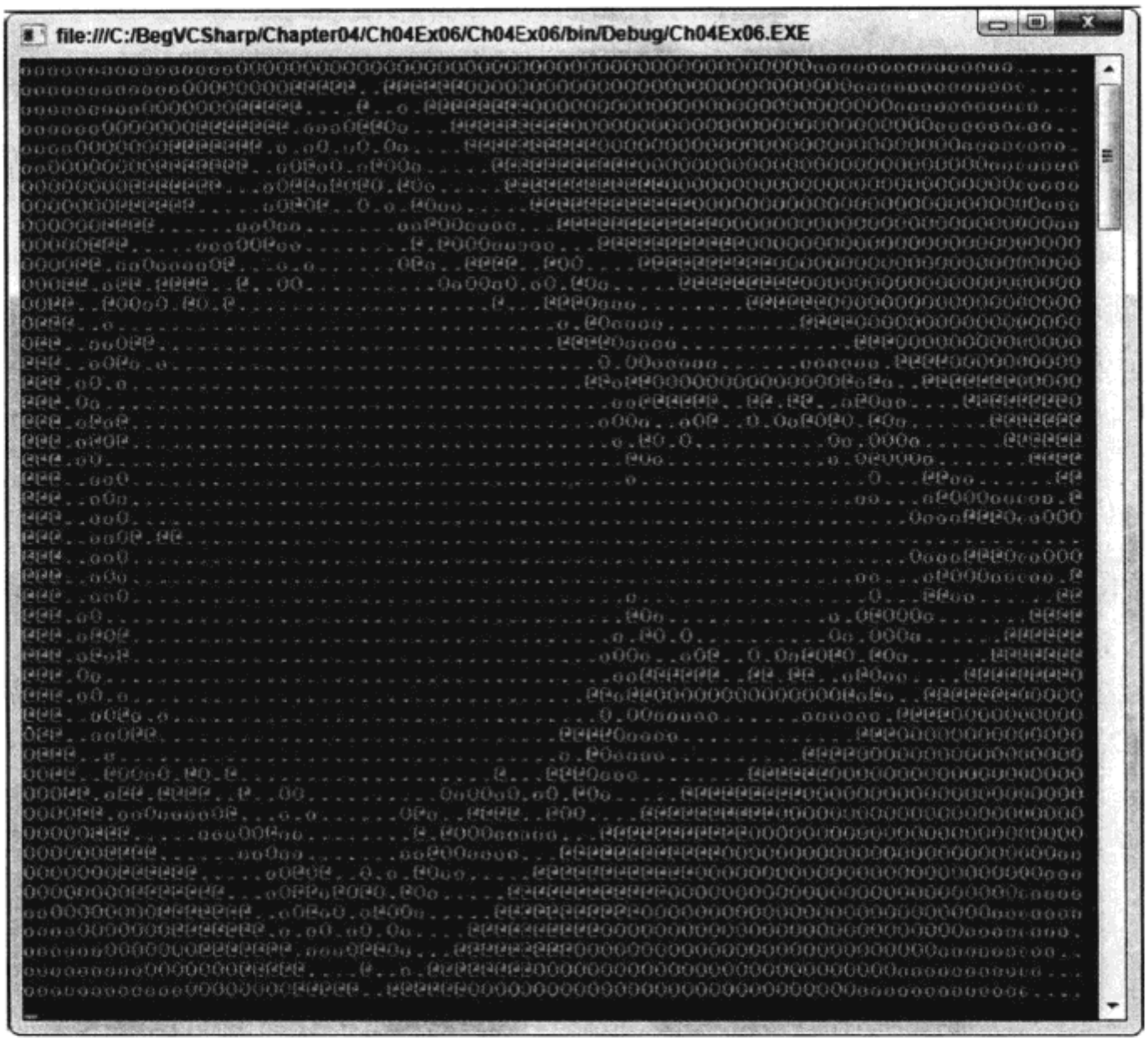


图 4-8

图像中的每个位置用参数 N 来表示，它是 x*x + y*y 的平方根。如果这个值大于或等于 2，则这个数字对应的位置值是 0。如果参数 N 的值小于 2，就把 N 的值改为 N*N - N(即 N = (x*x - y*y - x) + (2*x*y - y)*i))，并再次测试这个新 N 值。如果这个值大于或等于 2，则这个数字对应的位置值是 1。这个过程将一直继续下去，直到给图像中的位置赋一个值，或迭代执行的次数超过指定的次数为止。

根据给图像中每个点赋予的值，在图形环境下，屏幕上会显示某种颜色的像素。但是，本例使用的是文本环境，所以屏幕上显示的是一个字符。

下面看看代码，以及其中的循环。首先声明计算过程中需要的变量：

```
double realCoord, imagCoord;
 double realTemp, imagTemp, realTemp2, arg;
 int iterations;
```

其中 realCoord 和 imagCoord 是 N 的实数和虚数部分，其他 double 变量是计算过程中的临时信息。Iterations 记录在参数 N(arg)等于或大于 2 之前的迭代次数。

接着是两个 for 循环，迭代图像中所有点的坐标(使用比++ 或– –略复杂一些的语法来修改计数器，这是一种常见的功能强大的技术)：

```
for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
{
   for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
   {
```

这里选择合适的边界来显示 Mandelbrot 图像的主要部分。如果要放大这个图像，可以放大这些边界。

在这两个循环中，代码处理 Mandelbrot 图像中的一个点，给 N 指定一个值，这段代码执行要求的迭代计算，给定当前点的测试值。

首先初始化一些变量：

```
iterations = 0;
realTemp = realCoord;
imagTemp = imagCoord;
arg = (realCoord * realCoord) + (imagCoord * imagCoord);
```

接着用 while 循环执行迭代。使用 while 循环，而不是 do 循环，是为了防止 N 的初始值大于 2，如果 N 大于 2，iterations = 0 就是需要的答案，不再需要计算了。

注意这里没有全面计算参数，而仅获取 x*x + y*y 的值，并检查该值是否小于 4。这样简化了计算，因为 2 是 4 的平方根，不需要计算平方根。

```
while ((arg < 4) && (iterations < 40))
{
   realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp)
               - realCoord;
   imagTemp = (2 * realTemp * imagTemp) - imagCoord;
   realTemp = realTemp2;
   arg = (realTemp * realTemp) + (imagTemp * imagTemp);
   iterations += 1;
}
```

这个循环计算上述的数值，其最大迭代数是 40。

把当前点的值存储在 iterations 中后，再使用 switch 语句选择要输出的字符。这里只使用 4 个不同字符，而不是 40 个，且使用求余运算符(%)，这样 0、4、8 等使用一个字符，1、5、9 等使用另一个字符，依次类推：

```
switch (iterations % 4)
{
   case 0:
      Console.Write(".");
      break;
   case 1:
      Console.Write("o");
      break;
   case 2:
```

```
            Console.Write("O");
            break;
         case 3:
            Console.Write("@");
            break;
      }
```

注意这里使用的是Console.Write()，而不是Console.WriteLine()，因为每次输出一个字符时，并不需要从一个新行开始。在最内层的一个 for 循环结束后，需要结束一行，所以使用前面介绍的转义序列输出行结束符。

```
      }
      Console.Write("\n");
   }
```

这样，每行都与下一行分隔开来，并进行适当的排列。这个应用程序的最终结果尽管不是很漂亮，也能给人留下深刻的印象。它说明了循环和分支的用途。

4.4.4 循环的中断

有时需要更精细地控制循环代码的处理。C#为此提供了 4 个命令，其中的 3 个已经在其他情形中介绍过了：

- break——立即终止循环。
- continue——立即终止当前的循环(继续执行下一次循环)。
- goto——可以跳出循环，到已标记好的位置上(如果希望代码易于阅读和理解，最好不要使用该命令)。
- return——跳出循环及其包含的函数(参见第 6 章)。

break 命令可退出循环，继续执行循环后面的第一行代码，例如：

```
int i = 1;
while (i <= 10)
{
   if (i == 6)
      break;
   Console.WriteLine("{0}", i++);
}
```

这段代码输出 1~5 的数字，因为 break 命令在 i 的值为 6 时退出循环。

continue 仅终止当前的循环，而不是整个循环，例如：

```
int i;
for (i = 1; i <= 10; i++)
{
   if ((i % 2) == 0)
      continue;
   Console.WriteLine(i);
}
```

在上面的示例中，只要 i 除以 2 的余数是 0，continue 语句就终止当前的循环，所以只显示数字 1、3、5、7 和 9。

第 3 个方法使用前面的 goto 语句，例如：

```
int i = 1;
while (i <= 10)
{
   if (i == 6)
      goto exitPoint;
   Console.WriteLine("{0}", i++);
}
Console.WriteLine("This code will never be reached.");
exitPoint:
Console.WriteLine("This code is run when the loop is exited using goto.");
```

注意，使用 goto 语句退出循环是合法的(但会有点杂乱)，但使用 goto 语句从外部进入循环是非法的。

4.4.5　无限循环

可以通过编写错误代码或错误的设计，定义永不终止的循环，即所谓的无限循环。例如，下面的代码：

```
while (true)
{
   // code in loop
}
```

有时这种代码也是有用的，使用 break 语句或者手工使用 Windows 任务管理器总是可以退出这样的循环。但是，当这种情形偶然出现时，就会出问题。考虑下面的循环，它与上一节的 for 循环非常类似：

```
int i = 1;
while (i <= 10)
{
   if ((i % 2) == 0)
      continue;
   Console.WriteLine("{0}", i++);
}
```

i 是在循环的最后一行代码执行完后才递增的，即在 continue 语句执行完后递增。但在执行到这个 continue 语句(此时 i 为 2)时，程序会用相同的 i 值进行下一个循环，然后测试这个 i 值，继续循环，一直这样下去。这就冻结了应用程序。注意仍可以用一般方式退出已冻结的应用程序，所以此时不必重新启动计算机。

4.5　小结

本章介绍了可以在代码中使用的各种结构，扩展了您的编程知识。在开始编写更复杂的应用程序时，这些结构的正确使用是非常重要的。

首先用一定的篇幅介绍了布尔逻辑，以及一些按位逻辑的知识。在学习了本章的其他内容后，

再回过头来看看这些逻辑，可以确信，在谈到执行程序中的分支和循环代码时，这个主题是非常重要的。熟悉本节讨论的运算符和技术是很有必要的。

分支结构可以有条件地执行代码，当分支与循环一起使用时，可以在C#代码中创建出比较复杂的结构。把循环嵌套起来，再放在if结构中，就会发现代码的缩进是非常有用的。如果把所有代码都移到屏幕左端，就很难分析它们了，甚至难以调试。此时应确保代码的缩进——用户在以后使用时即可体会到它的种种优势。VS为此做了大量的工作，但最好在输入代码时进行缩进。

第5章将深入探讨变量。

4.6 练习

(1) 如果两个整数存储在变量var1 和 var2中，该进行什么样的布尔测试，看看其中的一个(但不是两个)是否大于10?

(2) 编写一个应用程序，其中包含练习(1)中的逻辑，要求用户输入两个数字，并显示它们，但拒绝接受两个数字都大于10的情况，并要求用户重新输入。

(3) 下面的代码存在什么错误?

```
int i;
for (i = 1; i <= 10; i++)
{
   if ((i % 2) = 0)
      continue;
   Console.WriteLine(i);
}
```

(4) 修改Mandelbrot集合应用程序，要求用户输入图像的边界，显示选中的图像部分。当前代码输出的字符应正好能放在控制台应用程序的一行上。考虑如何使每个选中的图像正好占据大小相同的空间，以最大化可视区域。

附录A给出了练习答案。

4.7 本章要点

主　题	重要概念
布尔逻辑	布尔逻辑使用布尔值(true 和 false)计算条件。布尔运算符用于比较数值，返回布尔结果。一些布尔运算符也用于对数值的底层位结构执行按位操作，还有一些专门的按位运算符
分支	可以使用布尔逻辑控制程序流。计算为布尔值的表达式可以用于确定是否执行某个代码块，可以使用if语句或?:(三元)运算符进行简单的分支，或者使用switch语句同时检查多个条件
循环	循环允许根据指定的条件多次执行代码块。使用do和while循环可以在布尔表达式为true时执行代码，使用for循环可以在循环代码中包含一个计数器。循环可以使用continue中断当前的迭代，或者使用break完全中断。一些循环只能在用户强制中断时结束，它们称为无限循环

PDG

变量的更多内容

本章内容：

- 如何在类型之间进行隐式和显式转换
- 如何创建和使用枚举类型
- 如何创建和使用结构类型
- 如何创建和使用数组
- 如何处理字符串值

前面介绍了有关 C#语言的一些内容，现在将回顾和讨论与变量相关的其他一些较复杂的论题。

首先要讨论的主题是类型转换，即把值从一种类型转换为另一种类型。前面已经描述了其中的一些信息，这里则要正式讨论。掌握这个论题可以更好地理解表达式中(有意或无意)混合使用的类型，更好地控制处理数据的方式。这有助于理顺代码，避免引起不必要的误解。

接着阐述另外一些类型的变量：

- **枚举**——变量类型，用户定义了一组可能的离散值，这些值可以用人们能理解的方式使用。
- **结构**——合成的变量类型，由用户定义的一组其他变量类型组成。
- **数组**——包含一种类型的多个变量，可以以索引方式访问各个数值。

这些类型比前面使用的简单类型复杂一些，但可以使工作更容易完成。最后，学习另一个与字符串相关的主题——基本字符串处理。

5.1 类型转换

本书前面说过，无论是什么类型，所有的数据都是一系列的位，即一系列 0 和 1。变量的含义是通过解释这些数据的方式来传达的。最简单的示例是 char 类型，这种类型用一个数字表示 Unicode 字符集中的一个字符。实际上，这个数字与 ushort 的存储方式完全相同——它们都存储 0~65535 之间的数字。

但一般情况下，不同类型的变量使用不同的模式来表示数据。这意味着，即使可以把一系列的

位从一种类型的变量移动到另一种类型的变量中(也许它们占用的存储空间相同，也许目标类型有足够的存储空间包含所有的源数据位)，结果也可能与期望的不同。

这并不是数据位从一个变量到另一个变量的一对一映射，而是需要对数据进行类型转换。类型转换采用以下两种形式：

- **隐式转换**：从类型 A 到类型 B 的转换可以在所有情况下进行，执行转换的规则非常简单，可以让编译器执行转换。
- **显式转换**：从类型 A 到类型 B 的转换只能在某些情况下进行，转换的规则比较复杂，应进行某种类型的处理。

5.1.1 隐式转换

隐式转换不需要做任何工作，也不需要另外编写代码。考虑下面的代码：

```
var1 = var2;
```

如果var2 的类型可以隐式地转换为var1 的类型，这个赋值语句就涉及到一个隐式转换。它也可能只处理相同类型的两个变量，不需要隐式转换。例如，ushort 和char 的值是可以互换的，因为它们都可以存储 0~65535 之间的数字，在这两个类型之间可以进行隐式转换，如下面的代码所示：

```
ushort destinationVar;
char sourceVar = 'a';
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

这里存储在 sourceVar 中的值放在 destinationVar 中。在用两个 Console.WriteLine()命令输出变量时，得到如下结果：

```
sourceVar val: a
destinationVar val: 97
```

即使两个变量存储的是相同的信息，使用不同的类型解释它们时，方式也是不同的。

简单类型有许多隐式转换；bool 和 string 没有隐式转换，但数值类型有一些隐式转换。表 5-1 列出了编译器可以隐式执行的数值转换(记住，char 存储的是数值，所以 char 被当作一个数值类型)。

表 5-1

类 型	可以安全地转换为
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal

(续表)

类　型	可以安全地转换为
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

不要担心——不需要记住这个表格，因为很容易看出编译器可以执行哪些隐式转换。第 3 章中的一个表列出了每种简单数字类型的取值范围。这些类型的隐式转换规则是：任何类型 A，只要其取值范围完全包含在类型 B 的取值范围内，就可以隐式转换为类型 B。

其原因是很简单的。如果要把一个值放在变量中，而该值超出了变量的取值范围，就会出问题。例如，short 类型的变量可以存储 0~32767 的数字，而 byte 可以存储的最大值是 255，所以如果要把一个 short 值转换为 byte 值，就会出问题。如果 short 包含的值在 256~32767 之间，相应数值就不能放在 byte 中。

但是，如果 short 类型变量中的值小于 255，就应能转换这个值，对吗？答案是可以。具体地说是可以，但必须使用显式转换。执行显式转换有点类似于“我已经知道你对我这么做提出了警告，但我将对其后果负责”。

5.1.2　显式转换

顾名思义，在明确要求编译器把数值从一种数据类型转换为另一种数据类型时，就是在执行显式转换。因此，这需要另外编写代码，代码的格式将随着转换方法而异。在学习显式转换代码前，先分析如果不添加任何显式转换代码，会发生什么情况。

例如，下面对上一节的代码进行修改，试着把 short 值转换为 byte：

```
byte destinationVar;
short sourceVar = 7;
destinationVar = sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

如果编译这段代码，就会产生如下错误：

```
Cannot implicitly convert type 'short' to 'byte'. An explicit conversion exists
(are you missing a cast?)
```

幸运的是，C#编译器可以检测出没有进行显式转换！

为了成功编译这段代码，需要添加代码，进行显式转换。最简单的方式是把 short 变量强制转换为 byte(由上述错误字符串提出)。强制转换就是强迫数据从一种类型转换为另一种类型，其语法比较简单：

```
<(destinationType)sourceVar>
```

这将把<sourceVar>中的值转换为<destinationType>。

这只在某些情况下是可行的。彼此之间几乎没有什么关系的类型或根本没有关系的类型不能进行强制转换。

因此可以使用这个语法修改示例，把 short 变量强制转换为 byte：

```
byte destinationVar;
short sourceVar = 7;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

得到如下结果：

```
sourceVar val: 7
destinationVar val: 7
```

在试图把一个值转换为不合适的变量时，会发生什么呢？修改代码，如下所示：

```
byte destinationVar;
short sourceVar = 281;
destinationVar = (byte)sourceVar;
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

结果如下：

```
sourceVar val: 281
destinationVar val: 25
```

会发生什么？看看这两个数字的二进制表示，以及可以存储在 byte 中的最大值 255：

```
281 = 100011001
 25 = 000011001
255 = 011111111
```

可以看出，源数据的最左边一位丢失了。这会导致一个问题：数据是何时丢失的？显然，当需要显式地把一种数据类型转换为另一种数据类型时，最好能够了解是否有数据丢失了。如果不知道这些，就会发生严重的问题，例如，记账应用程序或确定火箭飞往月球的轨道的应用程序。

一种方式是简单地检查源变量的值，把它与目标变量的取值范围进行比较。还有另一个技术，迫使系统特别注意运行期间的转换。在将一个值放在一个变量中时，如果该值过大，不能放在该类型的变量中，就会导致溢出，这就需要检查。

这里要用到两个关键字 checked 和 unchecked，称为表达式的溢出检查上下文。以下述方式使用这两个关键字：

```
checked(expression)
unchecked(expression)
```

下面对上一个示例进行溢出检查：

```
byte destinationVar;
short sourceVar = 281;
destinationVar = checked((byte)sourceVar);
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);
```

在执行这段代码时，程序会崩溃，并显示如图 5-1 所示的错误信息(在 OverflowCheck 项目中编译这段代码)。

```
byte destinationVar;
short sourceVar = 281;
destinationVar = checked((byte)sourceVar);
Console.WriteLine("sourceVar val: {0}", sourceVar);
Console.WriteLine("destinationVar val: {0}", destinationVar);

Console.ReadKey();
```

OverflowException was unhandled

Arithmetic operation resulted in an overflow.

Troubleshooting tips:

Make sure you are not dividing by zero.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

图　5-1

但是，在这段代码中，如果用 unchecked 替代 checked，就会得到与以前一样的结果，不会出现错误。这与前面的默认做法是一样的。

除了这两个关键字以外，还可以配置应用程序，让这种类型的表达式都包含 checked 关键字，除非表达式明确使用 unchecked 关键字(换言之，可以改变溢出检查的默认设置)。为此，应修改项目的属性：在 VS 中右击 Solution Explorer 窗口中的项目，选择 Properties 选项。单击窗口左边的 Build，打开 Build 设置，如图 5-2 所示。

图　5-2

要修改的属性是一个 Advanced 设置，所以单击 Advanced 按钮。在打开的对话框中，选中 Check for arithmetic overflow/underflow 选项，如图 5-3 所示。默认情况下禁用这个设置，激活它可以进行上述 checked 操作。

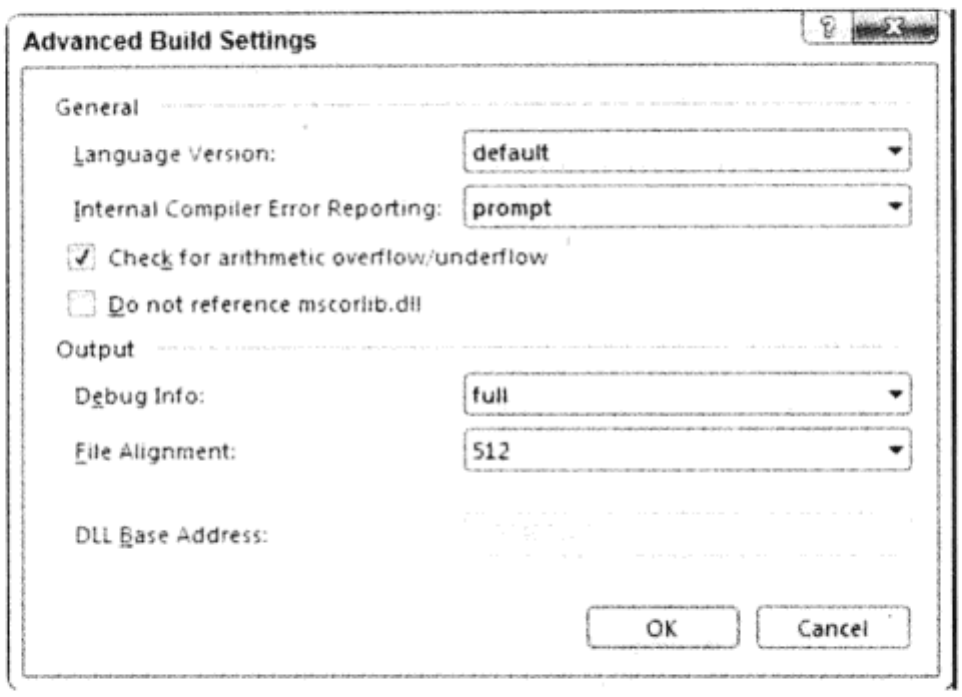


图　5-3

5.1.3　使用 Convert 命令进行显式转换

本书的许多“试一试”示例中使用的显式类型转换，与本章前面的示例有一些区别。前面使用 Convert .ToDouble()等命令把字符串值转换为数值，显然，这种方式并不适用于所有字符串。

例如，如果使用 Convert.ToDouble()把诸如 Number 的字符串转换为一个 double 值，执行代码，就会看到如图 5-4 所示的对话框。

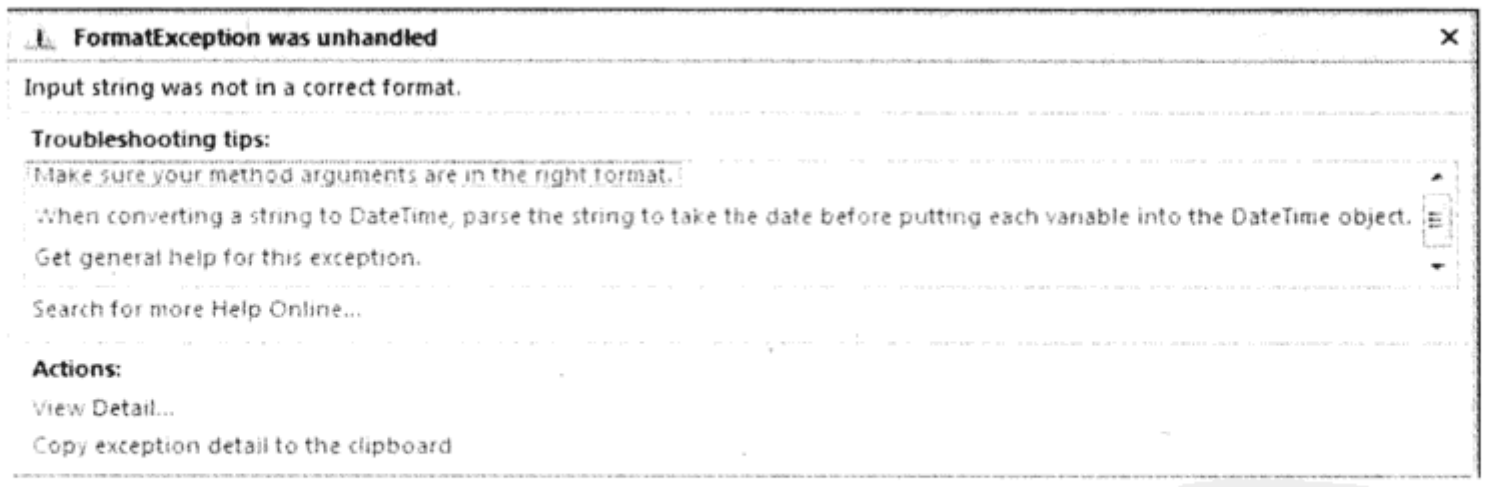


图　5-4

可以看出，执行失败。为了成功执行此类转换，所提供的字符串必须是数值的有效表达方式，该数还必须是不会溢出的数。数值的有效表达方式是：首先是一个可选符号(加号或减号)，然后是 0 位或多位数字，一个句点后跟一位或多位数字，接着是一个可选的 e 或 E，后跟一个可选符号和一位或多位数字(在这个序列之前或之后)和空格。利用这些可选的额外数据，就可以看出 - 1.2451e - 24 这样复杂的字符串是一个数值。

按这种方式可以进行许多显式转换，如表 5-2 所示。

表 5-2

命　令	结　果
Convert. ToBoolean(val)	val 转换为 bool
Convert. ToByte(val)	val 转换为 byte
Convert. ToChar(val)	val 转换为 char
Convert. ToDecimal(val)	val 转换为 decimal
Convert. ToDouble(val)	val 转换为 double
Convert. ToInt16(val)	val 转换为 short
Convert. ToInt32(val)	val 转换为 int
Convert. ToInt64(val)	val 转换为 long
Convert. ToSByte(val)	val 转换为 sbyte
Convert. ToSingle(val)	val 转换为 float
Convert. ToString(val)	val 转换为 string
Convert. ToUInt16(val)	val 转换为 ushort
Convert. ToUInt32(val)	val 转换为 uint
Convert. ToUInt64(val)	val 转换为 ulong

其中 val 可以是大多数变量类型(如果这些命令不能处理该类型的变量，编译器就会告诉用户)。

但如表 5-2 所示，转换的名称略不同于 C#类型名称，例如，要转换为 int，应使用 Convert.ToInt32()。这是因为这些命令来自于.NET Framework 的 System 名称空间，而不是本机 C#本身。这样它们就可以在除 C#以外的其他.NET 兼容语言中使用。

对于这些转换要注意的一个问题是，它们总是要进行溢出检查，checked 和 unchecked 关键字以及项目属性设置不起作用。

下面的示例包括本节介绍的许多转换类型。它声明和初始化许多不同类型的变量，再在它们之间进行隐式和显式转换。

试一试：类型转换的实践

(1) 在 C:\BegVCSharp\Chapter05 目录中创建一个新控制台应用程序 Ch05Ex01。

(2) 把下述代码添加到 Program.cs 中：


可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   short   shortResult, shortVal = 4;
   int     integerVal = 67;
   long    longResult;
   float   floatVal = 10.5F;
   double  doubleResult, doubleVal = 99.999;
   string  stringResult, stringVal = "17";
   bool    boolVal = true;

  Console.WriteLine("Variable Conversion Examples\n");
```

```
        doubleResult = floatVal * shortVal;
        Console.WriteLine("Implicit, -> double: {0} * {1} -> {2}", floatVal,
          shortVal, doubleResult);

        shortResult = (short)floatVal;
        Console.WriteLine("Explicit, -> short:  {0} -> {1}", floatVal,
          shortResult);

        stringResult = Convert.ToString(boolVal) +
          Convert.ToString(doubleVal);
        Console.WriteLine("Explicit, -> string: \"{0}\" + \"{1}\" -> {2}",
          boolVal, doubleVal, stringResult);

        longResult = integerVal + Convert.ToInt64(stringVal);
        Console.WriteLine("Mixed,    -> long:   {0} + {1} -> {2}",
          integerVal, stringVal, longResult);
        Console.ReadKey();
    }
```

代码段 Ch05Ex01\Program.cs

(3) 执行代码，结果如图 5-5 所示。

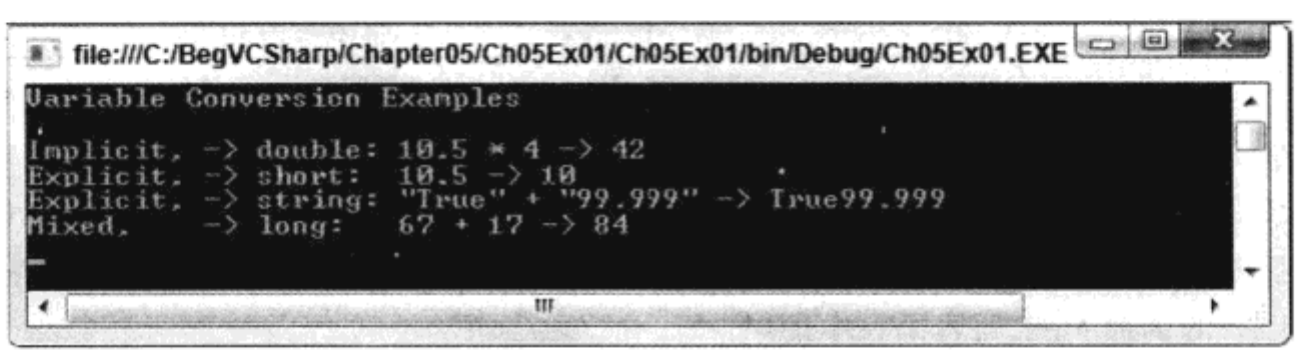


图　5-5

示例的说明

这个示例包含前面介绍的所有转换类型，既有像前面简短代码示例中的简单赋值，也有在表达式中进行的转换。必须考虑这两种情况，因为每个非一元运算符的处理都可能要进行类型转换，而不仅仅是赋值运算符。例如：

```
shortVal * floatVal
```

其中把一个 short 值与一个 float 值相乘。在这样的指令中，没有指定显式转换，应在可能的情况下进行隐式转换。在这个示例中，唯一有意义的隐式转换是把 short 值转换为 float(因为把 float 值转换为 short 需要显式转换)，所以这里使用隐式转换。

也可以重新编写这个过程，使用下述代码：

```
shortVal * (short)floatVal
```

这并不表示两个 short 相乘的结果将返回一个 short 值。因为这个操作很可能大于 32767(这是 short 可以包含的最大值)，所以这个操作的结果实际上是 int。

使用这个数据类型转换语法执行显式转换，其运算符的优先级与其他一元运算符一样，都是优先级中的最高级，如++(用作前缀)。

如果语句涉及混合类型，就根据运算符的优先级，在处理每个运算符时执行转换。这意味着可能出现“中间”转换，例如：

```
doubleResult = floatVal + (shortVal * floatVal);
```

要处理的第一个运算符是*，如上所述，它将把 shortVal 转换为 float。接着处理+运算符，它不需要进行任何转换，因为这是把两个 float 值相加 (floatVal 和 shortVal * floatVal 的 float 结果)。在最后处理＝运算符时，这个计算的 float 结果转换为 double。

这个转换过程初看起来比较复杂，但只要按照运算符的优先级，把表达式分解为不同的部分，就可以弄明白这个过程。

5.2　复杂的变量类型

除了这些简单的变量类型之外，C#还提供了 3 个较复杂(但非常有用)的变量：枚举、结构和数组。

5.2.1　枚举

本书迄今介绍的每种类型(除了 string 外)都有明确的取值范围。诚然，有些类型(如 double)的取值范围非常大，可以看作是连续的，却是一个固定的集合。最简单的示例是 bool 类型，它只能取两个值：true 或 false。

有时希望变量提取的是一个固定集合中的值。例如，orientation 类型可以存储 north、south、east 或 west 中的一个值。

此时可以使用枚举类型。枚举就可以完成这个 orientation 类型的任务：它们允许定义一个类型，其中包含提供的限定值集合中的一个值。所以，需要创建自己的枚举类型 orientation，它可以从上述 4 个值中提取一个值。

注意有一个附加的步骤——不是仅仅声明一个给定类型的变量，而是声明和描述一个用户定义的类型，再声明这个新类型的变量。

定义枚举

可以使用 enum 关键字来定义枚举，如下所示：

```
enum <typeName>
{
    <value1>,
    <value2>,
    <value3>,
    ...
    <valueN>
}
```

接着声明这个新类型的变量：

```
<typeName> <varName>;
```

并赋值：

```
<varName> = <typeName>.<value>;
```

枚举使用一个基本类型来存储。枚举类型可以提取的每个值都存储为该基本类型的一个值，默认情况下该类型为 int。在枚举声明中添加类型，就可以指定其他基本类型：

```
enum <typeName> : <underlyingType>
{
    <value1>,
    <value2>,
    <value3>,
    ...
    <valueN>
}
```

枚举的基本类型可以是 byte、sbyte、short、ushort、int、uint、long 和 ulong。

在默认情况下，每个值都会根据定义的顺序(从 0 开始)，自动赋给对应的基本类型值。这意味着 value1 的值是 0，value2 的值是 1，value3 的值是 2 等。可以重写这个赋值过程：使用＝运算符，并指定每个枚举的实际值：

```
enum <typeName> : <underlyingType>
{
    <value1> = <actualVal1>,
    <value2> = <actualVal2>,
    <value3> = <actualVal3>,
    ...
    <valueN> = <actualValN>
}
```

还可以使用一个值作为另一个枚举的基础值，为多个枚举指定相同的值：

```
enum <typeName> : <underlyingType>
{
    <value1> = <actualVal1>,
    <value2> = <value1>,
    <value3>,
    ...
    <valueN> = <actualValN>
}
```

没有赋值的任何值都会自动获得一个初始值，这里使用的值是从比上一个明确声明的值大 1 开始的序列。例如，在上面的代码中，<value3>的值是<value1> + 1。

注意这可能会产生预料不到的问题，在一个定义(如<value2> = <value1>)后指定的值可能与其他值相同。例如，在下面的代码中，<value4>的值与<value2>相同。

```
enum <typeName> : <underlyingType>
{
    <value1> = <actualVal1>,
    <value2>,
    <value3> = <value1>,
    <value4>,
```

```
    ...
    <valueN> = <actualValN>
}
```

当然，如果这正是希望的结果，则代码就是正确的。还要注意，以循环方式赋值可能会产生错误，例如：

```
enum <typeName> : <underlyingType>
{
    <value1> = <value2>,
    <value2> = <value1>
}
```

下面看一个示例。其代码定义了一个枚举 orientation，然后演示了它的用法。

试一试：使用枚举

(1) 在 C:\BegVCSharp\Chapter05 目录中创建一个新控制台应用程序 Ch05Ex02。

(2) 把下列代码添加到 Program.cs 中：

可从 wrox.com 下载源代码

```
namespace Ch05Ex02
{
   enum orientation : byte
   {
      north = 1,
      south = 2,
      east  = 3,
      west  = 4
   }

   class Program
   {
      static void Main(string[] args)
      {
         orientation myDirection = orientation.north;
         Console.WriteLine("myDirection = {0}", myDirection);
         Console.ReadKey();
      }
   }
}
```

代码段 Ch05Ex02\Program.cs

(3) 运行应用程序，应得到如图 5-6 所示的输出结果。

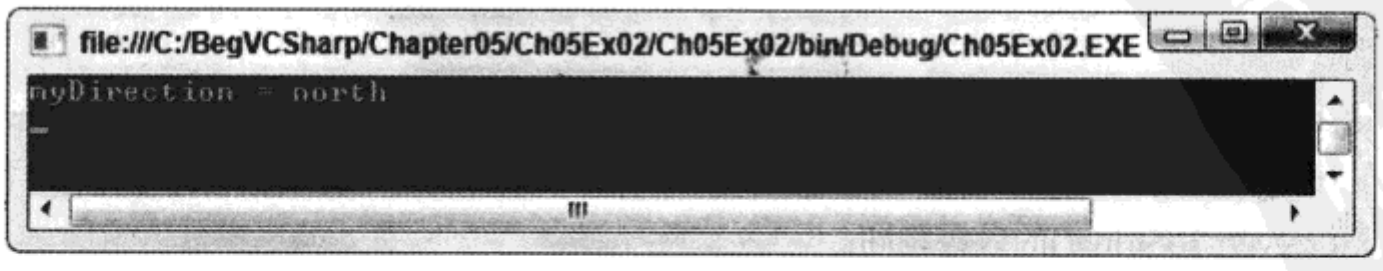


图　5-6

(4) 退出应用程序，修改代码，如下所示：

```
byte directionByte;
string directionString;
orientation myDirection = orientation.north;
Console.WriteLine("myDirection = {0}", myDirection);
directionByte = (byte)myDirection;
directionString = Convert.ToString(myDirection);
Console.WriteLine("byte equivalent = {0}", directionByte);
Console.WriteLine("string equivalent = {0}", directionString);
Console.ReadKey();
```

(5) 再次运行应用程序，输出结果如图 5-7 所示。

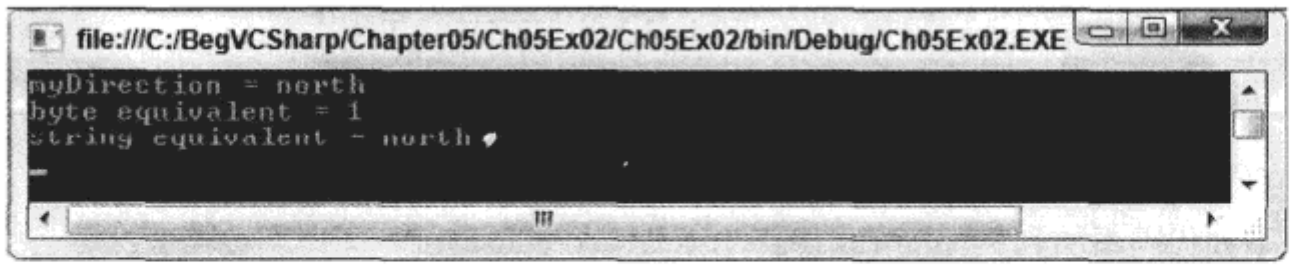


图　5-7

示例的说明

这段代码定义并使用了一个枚举类型 orientation。首先要注意的是，类型定义代码放在名称空间 Ch05Ex02 中，而没有与其余代码放在一起。这是因为在运行期间，定义代码并不是像执行应用程序中的代码那样一行一行地执行。应用程序是从已经习惯的位置开始执行的，并可以访问新类型，因为它属于同一个名称空间。

这个示例的第一个迭代演示了创建新类型的变量，给它赋值以及把它输出到屏幕上的基本方法。接着修改代码，把枚举值转换为其他类型。注意这里必须使用显式转换。即使 orientation 的基本类型是 byte，仍必须使用(byte)强制类型转换，把 myDirection 的值转换为 byte 类型：

```
directionByte = (byte)myDirection;
```

如果要将 byte 类型转换为 orientation，也同样需要进行显式转换。例如，可以使用下述代码将 byte 变量 myByte 转换为 orientation，并把这个值赋给 myDirection：

```
myDirection = (orientation)myByte;
```

当然，这里必须小心，因为并不是所有 byte 类型变量的值都可以映射为已定义的 orientation 值。orientation 类型可以存储其他 byte 值，所以不会直接产生一个错误，但会在应用程序的后面违反逻辑。

要获得枚举的字符串值，可以使用 Convert.ToString()：

```
directionString = Convert.ToString(myDirection);
```

使用(string)强制类型转换是行不通的，因为需要进行的处理并不仅仅是把存储在枚举变量中的数据放在 string 变量中，而是更复杂一些。另外，还可以使用变量本身的 ToString()命令。下面的代码与使用 Convert.ToString()的效果相同：

```
directionString = myDirection.ToString();
```

也可以把 string 转换为枚举值，但其语法稍复杂一些。有一个特定的命令用于此类转换，即 Enum.Parse()，其用法如下：

```
(enumerationType)Enum.Parse(typeof(enumerationType),enumerationValueString);
```

它使用了另一个运算符 typeof，可以得到操作数的类型。对 orientation 类型使用这个命令，如下所示：

```
string myString = "north";
orientation myDirection = (orientation)Enum.Parse(typeof(orientation),
                                                  myString);
```

当然，并非所有的字符串值都会映射为一个 orientation 值。如果传送的一个值不能映射为枚举值中的一个，就会产生错误。与 C#中的其他值一样，这些值是区分大小写的，所以如果字符串与一个值相同，但大小写不同(例如，myString 设置为 North，而不是 north)，就会产生错误。

5.2.2　结构

下一个要介绍的变量类型是结构(struct，structure 的简写)。结构就是由几个数据组成的数据结构，这些数据可能具有不同的类型。根据这个结构，可以定义自己的变量类型。例如，假定要存储从起点开始到某一位置的路径，其中路径由一个方向和一个距离值(英里)组成。为简单起见，假定该方向是指南针上的一点(这样，方向就可以用上一节的 orientation 枚举来表示)，距离值可以用一个 double 类型来表示。

通过前面的代码，可以用两个不同的变量来表示该路径：

```
orientation myDirection;
double      myDistance;
```

像这样使用两个变量，是没有错误的，但在一个地方存储这些信息应该更为简单(特别是在需要多个路由时，就更为简单)。

定义结构

使用 struct 关键字来定义结构，如下所示：

```
struct <typeName>
{
   <memberDeclarations>
}
```

*<memberDeclarations>*部分包含变量的声明(称为结构的数据成员)，其格式与往常一样。每个成员的声明都采用如下形式：

```
<accessibility> <type> <name>;
```

要让调用结构的代码访问该结构的数据成员，可以对<accessibility>使用关键字 public，例如：

```
struct route
{
   public orientation direction;
   public double      distance;
}
```

定义了结构类型后，就可以定义新类型的变量，来使用该结构：

```
route myRoute;
```

还可以通过句点字符访问这个组合变量中的数据成员：

```
myRoute.direction = orientation.north;
myRoute.distance = 2.5;
```

把这个类型放在下面的“试一试”示例中。其中使用上一节的 orientation 枚举和上面的 route 结构，然后在代码中处理这个结构，以便您了解结构的工作原理。

试一试：使用结构

(1) 在 C:\BegVCSharp\Chapter05 目录中创建一个新控制台应用程序 Ch05Ex03。

(2) 把下列代码添加到 Program.cs 中：

可从 wrox.com 下载源代码

```
namespace Ch05Ex03
{
   enum orientation : byte
   {
      north = 1,
      south = 2,
      east  = 3,
      west  = 4
   }
   struct route
   {
      public orientation direction;
      public double      distance;
   }
   class Program
   {
      static void Main(string[] args)
      {
         route myRoute;
         int myDirection = -1;
         double myDistance;
         Console.WriteLine("1) North\n2) South\n3) East\n4) West");
         do
         {
            Console.WriteLine("Select a direction:");
            myDirection = Convert.ToInt32(Console.ReadLine());
         }
         while ((myDirection < 1) || (myDirection > 4));
         Console.WriteLine("Input a distance:");
         myDistance = Convert.ToDouble(Console.ReadLine());
         myRoute.direction = (orientation)myDirection;
         myRoute.distance = myDistance;
         Console.WriteLine("myRoute specifies a direction of {0} and a " +
            "distance of {1}", myRoute.direction, myRoute.distance);
         Console.Readkey();
      }
   }
}
```

代码段 Ch05Ex03\Program.cs

(3) 执行代码，输入一个 1～4 之间的数字，以选择一个方向，输入一个距离值，结果如图 5-8 所示。

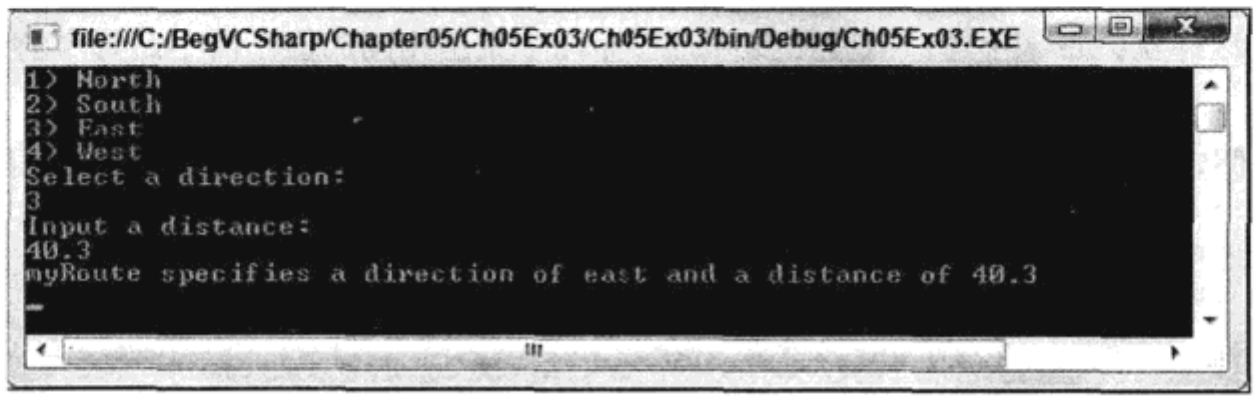


图 5-8

示例的说明

结构和枚举一样，也是在代码的主体之外声明的。在名称空间声明中声明 route 结构及其使用的 orientation 枚举：

```
enum orientation : byte
{
   north = 1,
   south = 2,
   east  = 3,
   west  = 4
}
struct route
{
   public orientation direction;
   public double      distance;
}
```

代码的主体结构与前面的一些示例类似，要求用户输入一些信息，并显示它们。把方向选项放在 do 循环中，对用户的输入进行有效性检查，拒绝不属于 1～4 范围的整数输入(选择了这样的值后，它们就会映射到枚举成员上，以方便赋值)。

不能解释为整数的输入会导致一个错误。本章后面会说明其原因和处理方法。

注意，在引用 route 的成员时，处理它们的方式与成员类型相同的变量完全一样。赋值语句如下所示：

```
myRoute.direction = (orientation)myDirection;
myRoute.distance = myDistance;
```

可以直接把输入的值放到 myRoute.distance 中，而不会有负面效果，如下所示：

```
myRoute.distance = Convert.ToDouble(Console.ReadLine());
```

还应进行有效性验证，但这段代码不存在这一步骤。对结构成员的任何访问都以相同的方式处理。<structVar>.<memberVar>形式的表达式可计算<memberVar>类型的变量。

5.2.3　数组

前面的所有类型都有一个共同点：它们都只存储一个值(结构中存储一组值)。有时，需要存储许多数据，这样就会带来不便。有时需要同时存储几个类型相同的值，而不是每个值使用不同的变量。

例如，假定要对所有朋友的姓名执行一些操作。可以使用简单的字符串变量，如下所示：

```
string friendName1 = "Robert Barwell";
string friendName2 = "Mike Parry";
string friendName3 = "Jeremy Beacock";
```

但这看起来需要很多工作，特别是需要编写不同的代码来处理每个变量。例如，不能在循环中迭代这个字符串列表。

另一种方式是使用数组。数组是一个变量的索引列表，存储在数组类型的变量中。例如，有一个数组 friendNames 存储上述的 3 个名字。在方括号中指定索引，即可访问该数组中的各个成员，如下所示：

```
friendNames[<index>]
```

这个索引是一个整数，第一个条目的索引是 0，第二个条目的索引是 1，依次类推。这样就可以使用循环遍历所有元素，例如：

```
int i;
for (i = 0; i < 3; i++)
{
  Console.WriteLine("Name with index of {0}: {1}", i, friendNames[i]);
}
```

数组有一个基本类型，数组中的各个条目都是这种类型。friendNames 数组的基本类型是字符串，因为它要存储 string 变量。数组的条目通常称为元素。

1. 声明数组

以下述方式声明数组：

```
<baseType>[] <name>;
```

其中，*<baseType>*可以是任何变量类型，包括本章前面介绍的枚举和结构类型。数组必须在访问之前初始化，不能像下面这样访问数组或给数组元素赋值：

```
int[] myIntArray;
myIntArray[10] = 5;
```

数组的初始化有两种方式。可以以字面形式指定数组的完整内容，也可以指定数组的大小，再使用关键字 new 初始化所有数组元素。

使用字面值指定数组，只需要提供一个用逗号分隔的元素值列表，该列表放在花括号中，例如：

```
int[] myIntArray = {5, 9, 10, 2, 99};
```

其中，myIntArray 有 5 个元素，每个元素都被赋予了一个整数值。

另一种方式需要使用下述语法：

```
int[] myIntArray = new int[5];
```

这里使用关键字 new 显式地初始化数组，用一个常量值定义其大小。这种方法会给所有的数组元素赋予同一个默认值，对于数值类型来说，其默认值是 0。也可以使用非常量的变量来进行初始化，例如：

```
int[] myIntArray = new int[arraySize];
```

还可以使用这两种初始化方式的组合：

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

使用这种方式，数组大小必须与元素个数相匹配。例如，不能编写如下代码：

```
int[] myIntArray = new int[10] {5, 9, 10, 2, 99};
```

其中数组定义为有 10 个元素，但只定义了 5 个元素，所以编译会失败。如果使用变量定义其大小，该变量必须是一个常量，例如：

```
const int arraySize = 5;
int[] myIntArray = new int[arraySize] {5, 9, 10, 2, 99};
```

如果省略了关键字 const，运行这段代码就会失败。

与其他变量类型一样，不见得在声明行中初始化数组。下面的代码是合法的：

```
int[] myIntArray;
myIntArray = new int[5];
```

下面的“试一试”示例利用了本节引言中的示例，创建并使用一个字符串数组。

试一试：使用数组

(1) 在 C:\BegVCSharp\Chapter05 目录中创建一个新控制台应用程序 Ch05Ex04。

(2) 把下列代码添加到 Program.cs 中：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   string[] friendNames = {"Robert Barwell", "Mike Parry",
                           "Jeremy Beacock"};
   int i;
   Console.WriteLine("Here are {0} of my friends:",
                     friendNames.Length);
   for (i = 0; i < friendNames.Length; i++)
   {
      Console.WriteLine(friendNames[i]);
   }
   Console.ReadKey();
}
```

代码段 Ch05Ex04\Program.cs

(3) 执行代码，结果如图 5-9 所示。

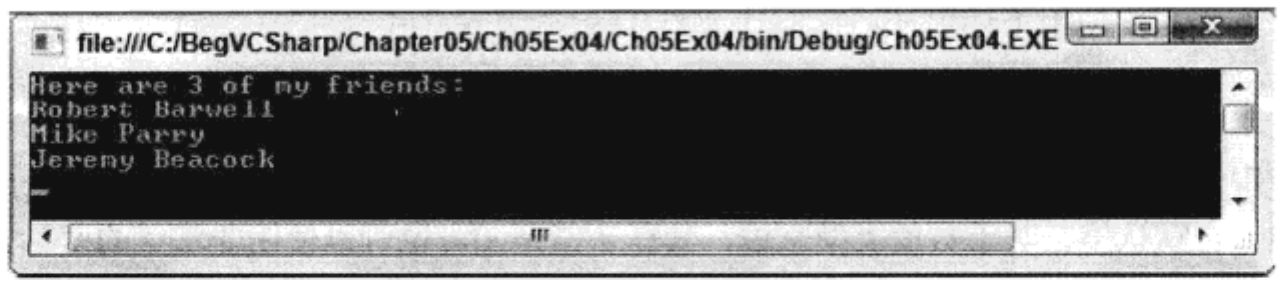


图　5-9

示例的说明

这段代码用 3 个值建立了一个 string 数组，并在 for 循环中把它们列在控制台上。使用 friendNames.Length 来确定数组中元素的个数：

```
Console.WriteLine("Here are {0} of my friends:", friendNames.Length);
```

这是获取数组大小的简便方法。在 for 循环中输出值容易出错。例如，把<改为<=，如下所示：

```
for (i = 0; i <= friendNames.Length; i++)
 {
    Console.WriteLine(friendNames[i]);
 }
```

编译代码，就会弹出如图 5-10 所示的对话框。

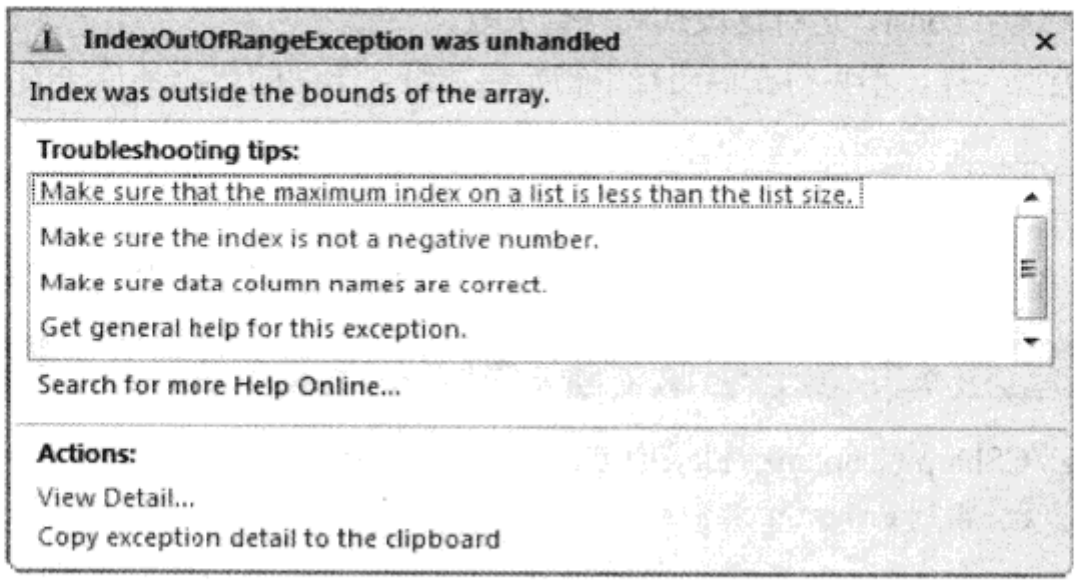


图　5-10

这里代码试图访问 friendNames[3]。记住，数组索引从 0 开始，所以最后一个元素是 friendNames[2]。如果试图访问超出数组大小的元素，代码就会出问题。还可以通过一个更具弹性的方法来访问数组的所有成员，即使用 foreach 循环。

2. foreach 循环

foreach 循环可以使用一种简便的语法来定位数组中的每个元素：

```
foreach (<baseType> <name> in <array>)
{
   // can use <name> for each element
}
```

这个循环会迭代每个元素，依次把每个元素放在变量<name>中，且不存在访问非法元素的危险。

不需要考虑数组中有多少个元素，并可以确保将在循环中使用每个元素。使用这个循环，可以修改上一个示例中的代码，如下所示：

```
static void Main(string[] args)
{
   string[] friendNames = {"Robert Barwell", "Mike Parry",
                           "Jeremy Beacock"};
  Console.WriteLine("Here are {0} of my friends:",
                    friendNames.Length);
   foreach (string friendName in friendNames)
   {
      Console.WriteLine(friendName);
   }
   Console.ReadKey();
}
```

这段代码的输出结果与前面的示例完全相同。使用这种方法和标准的 for 循环的主要区别在于：foreach 循环对数组内容进行只读访问，所以不能改变任何元素的值。例如，不能编写如下代码：

```
foreach (string friendName in friendNames)
{
   friendName = "Rupert the bear";
}
```

如果编译这段代码，就会失败。但如果使用简单的 for 循环，就可以给数组元素赋值。

3. 多维数组

多维数组是使用多个索引访问其元素的数组。例如，假定要确定一座山相对于某位置的高度，可以使用两个坐标 x 和 y 来指定一个位置。把这两个坐标用作索引，数组 hillHeight 就可以用每对坐标来存储高度，这就要使用多维数组了。

像这样的二维数组可以声明如下：

```
<baseType>[,] <name>;
```

多维数组只需要更多的逗号，例如：

```
<baseType>[,,,] <name>;
```

该语句声明了一个 4 维数组。赋值也使用类似的语法，用逗号分隔大小。要声明和初始化二维数组 hillHeight，其基本类型是 double，x 的大小是 3，y 的大小是 4，则需要：

```
double[,] hillHeight = new double[3,4];
```

还可以使用字面值进行初始的赋值。这里使用嵌套的花括号块，用逗号分隔开，例如：

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
```

这个数组的维度与前面的相同，也是 3 行 4 列。通过提供字面值隐式定义了这些维度。

要访问多维数组中的每个元素，只需指定它们的索引，并用逗号分隔开，例如：

```
hillHeight[2,1]
```

接着就可以像其他元素那样处理它了。这个表达式将访问上面定义的第 3 个嵌套数组中的第 2 个元素(其值是 4)。记住，索引从 0 开始，第一个数字是嵌套的数组。换言之，第一个数字指定花括号对，第 2 个数字指定该对花括号中的元素。用图 5-11 来表示这个数组。

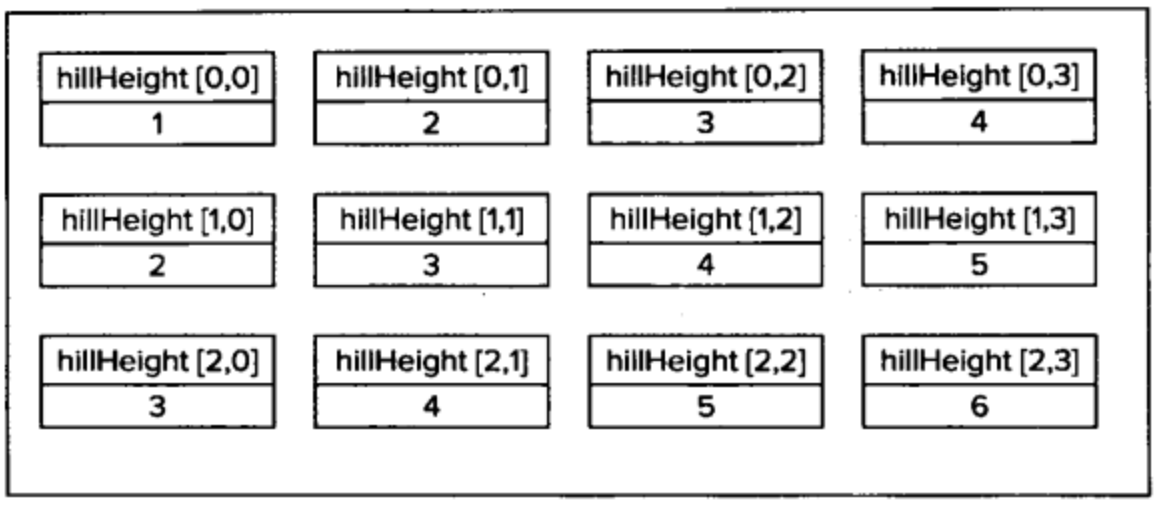


图　5-11

foreach 循环可以访问多维数组中的所有元素，其方式与访问一维数组相同，例如：

```
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
foreach (double height in hillHeight)
{
   Console.WriteLine("{0}", height);
}
```

元素的输出顺序与赋予字面值的顺序相同(这里显示了元素的标识符，而不是实际值)：

```
hillHeight[0,0]
hillHeight[0,1]
hillHeight[0,2]
hillHeight[0,3]
hillHeight[1,0]
hillHeight[1,1]
hillHeight[1,2]
```

4. 数组的数组

上一节讨论的多维数组可称为矩形数组，这是因为每一行的元素个数都相同。使用上一个示例，任何一个 x 坐标都可以对应 0~3 的 y 坐标。

也可以使用锯齿数组(jagged array)，其中每行都有不同的元素个数。为此，需要有这样一个数组，其中的每个元素都是另一个数组。也可以有数组的数组的数组，甚至更复杂的数组。但是，注意这些数组都必须有相同的基本类型。

声明数组的数组，其语法要在数组的声明中指定多个方括号对，例如：

```
int[][] jaggedIntArray;
```

但初始化这样的数组不像初始化多维数组那样简单，例如不能采用以下的声明方式：

```
jaggedIntArray = new int[3][4];
```

即使这样做了，也不是很有效，因为使用简单的多维数组可以较为轻松地获得相同的结果。也

不能使用下面的代码：

```
jaggedIntArray = {{1, 2, 3}, {1}, {1, 2}};
```

有两种方式：可以初始化包含其他数组的数组(为了清晰起见，称之为子数组)，然后依次初始化子数组：

```
jaggedIntArray = new int[2][];
jaggedIntArray[0] = new int[3];
jaggedIntArray[1] = new int[4];
```

也可以使用上述字面值赋值的一种改进形式：

```
jaggedIntArray = new int[3][] { new int[] { 1, 2, 3 }, new int[] { 1 },
                                new int[] { 1, 2 } };
```

也可以进行简化，把数组的初始化和声明放在同一行上，如下所示：

```
int[][] jaggedIntArray = { new int[] { 1, 2, 3 }, new int[] { 1 },
                           new int[] { 1, 2 } };
```

对锯齿数组可以使用 foreach 循环，但通常需要使用嵌套方法，才能得到实际数据。例如，假定下述锯齿数组包含 10 个数组，每个数组又包含一个整数数组，其元素是 1~10 的约数：

```
int[][] divisors1To10 = {new int[] {1},
                         new int[] {1, 2},
                         new int[] {1, 3},
                         new int[] {1, 2, 4},
                         new int[] {1, 5},
                         new int[] {1, 2, 3, 6},
                         new int[] {1, 7},
                         new int[] {1, 2, 4, 8},
                         new int[] {1, 3, 9},
                         new int[] {1, 2, 5, 10}};
```

下面的代码会失败：

```
foreach (int divisor in divisors1To10)
{
   Console.WriteLine(divisor);
}
```

这是因为数组 divisors1To10 包含 int[ ]元素，而不是 int 元素。必须循环每个子数组和数组本身：

```
foreach (int[] divisorsOfInt in divisors1To10)
{
   foreach(int divisor in divisorsOfInt)
   {
      Console.WriteLine(divisor);
   }
}
```

可以看出，使用锯齿数组的语法要复杂得多！在大多数情况下，使用矩形数组比较简单，这是一种比较简单的存储方式。但是，有时必须使用锯齿数组，且工作效率并不会因此而降低。

5.3　字符串的处理

到目前为止，对字符串的使用还仅限于把字符串写到控制台上，从控制台上读取字符串，以及使用+运算符连接字符串。在编写较有趣的应用程序时，会发现字符串的操作非常多。所以，下面用几页的篇幅介绍 C#中比较常用的字符串处理技巧。

首先要注意，string 类型变量可以看作是 char 变量的只读数组。这样，就可以使用下面的语法访问每个字符：

```
string myString = "A string";
char myChar = myString[1];
```

但是，不能用这种方式为各个字符赋值。为了获得一个可写的 char 数组，可以使用下面的代码，其中使用了数组变量的 ToCharArray()命令：

```
string myString = "A string";
char[] myChars = myString.ToCharArray();
```

接着就可以采用标准方式处理 char 数组了。也可以在 foreach 循环中使用字符串，例如：

```
foreach (char character in myString)
{
   Console.WriteLine("{0}", character);
}
```

与数组一样，还可以使用 myString.Length 获取元素的个数，这将给出字符串中的字符数，例如：

```
string myString = Console.ReadLine();
Console.WriteLine("You typed {0} characters.", myString.Length);
```

其他的基本字符串处理技巧采用与这个<string>.ToCharArray()命令类似的格式使用命令。两个简单但很有效的命令是<string>.ToLower()和<string>.ToUpper()。它们可以分别把字符串转换为大写或小写形式。要明白为什么它们非常有用，可以考虑下面的情形：要检查用户的某个响应，例如字符串 yes。如果可以把用户输入的字符串转换为小写形式，就也能检查字符串 YES、Yes、yeS 等，第 4 章介绍了这样一个示例：

```
string userResponse = Console.ReadLine();
if (userResponse.ToLower() == "yes")
{
   // Act on response.
}
```

注意，这个命令与本节的其他命令一样，并没有真正改变应用它的字符串。把这个命令与字符串合并使用，就会创建一个新的字符串，以便与另一个字符串进行比较(如上所述)，或者赋给另一个变量。该变量可能与当前操作的变量相同，例如：

```
userResponse = userResponse.ToLower();
```

这是一个要点，因为只写出下面的代码是没用的：

```
userResponse.ToLower();
```

下面看看在简化用户输入方面还可以做什么。如果用户无意间在输入内容的前面或后面添加了额外的空格，会怎样？此时，上述代码就不起作用了。这就需要删除输入字符串中的空格，此时可以使用<string>.Trim()命令来处理。

```
string userResponse = Console.ReadLine();
userResponse = userResponse.Trim();
if (userResponse.ToLower() == "yes")
{
   // Act on response.
}
```

使用该命令，还可以检测如下的字符串：

```
"  YES"
"Yes  "
```

也可以使用这些命令删除其他字符，只要在一个 char 数组中指定这些字符即可，例如：

```
char[] trimChars = {' ', 'e', 's'};
string userResponse = Console.ReadLine();
userResponse = userResponse.ToLower();
userResponse = userResponse.Trim(trimChars);
if (userResponse == "y")
{
   // Act on response.
}
```

这将从字符串的前面或后面删除所有空格、字母 e 和 s。如果字符串中没有其他字符，就会检测以下字符串：

```
"Yeeeees"
"  y"
```

还可以使用<string>.TrimStart()和<string>.TrimEnd()命令。它们可以把字符串的前面或后面的空格删掉。这些命令也需要指定 char 数组。

还有另外两个字符串命令可以处理字符串的空格：<string>.PadLeft()和<string>.PadRight()。它们可以在字符串的左边或右边添加空格，使字符串达到指定的长度。其语法如下：

```
<string>.PadX(<desiredLength>);
```

例如：

```
myString = "Aligned";
myString = myString.PadLeft(10);
```

这将在 myString 中把 3 个空格添加到单词 Aligned 的左边。这些方法可以用于在列中对齐字符串，特别适用于放置包含数字的字符串。

与修整命令一样，还可以按照第二种方式使用这些命令，即提供要添加到字符串上的字符，这需要一个 char，而不是像修整命令那样指定一个 char 数组。例如：

```
myString = "Aligned";
myString = myString.PadLeft(10, '-');
```

这将在 myString 的开头加上 3 个短横线。

还有许多这样的字符串处理命令，其中一些只用于非常特殊的情况，在后面的章节中遇到它们时进行讨论。在继续下面内容之前，介绍 Visual C# 2010 Express Edition 和 Visual Studio 2010 中的一个前几章涉及到(特别是本章)的特性。下面的示例会试验语句自动完成功能，IDE 会给出用户可能要插入的代码。

试一试：VS 中的语句自动完成功能

(1) 在 C:\BegVCSharp\Chapter05 目录中创建一个新控制台应用程序 Ch05Ex05。

(2) 在 Program.cs 中输入下列代码，注意输入过程中弹出的窗口：


```
static void Main(string[] args)
{
    string myString = "This is a test.";
    char[] separator = {' '};
    string[] myWords;
    myWords = myString.
}
```

代码段 Ch05Ex05\Program.cs

(3) 输入最后的句点时，注意会弹出如图 5-12 所示的窗口。

图　5-12

(4) 不要移动光标，键入 sp，弹出窗口就会改变，显示一个黄色的工具提示窗口，如图 5-13 所示。

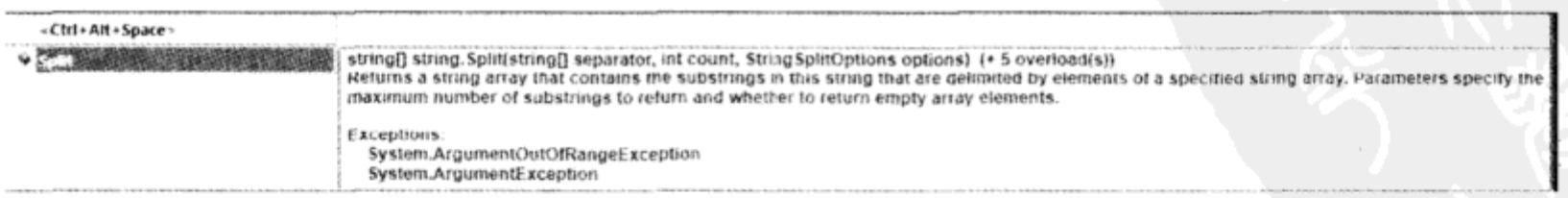


图　5-13

(5) 输入字符“(se”，会弹出另一个窗口，如图 5-14 所示。

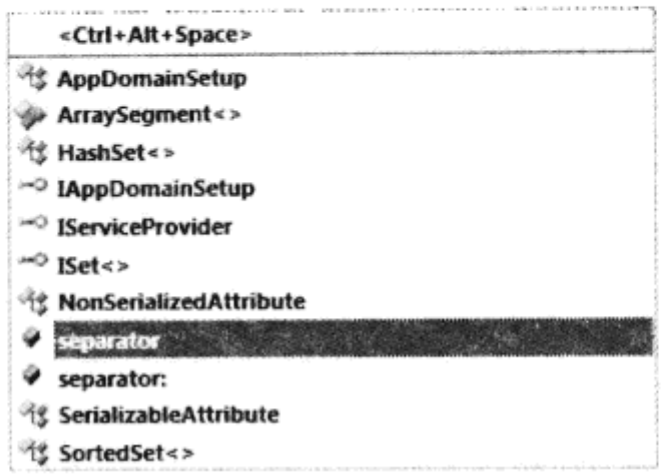


图　5-14

(6) 输入两个字符“);”，代码如下所示，弹出窗口随之消失。

```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.Split(separator);
}
```

(7) 添加下述代码，注意弹出的窗口。

```
static void Main(string[] args)
{
   string myString = "This is a test.";
   char[] separator = {' '};
   string[] myWords;
   myWords = myString.Split(separator);
   foreach (string word in myWords)
   {
      Console.WriteLine("{0}", word);
   }
   Console.ReadKey();
}
```

(8) 执行代码，结果如图 5-15 所示。

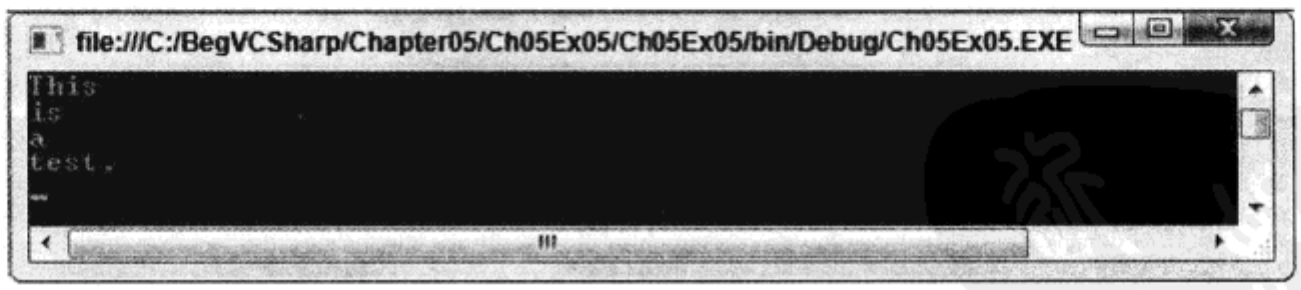


图　5-15

示例的说明

在这段代码中，要注意两点。第一点是所使用的新字符串命令，第二点是使用了自动完成功能。使用命令<string>.Split()把 string 转换为 string 数组，把它在指定的位置分隔开。这些位置采用 char 数组的形式，在本例中该数组只有一个元素，即空格字符：

```
char[] separator = {' '};
```

下面的代码把字符串在每个空格处分解开时，会得到其子字符串，即得到包含单个单词的数组：

```
string[] myWords;
myWords = myString.Split(separator);
```

接着使用 foreach 循环迭代这个数组中的单词，并把这些单词写到控制台上：

```
foreach (string word in myWords)
{
  Console.WriteLine("{0}", word);
}
```

得到的每个单词都没有空格，单词的内部和两端都没有空格。在使用 split()时，删除了分隔符。

接着，看看自动完成功能。VS 和 VCE 都是智能化极高的程序包，在用户键入代码时会提供许多关于代码的信息。即使在一个新行上键入第一个字符，IDE 也试图帮助用户，建议输入关键字、变量名、类型名等。只要在上面的代码中输入 3 个字母 str，IDE 就会猜出要输入 string。在键入变量名时，这个功能会更有用。在较长的代码中，常常会忘记要使用的变量名。IDE 会在用户键入代码的过程中弹出一系列变量名，用户可以从中选择，无需查看以前的代码。

在 myString 的后面键入句点时，IDE 知道 myString 是一个字符串，也知道您要指定一个字符串命令，于是显示可用选项。此时可以停止键入，使用上下箭头键选择需要的命令。在这些可用命令中移动时，IDE 会告诉您当前选中命令的含义，以及使用它的语法。

在开始键入更多字符时，IDE 会把选中的命令移动到命令的顶部，以便自动键入该命令。一旦它显示出需要的命令，就可以继续键入，就像键入完整的名称一样，所以键入"("会直接跳到指定的位置上，在该位置上，有某些命令需要的额外信息——IDE 甚至会告诉用户该信息必须采用的格式，并显示这些接受各种信息的命令选项。

IDE 的这个功能(也称为 IntelliSense)使用起来非常方便，可以使您轻松地找到奇怪类型的信息。查看 string 类型的所有命令是很有趣的，这不会使计算机崩溃，试一试吧！

有时所显示的信息会遮挡前面已经键入的代码，这是很恼人的。因为在键入时需要引用被遮挡的代码。此时可以按下 Ctrl 键，使命令列表变成透明的，以便查看被遮挡的代码。

5.4 小结

本章用一定的篇幅扩展了变量的知识。本章最重要的话题是类型转换，后面还要讨论它。掌握本章介绍的概念，会使以后的学习容易得多。

另外还介绍了几个变量类型，它们可以采用更友好的方式存储数据。本章阐述了枚举如何使数值更容易辨识，从而使代码的可读性更高，结构如何在一个地方合并多个相关的数据元素，如何把类似的数据组合到数组中。本书其他地方常用到这些类型。

最后介绍了字符串的处理，讨论了一些基本技巧和规则。C#提供了许多字符串命令，但这里只介绍了其中几个，还说明了如何查看 IDE 中可用的命令。使用这些技巧可以尝试许多工作。至少下面的练习可以使用本章没有讨论的一个或多个字符串命令来完成。

本章扩展了变量的知识，包括：

- 类型转换
- 枚举
- 结构
- 数组
- 字符串处理

5.5　练习

(1) 下面的转换哪些不是隐式转换？

a. int 转换为 short

b. short 转换为 int

c. bool 转换为 string

d. byte 转换为 float

(2) 基于 short 类型的 color 枚举包含彩虹的颜色，再加上黑色和白色，据此编写 color 枚举的代码。这个枚举可以使用 byte 类型吗？

(3) 修改第 4 章的 Mandelbrot 集合生成程序示例，使用下面的结构表示复数：

```
struct imagNum
{
   public double real, imag;
}
```

(4) 下面的代码可以成功编译吗？为什么？

```
string[] blab = new string[5]
string[5] = 5th string.
```

(5) 编写一个控制台应用程序，它接收用户输入的一个字符串，将其中的字符以与输入相反的顺序输出。

(6) 编写一个控制台应用程序，它接收一个字符串，用 yes 替换字符串中所有的 no。

(7) 编写一个控制台应用程序，给字符串中的每个单词加上双引号。

附录 A 给出了练习答案。

5.6　本章要点

主　　题	重 要 概 念
类型转换	值可以从一种类型转换为另一种类型，但在转换时应遵循一些规则。隐式转换是自动进行的，但只有源值类型的所有可能值都可以在目标值类型中使用时，才能进行隐式转换。也可以进行显式转换，但可能得不到期望的值，甚至可能出错
枚举	枚举是包含一组离散值的类型，每个离散值都有一个名称。枚举用 enum 关键字定义，以便在代码中理解它们，因为它们的可读性都很高。枚举有基本的数值类型(默认是 int)，可以使用枚举值的这个属性在枚举值和数值之间转换，或者标识枚举值
结构	结构是同时包含几个不同的值的类型。结构用 struct 关键字定义。包含在结构中的每个值都有名称和类型，存储在结构中的每个值的类型不一定相同
数组	数组是同类型数值的集合。数组有固定的大小或长度，确定了数组可以包含多少个值。可以定义多维数组或锯齿数组，来保存不同数量和形状的数据。还可以使用 foreach 循环迭代数组中的值

第 6 章

函　数

本章内容：

- 如何定义和使用不接受或返回任何数据的简单函数
- 如何在函数中传入传出数据
- 使用变量作用域
- 如何结合使用 Main()函数和命令行参数
- 如何把函数提供为结构类型的成员
- 如何使用函数重载
- 如何使用委托

我们迄今看到的代码都是以单个代码块的形式出现的，其中包含一些重复执行的循环代码，以及有条件地执行的分支语句。如果要对数据执行某种操作，就应把所需要的代码放在合适的地方。

这种代码结构的作用是有限的。某些任务常常需要在一个程序中执行好几次，例如，查找数组中的最大值。此时可以把相同(或几乎相同)的代码块按照需要放在应用程序中，但这样做也会存在问题。在某个常见任务中，即使进行非常小的改动(例如，修改某个代码错误)，也需要修改多个代码块，这些代码块可能分布在整个应用程序中。如果忘了修改其中的一个代码块，就会产生很大的影响，导致整个应用程序失败。另外，应用程序也较长。

解决这个问题的方法是使用函数。在 C#中，函数是一种方法，可提供在应用程序中的任何一处执行的代码块。

本章介绍的特定类型的函数称为“方法”。但是，这个术语在.NET 编程中有非常特殊的含义，本书后面会详细讨论它，所以现在不使用这个术语。

例如，有一个函数返回数组中的最大值，可以在代码的任何位置使用这个函数，且在每个地方都使用相同的代码行。因为只需要提供一次这段代码，所以对代码的任何修改将影响使用该函数进

行的计算。这个函数可以看作包含可重用的代码。

函数还可以提高代码的可读性，因为可以使用函数将相关代码组合在一起。这样，应用程序主体就会非常短，因为代码的内部工作被分散了。这类似于在 IDE 中使用大纲视图将代码区域折叠在一起，应用程序的结构更加合理。

函数还可以用于创建多用途的代码，让它们对不同的数据执行相同的操作。可以采用参数形式为函数提供信息，以返回值的形式得到函数的结果。在上面的示例中，参数就是一个要搜索的数组，而返回值就是数组中的最大值。这意味着每次可以使用同一个函数处理不同的数组。函数的名称和参数(不是返回类型)共同定义了函数的签名。

6.1　定义和使用函数

本节介绍如何将函数添加到应用程序中，以及如何在代码中使用(调用)它们。首先从基础知识开始，看看不与调用代码交换任何数据的简单函数，然后介绍更高级的函数用法。首先看一个示例。

试一试：定义和使用基本函数

(1) 在 C:\BegVCSharp\Chapter06 目录中创建一个新控制台应用程序 Ch06Ex01。

(2) 把下述代码添加到 Program.cs 中：


```
class Program
{
   static void Write()
   {
      Console.WriteLine("Text output from function.");
   }

   static void Main(string[] args)
   {
      Write();
      Console.ReadKey();
   }
}
```

代码段 Ch06Ex01\Program.cs

(3) 执行代码，结果如图 6-1 所示。

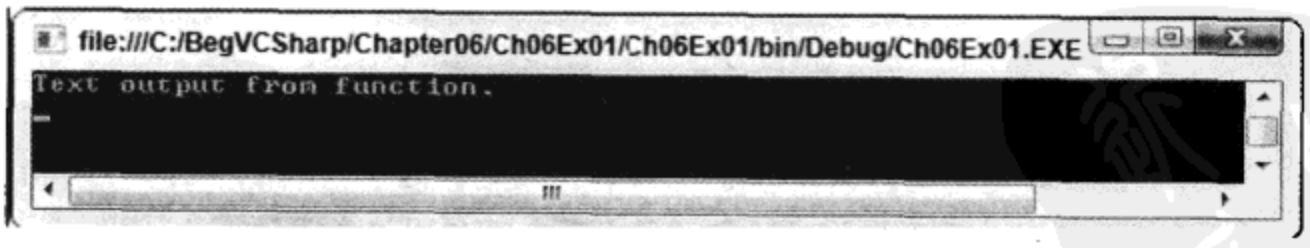


图　6-1

示例的说明

下面的 4 行代码定义了函数 Write()：

```
static void Write()
```

```
{
    Console.WriteLine("Text output from function.");
}
```

这些代码把一些文本输出到控制台窗口中。但此时这些并不重要，我们更关心定义和使用函数的机制。

函数定义由以下几部分组成：

- 两个关键字：static 和 void
- 函数名后跟圆括号，如 Write()
- 一个要执行的代码块，放在花括号中

一般采用 PascalCase 形式编写函数名。

定义 Write()函数的代码非常类似于应用程序中的其他代码：

```
static void Main(string[] args)
{
   ...
}
```

这是因为，到目前为止我们编写的所有代码(除了类型定义之外)都是函数的一部分。函数 Main()是控制台应用程序的入口点函数。当执行 C#应用程序时，就会调用它包含的入口点函数，这个函数执行完毕后，应用程序就终止了。所有 C#可执行代码都必须有一个入口点。

Main()函数和 Write()函数的唯一区别(除了它们包含的代码)是函数名 Main 后面的圆括号中还有一些代码，这是指定参数的方式，详见后面的内容。

如上所述，Main()函数和 Write()函数都是使用关键字 static 和 void 定义的。关键字 static 与面向对象的概念相关，本书在后面讨论。现在只需记住，在本节的应用程序中所使用的所有函数都必须使用这个关键字。

而 void 更容易解释。这个关键字表明函数没有返回值。本章后面将讨论函数有返回值时需要编写什么代码。

继续下去，调用函数的代码如下所示：

```
Write();
```

键入函数名，后跟空括号即可。在程序执行到这行代码时，就会运行 Write()函数中的代码。

在定义函数和调用函数时，必须使用圆括号。如果删除它们，将无法编译代码。

6.1.1 返回值

通过函数进行数据交换的最简单方式是利用返回值。有返回值的函数会计算这个值，其方式与在表达式中使用变量计算它们包含的值完全相同。与变量一样，返回值也有数据类型。

例如，有一个函数 GetString()，其返回值是一个字符串，可以在代码中使用该函数，如下所示：

```
string myString;
myString = GtString();
```

还有一个函数 GetVal()，它返回一个 double 值，可以在数学表达式中使用它。

```
double myVal;
double multipler = 5.3;
myVal = GetVal() * multiplier;
```

当函数返回一个值时，可以采用以下两种方式修改函数：

- 在函数声明中指定返回值的类型，但不使用关键字 void。
- 使用 return 关键字结束函数的执行，把返回值传送给调用代码。

从代码角度分析，控制台应用程序函数中的下述代码看起来像是前面见过的函数类型：

```
static <returnType> <functionName>()
{
   ...
   return <returnValue>;
}
```

这里唯一的限制是<returnValue>必须是一个值，其类型可以是<returnType>，也可以隐式转换为该类型。但是，<returnType>可以是任何类型，包括前面介绍的较复杂的类型。这段代码可以很简单：

```
static double GetVal()
{
   return 3.2;
}
```

但是，返回值通常是函数执行的一些处理的结果，只需使用 const 变量即可得到以上结果。

在执行到 return 语句时，程序会立即返回调用代码。这个语句后面的代码都不会执行。但是，这并不意味着 return 语句只能放在函数体的最后一行。可以在前边的代码里使用 return，也可能在执行了分支逻辑之后使用。把 return 语句放在 for 循环、if 块或其他结构中会使该结构立即终止，函数也立即终止。例如：

```
static double GetVal()
{
   double checkVal;
   // checkVal assigned a value through some logic(not shown here).
   if (checkVal < 5)
      return 4.7;
      return 3.2;
}
```

根据 checkVal 的值，将返回两个值中的一个。这里唯一的限制是 return 语句必须在函数的闭合花括号 } 之前处理。下面的代码是不合法的：

```
static double GetVal()
{
   double checkVal;
   // checkVal assigned a value through some logic.
```

```
    if (checkVal < 5)
      return 4.7;
}
```

如果 checkVal>= 5，就不会执行到 return 语句，这是不允许的。所有的处理路径都必须执行到 return 语句。在大多数情况下，编译器会检查是否执行到 return 语句，如果没有，就给出一个错误“并不是所有的处理路径都返回一个值”。

最后需要注意的是，return 可以用在通过 void 关键字声明的函数中(没有返回值)。如果这么做，函数就会立即终止。以这种方式使用 return 语句时，在 return 关键字和其后的分号之间提供返回值是错误的。

6.1.2 参数

当函数接受参数时，就必须指定下述内容：

- 函数在其定义中指定接受的参数列表，以及这些参数的类型。
- 在每个函数调用中匹配的参数列表。

这涉及到下述代码：

```
static <returnType> <functionName>(<paramType> <paramName>, ...)
{
   ...
   return <returnValue>;
}
```

其中可以有任意多个参数，每个参数都有一个类型和一个名称。参数用逗号分隔开。每个参数都在函数的代码中用作一个变量。例如，下面是一个简单的函数，带有两个 double 参数，并返回它们的乘积：

```
static double Product(double param1, double param2)
{
   return param1 * param2;
}
```

下面看一个较复杂的示例。

试一试：通过函数交换数据(1)

(1) 在 C:\BegVCSharp\Chapter06 目录中创建一个新控制台应用程序 Ch06Ex02。

(2) 把下列代码添加到 Program.cs 中：


```
class Program
{
   static int MaxValue(int[] intArray)
   {
      int maxVal = intArray[0];
      for (int i = 1; i < intArray.Length; i++)
      {
         if (intArray[i] > maxVal)
            maxVal = intArray[i];
      }
      return maxVal;
```

```
    }

    static void Main(string[] args)
    {
      int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
      int maxVal = MaxValue(myArray);
      Console.WriteLine("The maximum value in myArray is {0}", maxVal);
      Console.ReadKey();
    }
}
```

代码段 Ch06Ex02\Program.cs

(3) 执行代码，结果如图 6-2 所示。

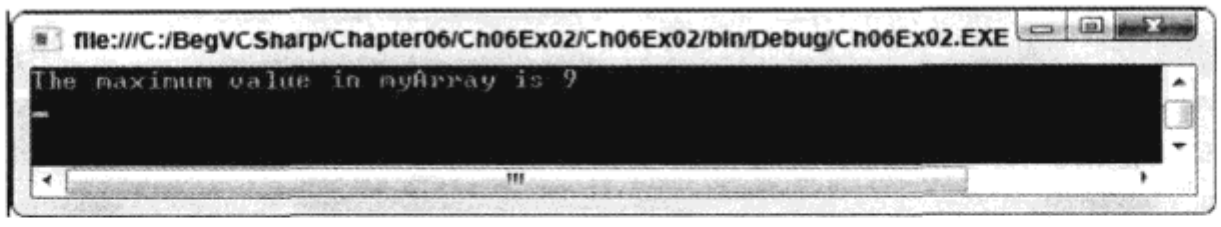


图 6-2

示例的说明

这段代码包含一个函数，它执行的任务就是本章引言中示例函数所完成的任务。该函数的参数是一个整数数组，返回该数组中的最大值。该函数的定义如下所示：

```
static int MaxValue(int[] intArray)
{
  int maxVal = intArray[0];
  for (int i = 1; i < intArray.Length; i++)
  {
    if (intArray[i] > maxVal)
      maxVal = intArray[i];
  }
  return maxVal;
}
```

函数 MaxValue()定义了一个参数，即 int 数组 intArray，它还有一个 int 类型的返回值。最大值的计算是很简单的。局部整型变量 maxVal 初始化为数组中的第一个值，然后把这个值与数组中后面的每个元素依次进行比较。如果一个元素的值比 maxVal 大，就用这个值代替当前的 maxVal 值。循环结束时，maxVal 就包含数组中的最大值，用 return 语句返回。

Main()中的代码声明并初始化一个简单的整数数组，用于 MaxValue()函数：

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
```

调用 MaxValue()，把一个值赋给 int 变量 maxVal：

```
int maxVal = MaxValue(myArray);
```

接着，使用 Console.WriteLine()把这个值写到屏幕上：

```
Console.WriteLine("The maximum value in myArray is {0}", maxVal);
```

1. 参数匹配

在调用函数时，必须使参数与函数定义中指定的参数完全匹配，这意味着要匹配参数的类型、个数和顺序。例如，下面的函数：

```
static void MyFunction(string myString, double myDouble)
{
   ...
}
```

不能使用下面的代码调用：

```
MyFunction (2.6, "Hello");
```

这里试图把一个 double 值作为第一个参数传递，把 string 值作为第二个参数传递，参数的顺序与函数声明中定义的顺序不匹配。

也不能使用下面的代码：

```
MyFunction("Hello");
```

这里仅传送了一个 string 参数，而该函数需要两个参数。使用上述两个函数调用都会产生编译错误，因为编译器要求必须匹配所用函数的签名。

使用函数的名称和参数定义函数的签名。

再回顾这个示例，MaxValue()只能用于获取整数数组中的最大 int 值。如果用下面的代码替换 Main()中的代码：

```
static void Main(string[] args)
{
    double[] myArray = {1.3, 8.9, 3.3, 6.5, 2.7, 5.3};
    double maxVal = MaxValue(myArray);
    Console.WriteLine("The maximum value in myArray is {0}", maxVal);
    Console.ReadKey();
}
```

就不能编译这段代码，因为参数类型是错误的。在本章后面的“重载函数”一节将介绍解决这个问题的一个有效技术。

2. 参数数组

C#允许为函数指定一个(只能指定一个)特定的参数，这个参数必须是函数定义中的最后一个参数，称为参数数组。参数数组可以使用个数不定的参数调用函数，可以使用 params 关键字定义它们。

参数数组可以简化代码，因为不必从调用代码中传递数组，而是传递同类型的几个参数，这些参数放在可在函数中使用的一个数组中。

定义使用参数数组的函数时，需要使用下列代码：

```
static <returnType> <functionName>(<p1Type> <p1Name>, ... ,
```

```
                                        params <type>[] <name>)
{
   ...
   return <returnValue>;
}
```

使用下面的代码可以调用该函数。

```
<functionName>(<p1>, ... , <val1>, <val2>, ...)
```

其中<val1>和<val2>等都是<type>类型的值，用于初始化<name>数组。可以指定的参数个数几乎不受限制。唯一的限制是它们都必须是<type>类型。甚至可以根本不指定参数。

这一点使参数数组特别适合于为在处理过程中要使用的函数指定其他信息。例如，假定有一个函数 GetWord()，它的第一个参数是一个 string 值，并返回字符串中的第一个单词。

```
string firstWord = GetWord("This is a sentence.");
```

其中 firstWord 被赋予字符串 This。

可在 GetWord()中添加一个 params 参数，以根据其索引选择另一个要返回的单词：

```
string firstWord = GetWord("This is a sentence.", 2);
```

假定第一个单词计数为 1，则 firstWord 就被赋予字符串 is。

也可以在第 3 个参数中限制返回的字符个数，同样通过 params 参数来实现：

```
string firstWord = GetWord("This is a sentence.", 4, 3);
```

此时 firstWord 被赋予字符串 sen。

下面的示例定义并使用带有 params 类型参数的函数。

试一试：通过函数交换数据(2)

(1) 在 C:\BegVCSharp\Chapter06 目录中创建一个控制台应用程序 Ch06Ex03。

(2) 把下述代码添加到 Program.cs 中：

可从 wrox.com 下载源代码

```
class Program
{
   static int SumVals(params int[] vals)
   {
     int sum = 0;
      foreach (int val in vals)
      {
         sum += val;
      }
      return sum;
   }

   static void Main(string[] args)
   {
      int sum = SumVals(1, 5, 2, 9, 8);
      Console.WriteLine("Summed Values = {0}", sum);
```

```
            Console.Readkey();
        }
    }
```

代码段 Ch06Ex03\Program.cs

(3) 执行代码，结果如图 6-3 所示。

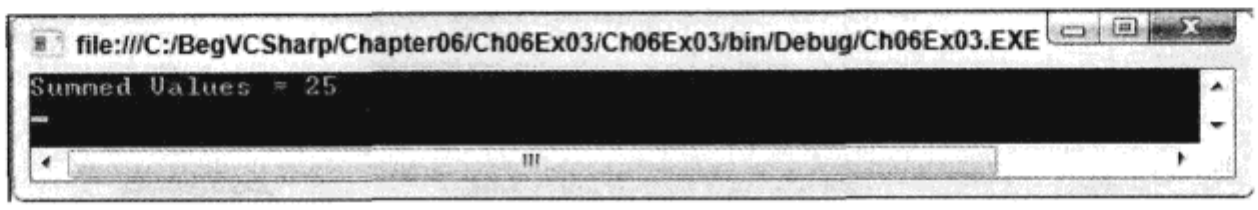


图 6-3

示例的说明

这个示例用关键字 params 定义函数 sumVals()，该函数可以接受任意个 int 参数(但不接受其他类型的参数)：

```
static int SumVals(params int[] vals)
{
   ...
}
```

这个函数对 vals 数组中的值进行迭代，把这些值加在一起，返回其结果。

在 Main()中，用 5 个整型参数调用函数 SumVals()：

```
int sum = SumVals(1, 5, 2, 9, 8);
```

也可以用 0、1、2 或 100 个整型参数调用这个函数——参数的数量不受限制。

3. 引用参数和值参数

本章迄今定义的所有函数都带有值参数。其含义是，在使用参数时，是把一个值传递给函数使用的一个变量。对函数中此变量的任何修改都不影响函数调用中指定的参数。例如，下面的函数使传递过来的参数值加倍，并显示出来：

```
static void ShowDouble(int val)
{
   val *= 2;
   Console.WriteLine("val doubled = {0}", val);
}
```

参数 val 在这个函数中被加倍，如果按以下方式调用它：

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

输出到控制台上的文本如下所示：

```
myNumber = 5
```

```
val doubled = 10
myNumber = 5
```

把 myNumber 作为一个参数，调用 ShowDouble()并不影响 Main()中 myNumber 的值，即使赋予 val 的参数被加倍，myNumber 的值也不变。

这很不错，但如果要改变 myNumber 的值，就会有问题。可以使用一个为 myNumber 返回新值的函数：

```
static int DoubleNum(int val)
{
   val *= 2;
   return val;
}
```

并使用下面的代码调用它：

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
myNumber = DoubleNum(myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

但这段代码一点也不直观，且不能改变用作参数的多个变量值(因为函数只有一个返回值)。

此时可以通过“引用”传递参数。即函数处理的变量与函数调用中使用的变量相同，而不仅仅是值相同的变量。因此，对这个变量进行的任何改变都会影响用作参数的变量值。为此，只需使用 ref 关键字指定参数：

```
static void ShowDouble(ref int val)
{
   val *= 2;
   Console.WriteLine("val doubled = {0}", val);
}
```

在函数调用中(这是必须的，因为 ref 参数是函数签名的一部分)再次指定它：

```
int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

输出到控制台的文本如下所示：

```
myNumber = 5
val doubled = 10
myNumber = 10
```

这次，myNumber 被 ShowDouble()修改了。

用作 ref 参数的变量有两个限制。首先，函数可能会改变引用参数的值，所以必须在函数调用中使用“非常量”变量。所以，下面的代码是非法的：

```
const int myNumber = 5;
Console.WriteLine("myNumber = {0}", myNumber);
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

其次，必须使用初始化过的变量。C#不允许假定 ref 参数在使用它的函数中初始化，下面的代码也是非法的：

```
int myNumber;
ShowDouble(ref myNumber);
Console.WriteLine("myNumber = {0}", myNumber);
```

4. 输出参数

除了按引用传递值之外，还可以使用 out 关键字，指定所给的参数是一个输出参数。out 关键字的使用方式与 ref 关键字相同(在函数定义和函数调用中用作参数的修饰符)。实际上，它的执行方式与引用参数完全一样，因为在函数执行完毕后，该参数的值将返回给函数调用中使用的变量。但是，存在一些重要区别。

- 把未赋值的变量用作 ref 参数是非法的，但可以把未赋值的变量用作 out 参数。
- 另外，在函数使用 out 参数时，out 参数必须看作是还未赋值。

即调用代码可以把已赋值的变量用作 out 参数，存储在该变量中的值会在函数执行时丢失。

例如，考虑前面返回数组中最大值的 MaxValue()函数，略微修改该函数，获取数组中最大值的元素索引。为简单起见，如果数组中有多个元素的值都是这个最大值，只提取第一个最大值的索引。为此，修改函数，添加一个 out 参数，如下所示：

```
static int MaxValue(int[] intArray, out int maxIndex)
{
   int maxVal = intArray[0];
   maxIndex = 0;
   for (int i = 1; i < intArray.Length; i++)
   {
      if (intArray[i] > maxVal)
      {
         maxVal = intArray[i];
         maxIndex = i;
      }
   }
   return maxVal;
}
```

可以采用以下方式使用该函数：

```
int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
int maxIndex;
Console.WriteLine("The maximum value in myArray is {0}",
                  MaxValue(myArray, out maxIndex));
Console.WriteLine("The first occurrence of this value is at element {0}",
                  maxIndex + 1);
```

结果如下：

```
The maximum value in myArray is 9
The first occurrence of this value is at element 7
```

注意，必须在函数调用中使用 out 关键字，就像 ref 关键字一样。

在屏幕上显示结果时，给返回的 maxIndex 的值加上 1。这样可以使索引更容易读懂，因此数组的第一个元素称为元素 1，而不是元素 0。

6.2　变量的作用域

在上一节中，读者可能想知道为什么需要利用函数交换数据。原因是 C#中的变量仅能从代码的本地作用域访问。给定的变量有一个作用域，访问该变量要通过这个作用域来实现。

变量的作用域是一个重要的主题，最好用一个示例加以说明。下面的示例将演示变量在一个作用域中定义，但试图在另一个作用域中使用的情形。

试一试：变量的作用域

(1) 对 Ch06Ex01 中的 Program.cs 进行如下修改：

可从 wrox.com 下载源代码

```
class Program
{
   static void Write()
   {
      Console.WriteLine("myString = {0}", myString);
   }

   static void Main(string[] args)
   {
      string myString = "String defined in Main()";
      Write();
      Console.ReadKey();
   }
}
```

代码段 Ch06Ex01\Program.cs

(2) 编译代码，注意显示在任务列表中的错误和警告：

```
The name 'myString' does not exist in the current context
The variable 'myString' is assigned but its value is never used
```

示例的说明

什么地方出错了？不能在 Write()函数中访问在应用程序主体(Main()函数)中定义的变量 myString。

原因是变量是有作用域的，在这个作用域中，变量才是有效的。这个作用域包括定义变量的代码块和直接嵌套在其中的代码块。函数中的代码块与调用它们的代码块是不同的。在 Write()中，没有定义 myString，在 Main()中定义的 myString 则超出了作用域——它只能在 Main()中使用。

实际上，在 Write()中可以有一个完全独立的变量 myString，修改代码，如下所示：

```
class Program
{
```

```
    static void Write()
    {
        string myString = "String defined in Write()";
        Console.WriteLine("Now in Write()");
        Console.WriteLine("myString = {0}", myString);
    }

    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
        Write();
        Console.WriteLine("\nNow in Main()");
        Console.WriteLine("myString = {0}", myString);
        Console.ReadKey();
    }
}
```

这段代码就可以编译，输出结果如图 6-4 所示。

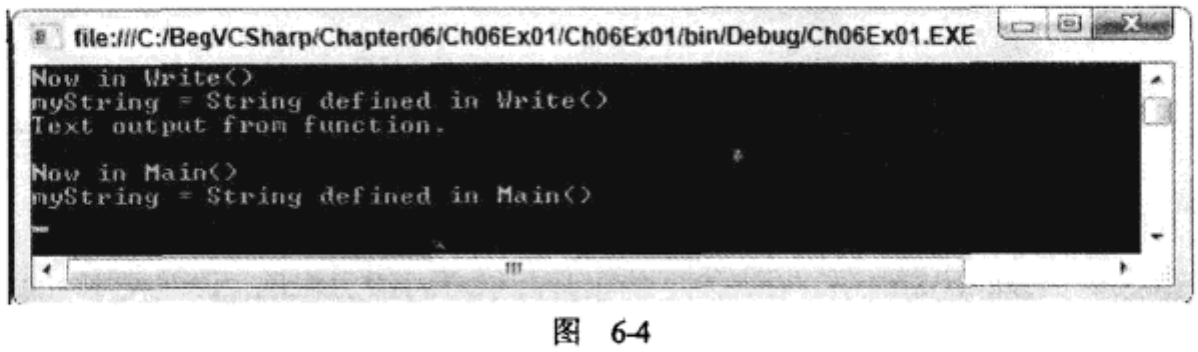


图 6-4

这段代码执行的操作如下：

- Main()定义和初始化字符串变量 myString。
- Main() 把控制权传送给 Write()。
- Write()定义和初始化字符串变量 myString，它与 Main()中定义的 myString 变量完全不同。
- Write()把一个字符串输出到控制台上，该字符串包含在 Write()中定义的 myString 的值。
- Write()把控制权传送回 Main()。
- Main()把一个字符串输出到控制台上，该字符串包含在 Main()中定义的 myString 的值。

其作用域以这种方式覆盖一个函数的变量称为局部变量。还有一种全局变量，其作用域可覆盖多个函数。修改代码，如下所示：

```
class Program
{
    static string myString;

    static void Write()
    {
        string myString = "String defined in Write()";
        Console.WriteLine("Now in Write()");
        Console.WriteLine("Local myString = {0}", myString);
        Console.WriteLine("Global myString = {0}", Program.myString);
    }

    static void Main(string[] args)
    {
        string myString = "String defined in Main()";
```

```
        Program.myString = "Global string";
        Write();
        Console.WriteLine("\nNow in Main()");
        Console.WriteLine("Local myString = {0}", myString);
        Console.WriteLine("Global myString = {0}", Program.myString);
        Console.ReadKey();
    }
}
```

结果如图 6-5 所示。

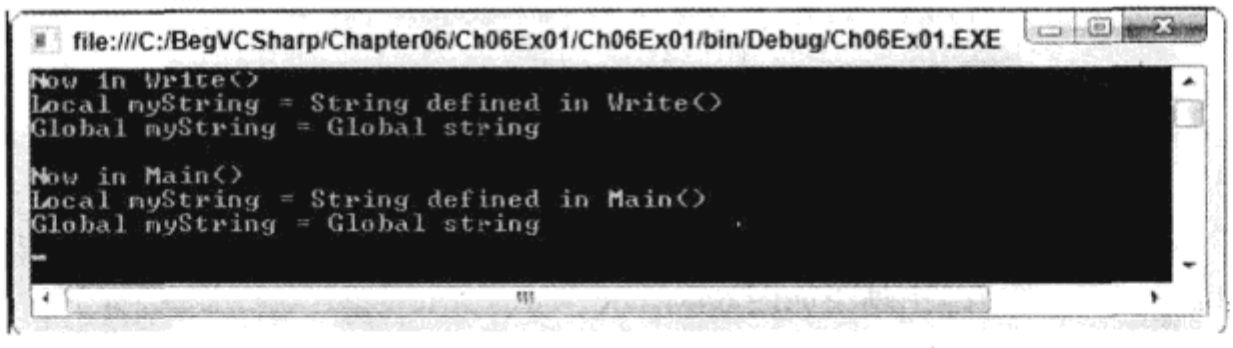


图　6-5

这里添加了另一个变量 myString，这次进一步加深了代码中的名称层次。这个变量定义如下：

```
static string myString;
```

注意，这里也需要 static 关键字。在这种类型的控制台应用程序中，必须使用 static 或 const 关键字，来定义这种形式的全局变量。如果要修改全局变量的值，就需要使用 static，因为 const 禁止修改变量的值。

为了区分这个变量和 Main()与 Write()中的同名局部变量，必须用一个完整限定的名称为变量名分类，参见第 3 章。这里把全局变量称为 Program.myString。注意，在全局变量和局部变量同名时，这是必需的。如果没有局部 myString 变量，就可以使用 myString 表示全局变量，而不需要使用 Program.myString。如果局部变量和全局变量同名，全局变量就会被屏蔽。

全局变量的值在 Main()中设置如下：

```
Program.myString = "Global string";
```

在 Write()中可以通过如下语句访问：

```
Console.WriteLine("Global myString = {0}", Program.myString);
```

为什么不能使用这个技术通过函数交换数据，而要使用前面介绍的参数来交换数据？有时，这确实是一种交换数据的首选方式，但在许多情况下不应使用这种方式。是否使用全局变量取决于函数的位置。使用全局变量的问题在于，它们通常不适合于“常规用途”的函数——这些函数能处理我们所提供的任意数据，而不仅限于处理特定全局变量中的数据。详见本章后面的内容。

6.2.1　其他结构中变量的作用域

上一节的一个要点总结了上述内容，并超出了函数之间的变量作用域。前面说过，变量的作用域包含定义它们的代码块和直接嵌套在其中的代码块。这也可以应用到其他代码块上，例如分支和循环结构的代码块。考虑下面的代码：

```
int i;
for (i = 0; i < 10; i++)
{
   string text = "Line " + Convert.ToString(i);
   Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

字符串变量 text 是 for 循环的局部变量，这段代码不能编译，因为在该循环外部调用的 Console.WriteLine()试图使用该变量 text，这超出了循环的作用域。修改代码，如下所示：

```
int i;
string text;
for (i = 0; i < 10; i++)
{
   text = "Line " + Convert.ToString(i);
   Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

这段代码也会失败，原因是必须在使用变量前对其进行声明和初始化，而 text 是在 for 循环中初始化的。赋给 text 的值在循环块退出时就丢失了。但是还可以进行如下修改：


```
int i;
string text = "";
for (i = 0; i < 10; i++)
{
   text = "Line " + Convert.ToString(i);
   Console.WriteLine("{0}", text);
}
Console.WriteLine("Last text output in loop: {0}", text);
```

代码段 VariableScopeInLoops\Program.cs

这次 text 是在循环外部初始化的，可以访问它的值。这段简单代码的结果如图 6-6 所示。

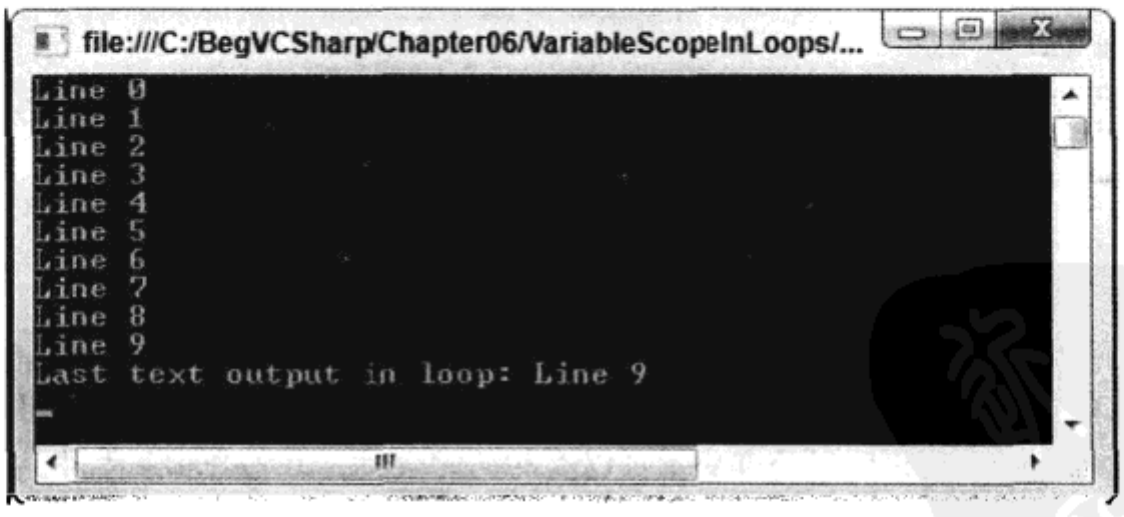


图 6-6

在循环中最后赋给 text 的值可以在循环外部访问。可以看出，这个主题的内容需要花一点时间来掌握。在前面的示例中，循环之前赋给 text 空字符串，而在循环之后的代码中，该 text 就不会是空字符串了，其原因并不明显。

这种情况的解释涉及到分配给 text 变量的内存空间，实际上任何变量都是这样。只声明一个简单变量类型，并不会引起其他的变化。只有在给变量赋值后，这个值才占用一块内存空间。如果这种占据内存空间的行为在循环中发生，该值实际上定义为一个局部值，在循环的外部会超出了其作用域。

即使变量本身没有局部化到循环上，循环所包含的值也局部化到该循环上。但是，在循环外部赋值可以确保该值是主体代码的局部值，在循环内部它仍处于其作用域中。这意味着变量在退出主体代码块之前是没有超出作用域的，所以可以在循环外部访问它的值。

幸好，C#编译器可检测变量作用域的问题，它生成的响应错误信息有助于我们理解变量作用域问题。

最后一个要注意的问题是，应采用“最佳实践方式”。一般情况下，最好在声明和初始化所有变量后，再在代码块中使用它们。一个例外是把循环变量声明为循环块的一部分，例如：

```
for (int i = 0; i < 10; i++)
{
   ...
}
```

其中 i 局部化于循环代码块中，但这是可以的，因为很少需要在外部代码中访问这个计数器。

6.2.2　参数和返回值与全局数据

本节将详细介绍如何通过全局数据以及参数和返回值与函数交换数据。先看看下面的代码：

```
class Program
{
   static void ShowDouble(ref int val)
   {
      val *= 2;
      Console.WriteLine("val doubled = {0}", val);
   }

   static void Main(string[] args)
   {
      int val = 5;
      Console.WriteLine("val = {0}", val);
      ShowDouble(ref val);
      Console.WriteLine("val = {0}", val);
   }
}
```

这段代码与本章前面的代码稍有不同，在前面的示例中，在 Main()中使用了变量名 myNumber，这说明了局部变量可以有相同的名称，且不会相互干涉。这里列出的两个代码示例更加类似，以便我们集中精力研究它们的区别，而无需担心变量名。

和下面的代码比较：

```
class Program
{
   static int val;

   static void ShowDouble()
   {
      val *= 2;
      Console.WriteLine("val doubled = {0}", val);
   }

   static void Main(string[] args)
   {
      val = 5;
      Console.WriteLine("val = {0}", val);
      ShowDouble();
      Console.WriteLine("val = {0}", val);
   }
}
```

这两个 showDouble()函数的结果是相同的。

使用哪种方法并没有什么硬性规定，这两种方法都十分有效。但是，需要考虑一些规则。

首先，在第一次讨论这个问题时，使用全局值的 showDouble()版本只使用全局变量 val。为了使用这个版本，就必须使用这个全局变量。这会对该函数的多样性有轻微的限制，如果要存储结果，就必须总是把这个全局变量值复制到其他变量中。另外，全局数据可以在应用程序的其他地方由代码修改，这会导致预料不到的结果(其值可能会改变，等我们没有认识到这一点时为时已晚)。

但是，损失了多样性常常是有好处的。我们常常希望把一个函数只用于一个目的，使用全局数据存储能减少在函数调用中犯错的可能性，例如把它传递给错误的变量。

当然，也可以说，这种简化实际上使代码更难理解。显式指定参数可以一眼看出发生了什么改变。如 FunctionName(val1, out val2)函数调用，其中 val1 和 val2 都是要考虑的重要变量，在函数执行结束后，val2 就会被赋予一个新值。反之，如果这个函数不带参数，就不能对它处理了什么数据做任何假设。

最后，记住未必总是能使用全局数据。本书的后面将介绍在不同的文件中编写的代码，以及不同名称空间中的代码如何通过函数彼此通信。像这样的情况，代码常常要分开编写，显然不能使用全局存储方式。

总之，可以自由选择使用哪种技术来交换数据。一般情况下，最好使用参数，而不使用全局数据，但有时使用全局数据更合适，使用这种技术并没有错。

6.3 Main()函数

前面介绍了创建和使用函数时涉及的大多数简单技术，下面详细论述 Main()函数。

Main()是 C#应用程序的入口点，执行这个函数就是执行应用程序。也就是说，在执行过程开始时，会执行 Main()函数，在 Main()函数执行完毕时，执行过程就结束了。

这个函数可以返回 void 或 int，有一个可选参数 string[] args。Main()函数可以使用如下 4 种版本：

```
static void Main()
static void Main(string[] args)
static int Main()
static int Main(string[] args)
```

上面的第三、四个版本返回一个 int 值，它们可以用于表示应用程序如何终止，通常用作一种错误提示(但这不是强制的)，一般情况下，返回 0 反映了“正常”的终止(即应用程序执行完毕，并安全地终止)。

Main()的可选参数 args 是从应用程序的外部接受信息的方法，这些信息在运行期间指定，其形式是命令行参数。

前面已经遇到了命令行参数，在命令行上执行应用程序时，通常可以直接指定信息，如在执行应用程序时加载一个文件。例如，考虑 Windows 中的记事本应用程序。在命令提示窗口中键入 notepad，或者在 Windows 的 Start 菜单中选择 Run 选项，再在打开的窗口中键入 notepad，就可以运行该应用程序。也可以键入 notepad "myfile.txt"，结果是 Notepad 在运行时将加载文件 myfile.txt，如果该文件不存在，Notepad 也会创建该文件。这里 myfile.txt 是一个命令行参数。利用 args 参数，可以编写以类似方式工作的控制台应用程序。

在执行控制台应用程序时，指定的任何命令行参数都放在这个 args 数组中，接着可以根据需要在应用程序中使用这些参数。下面用一个示例来说明。这个示例可以指定任意数量的命令行参数，每个参数都输出到控制台上。

试一试：命令行参数

(1) 在 C:\BegVCSharp\Chapter06 目录中创建一个新控制台应用程序 Ch06Ex04。

(2) 把下列代码添加到 Program.cs 中：


```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("{0} command line arguments were specified:",
                        args.Length);
      foreach (string arg in args)
         Console.WriteLine(arg);
      Console.ReadKey();
   }
}
```

代码段 Ch06Ex04\Program.cs

(3) 打开项目的属性页面(在 Solution Explorer 窗口中右击 Ch06Ex04 项目名称，选择 Properties 选项)。

(4) 选择 Debug 页面，在 Command Line Arguments 设置中添加所希望的命令行参数，如图 6-7 所示。

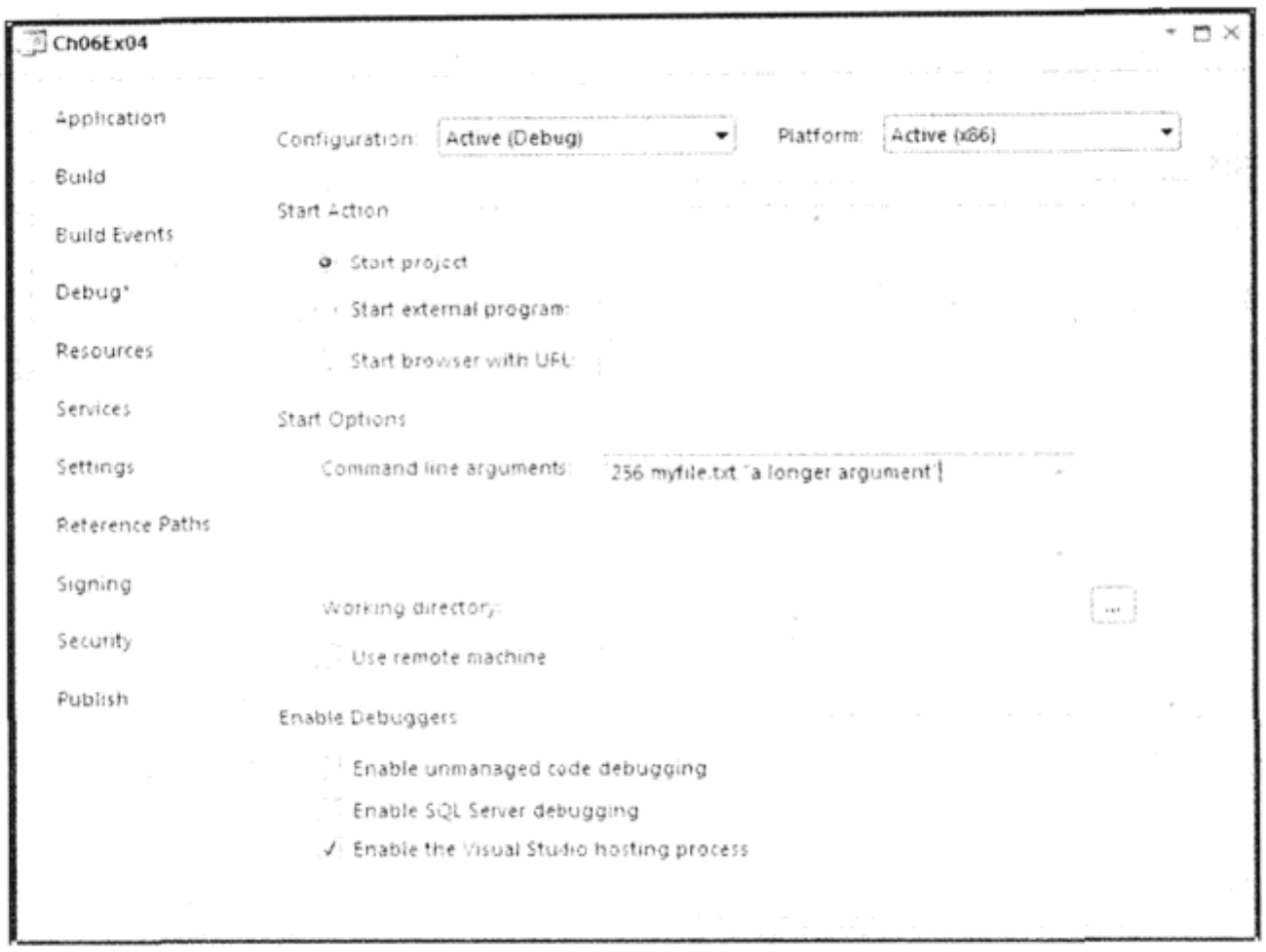


图 6-7

(5) 运行应用程序，输出结果如图 6-8 所示。

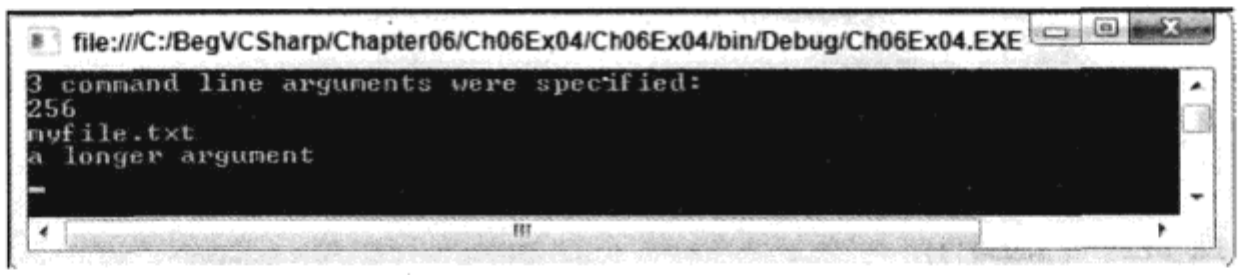


图 6-8

示例的说明

这里使用的代码非常简单：

```
Console.WriteLine("{0} command line arguments were specified:",
                  args.Length);
foreach (string arg in args)
  Console.WriteLine(arg);
```

使用 args 参数与使用其他字符串数组类似。我们没有对参数进行任何异样的操作，只是把指定的信息写到屏幕上。在本示例中，通过 IDE 中的项目属性提供参数，这是一种很便捷的方式，只要在 IDE 中运行应用程序，就可以使用相同的命令行参数，无需每次都在命令行提示窗口中键入它们。在项目输出所在的目录(C:\BegVSharp\Chapter06\Ch06Ex04\bin\Debug)下打开命令提示窗口，键入下述代码，也可以得到同样的结果：

```
Ch06Ex04 256 myFile.txt "a longer argument"
```

每个参数都用空格分开，如果参数包含空格，就可以用双引号把参数括起来，这样才不会把这

个参数解释为多个参数。

6.4 结构函数

第 5 章介绍了结构类型，它可在一个地方存储多个数据元素，结构可以做的工作远不止此。一个重要的功能就是结构可以包含函数和数据。这初看起来很奇怪，但实际上是非常有用的。例如，考虑以下结构：

```
struct customerName
{
   public string firstName, lastName;
}
```

如果变量的类型是 customerName，并且要在控制台上输出一个完整的名称，就必须从其组件部分建立该名称。例如，customerName 变量 myCustomer 可以使用下述语法：

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine("{0} {1}", mycustomer.firstName, mycustomer.lastName);
```

把函数添加到结构中，就可以集中处理常见任务，从而简化这个过程。可以把合适的函数添加到结构类型中，如下所示：

```
struct customerName
{
   public string firstName, lastName;

   public string Name ()
   {
      return firstName + " " + lastName;
   }
}
```

看起来这与本章前面的其他函数很类似，但没有使用static 修饰符。本书将在后面阐明其原因，现在知道该关键字不是结构函数所必须的即可。这个函数的用法如下所示：

```
customerName myCustomer;
myCustomer.firstName = "John";
myCustomer.lastName = "Franklin";
Console.WriteLine(mycustomer.Name());
```

这个语法比前面的语法简单得多，也更容易理解。注意，Name()函数可以直接访问 firstName 和 lastName 结构成员，在 customerName 结构中，它们可以被看作是全局成员。

6.5 函数的重载

本章的前面介绍了在调用函数时，必须匹配函数的签名。这表明，需要让多个函数操作不同类

型的变量。函数重载允许创建多个同名函数，这些函数可使用不同的参数类型。例如，前面使用了下述代码，其中包含函数 MaxValue()：

```
class Program
{
   static int MaxValue(int[] intArray)
   {
      int maxVal = intArray[0];
      for (int i = 1; i < intArray.Length; i++)
      {
         if (intArray[i] > maxVal)
            maxVal = intArray[i];
      }
      return maxVal;
   }

   static void Main(string[] args)
   {
      int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};
      int maxVal = MaxValue(myArray);
      Console.WriteLine("The maximum value in myArray is {0}", maxVal);
      Console.ReadKey();
   }
}
```

这个函数只能用于处理 int 数组，现在要为不同的参数类型提供不同名称的函数，可以把上述函数重命名为 IntArrayMaxValue()，添加诸如 DoubleArrayMaxValue()的函数处理其他类型。另外，还可以在代码中添加如下函数：

```
...
static double MaxValue(double[] doubleArray)
{
   double maxVal = doubleArray[0];
   for (int i = 1; i < doubleArray.Length; i++)
   {
      if (doubleArray[i] > maxVal)
         maxVal = doubleArray[i];
   }
   return maxVal;
}
...
```

这里的区别是使用了 double 值。函数名称 MaxValue()是相同的，但其签名是不同的。这是因为如前所述，函数的签名包含函数的名称及其参数。用相同的签名定义两个函数是错误的，但因为这两个函数的签名不同，所以是可行的。

函数的返回类型不是其签名的一部分，所以不能定义两个仅返回类型不同的函数，它们实际上有相同的签名。

添加了前面的代码后，现在有两个版本的 MaxValue()，它们的参数是 int 和 double 数组，分别

返回 int 或 double 最大值。

这种代码的优点是不必显式地指定要使用哪个函数。只需提供一个数组参数，就可以根据使用的参数类型执行相应的函数。

此时，应注意 VS 和 VCE 中 IntelliSense 的另一项功能。如果在应用程序中有上述两个函数，而且要在 Main()中键入函数的名称，IDE 就可以显示出可用的重载函数。如果键入下面的代码：

```
double result = MaxValue(
```

IDE 就会提供两个 MaxValue()版本的信息，使用上下箭头键在其间滚动，如图 6-9 所示。

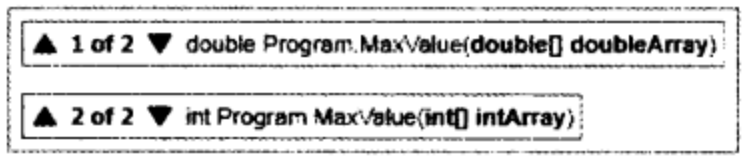


图　6-9

在重载函数时，应包括函数签名的所有方面。例如，有两个不同的函数，它们分别带有值参数和引用参数：

```
static void ShowDouble(ref int val)
{
   ...
}

static void ShowDouble(int val)
{
   ...
}
```

选择使用哪个版本纯粹是根据函数调用是否包含 ref 关键字来确定的。下面的代码将调用引用版本：

```
ShowDouble(ref val);
```

下面的代码是调用值版本：

```
ShowDouble(val);
```

另外，还可以根据参数的个数等来区分函数。

6.6　委托

委托(delegate)是一种可以把引用存储为函数的类型。这听起来相当棘手，但其机制是非常简单的。委托最重要的用途在本书后面介绍到事件和事件处理时才能解释清楚，但这里也将介绍有关委托的许多内容。委托的声明非常类似于函数，但不带函数体，且要使用 delegate 关键字。委托的声明指定了一个返回类型和一个参数列表。

在定义了委托后，就可以声明该委托类型的变量。接着把这个变量初始化为与委托有相同返回类型和参数列表的函数引用。之后，就可以使用委托变量调用这个函数，就像该变量是一个函数一样。

有了引用函数的变量后，还可以执行不能用其他方式完成的操作。例如，可以把委托变量作为参数传递给一个函数，这样，该函数就可以使用委托调用它引用的任何函数，而且在运行之前无需知道调用的是哪个函数。下面的示例使用委托访问两个函数中的一个。

试一试：使用委托来调用函数

(1) 在 C:\BegVCSharp\Chapter06 目录中创建一个新控制台应用程序 Ch06Ex05。

(2) 把下列代码添加到 Program.cs 中：

可从 wrox.com 下载源代码

```
class Program
{
   delegate double ProcessDelegate(double param1, double param2);

   static double Multiply(double param1, double param2)
   {
      return param1 * param2;
   }
   static double Divide(double param1, double param2)
   {
      return param1 / param2;
   }

   static void Main(string[] args)
   {
      ProcessDelegate process;
      Console.WriteLine("Enter 2 numbers separated with a comma:");
      string input = Console.ReadLine();
      int commaPos = input.IndexOf(',');
      double param1 = Convert.ToDouble(input.Substring(0, commaPos));
      double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
                                       input.Length - commaPos - 1));
      Console.WriteLine("Enter M to multiply or D to divide:");
      input = Console.ReadLine();
      if (input == "M")
         process = new processDelegate(Multiply);
      else
         process = new processDelegate(Divide);
      Console.WriteLine("Result: {0}", process(param1, param2));
      Console.ReadKey();
   }
}
```

代码段 Ch06Ex05\Program.cs

(3) 执行代码，结果如图 6-10 所示。

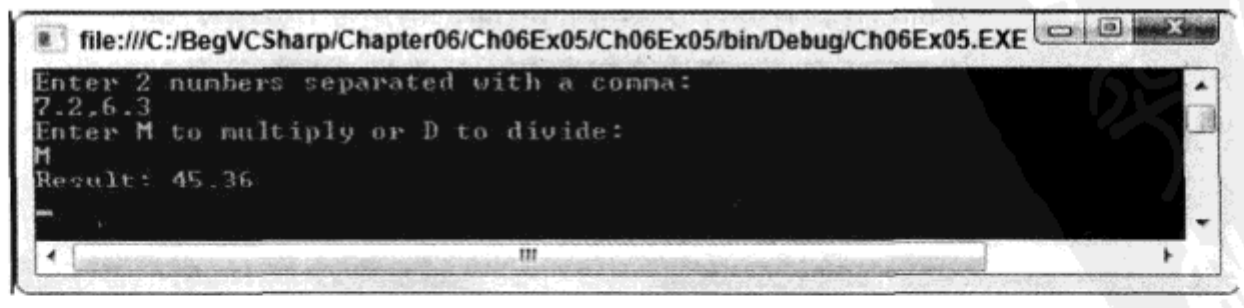


图 6-10

示例的说明

这段代码定义了一个委托 ProcessDelegate，其返回类型和参数与函数 Multiply()和 Divide()相匹配。委托的定义如下所示：

```
delegate double ProcessDelegate(double param1, double param2);
```

delegate 关键字指定该定义是用于委托的，而不是用于函数的(该定义所在的位置与函数定义相同)。接着，该定义指定 double 返回类型和两个 double 参数。实际使用的名称可以是任意的，所以可以给委托类型和参数指定任意名称。这里委托名是 ProcessDelegate，double 参数名是 param1 和 param2。

Main()中的代码首先使用新的委托类型声明一个变量：

```
static void Main(string[] args)
{
   ProcessDelegate process;
```

接着用一些比较标准的C#代码请求由逗号分隔的两个数字，并把这些数字放在两个double变量中：

```
Console.WriteLine("Enter 2 numbers separated with a comma:");
 string input = Console.ReadLine();
 int commaPos = input.IndexOf(',');
 double param1 = Convert.ToDouble(input.Substring(0, commaPos));
 double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
                                  input.Length -- commaPos -- 1));
```

为了说明问题，这里没有验证用户输入的有效性。如果这些是“现实中的”代码，就应花更多的时间来确保在局部变量 param1 和 param2 中得到有效的值。

接着，询问用户是要相乘，还是相除这两个数字：

```
Console.WriteLine("Enter M to multiply or D to divide:");
 input = Console.ReadLine();
```

根据用户的选择，初始化 process 委托变量：

```
if (input == "M")
   process = new processDelegate(Multiply);
else
   process = new processDelegate(Divide);
```

要把一个函数引用赋给委托变量，需要使用略显古怪的语法。这个过程比较类似于给数组赋值，必须使用 new 关键字创建一个新委托。在这个关键字的后面，指定委托类型，提供一个引用所需函数的参数，该函数是 Multiply()或 Divide()。注意这个参数与委托类型或目标函数的参数不匹配，这是委托赋值的一个独特语法，参数是要使用的函数名，且不带括号。

实际上，这里可以使用略微简单的语法：

```
if (input == "M")
   process = Multiply;
```

```
else
   process = Divide;
```

编译器会发现，process 变量的委托类型匹配两个函数的签名，于是自动初始化一个委托。可以自行确定使用哪个语法，但一些人喜欢使用较长的版本，因为它更容易一眼看出会发生什么。

最后，使用该委托调用所选的函数。无论委托引用的是什么函数，该语法都是有效的：

```
        Console.WriteLine("Result: {0}", process(param1, param2));
        Console.ReadKey();
    }
```

这里把委托变量看作一个函数名。但与函数不同，我们还可以对这个变量执行更多的操作，例如，通过参数将其传递给一个函数，这个函数的一个简单示例如下：

```
static void ExecuteFunction(ProcessDelegate process)
{
   process(2.2, 3.3);
}
```

就像选择一个要使用的“插件”一样，把它们传递给函数委托，就可以控制函数的执行。例如，一个函数要对字符串数组按照字母进行排序。对列表排序有几个不同的方法，它们的性能取决于要排序的列表特性。使用委托可以把一个排序算法函数委托传递给排序函数，指定要使用的方法。

委托有许多用途，但如前所述，它们的大多数常见用途主要与事件处理有关，具体内容详见第13章。

6.7 小结

本章相当全面地介绍了 C#代码中函数的用法。函数提供的其他许多特性(特别是委托)比较抽象，我们将在第 8 章的面向对象编程中讨论它们。

如何使用函数的知识是将来要完成的所有编程工作的核心。后面的章节，特别是学习 OOP(从第 8 章开始)的部分，将介绍函数的正式结构，以及如何把它们应用于类。从现在开始，把代码放在可重用块中将成为 C#编程中最有用的部分。

6.8 练习

(1) 下面两个函数都存在错误，请指出这些错误。

```
static bool Write()
{
   Console.WriteLine("Text output from function.");
}

static void myFunction(string label, params int[] args, bool showLabel)
{
   if (showLabel)
      Console.WriteLine(label);
   foreach (int i in args)
```

```
        Console.WriteLine("{0}", i);
    }
```

(2) 编写一个应用程序，该程序使用两个命令行参数，分别把值放在一个字符串和一个整型变量中，然后显示这些值。

(3) 创建一个委托，在请求用户输入时，使用它模拟 Console.ReadLine()函数。

(4) 修改下面的结构，使之包含一个返回订单总价格的函数。

```
struct order
{
   public string itemName;
   public int    unitCount;
   public double unitCost;
}
```

(5) 在 order 结构中添加另一个函数，该结构返回一个格式化的字符串(一行文本，以合适的值替换用尖括号括起来的斜体条目)。

```
    Order Information: <unit count> <item name> items at $<unit cost> each,
total cost $<total cost>
```

附录 A 给出了练习答案。

6.9　本章要点

主　题	重 要 概 念
定义函数	函数用函数名、0 个或多个参数及返回类型来定义。函数的名称和参数统称为函数的签名。可以定义名称相同、但签名不同的多个函数——这称为函数的重载。也可以在结构类型中定义函数
返回值和参数	函数的返回类型可以是任意类型，如果函数没有返回值，其返回类型就是 void。参数也可以是任意类型，由一个用逗号分隔的类型和名称对组成。调用函数时，所指定的参数的类型和顺序必须匹配函数的定义。个数不定的特定类型的参数可以通过参数数组来指定。参数可以指定为 ref 或 out，以便给调用者返回值
变量作用域	变量根据定义它们的代码块来界定其使用范围。代码块包括方法和其他结构，例如循环体。可以在不同的作用域中定义多个不同的同名变量
命令行参数	在执行应用程序时，控制台应用程序中的 Main()函数可以接收传送给应用程序的命令行参数。这些参数用空格隔开，但较长的参数可以放在引号中传送
委托	除了直接调用函数之外，还可以通过委托调用它们。委托是用返回类型和参数列表定义的变量。给定的委托类型可以匹配返回类型和参数与委托定义相同的方法

第 7 章

调试和错误处理

本章内容：

- IDE 中的调试方法
- C#中的错误处理技术

本书到目前为止介绍了在 C#中进行简单编程的所有基础知识。在讨论本书后面章节的面向对象编程之前，先看看 C#代码中的调试和错误处理问题。

代码中有时难免存在错误。无论程序员多么优秀，程序总是会出现一些问题，出色的程序员会找出其中一部分错误，并更正它们。当然，一些问题比较小，不会影响应用程序的执行，例如，按钮上的拼写错误等，但一些错误可能比较严重，会导致应用程序完全失败(通常称为致命错误)，致命错误包括妨碍代码编译的简单错误(语法错误)，或者只在运行期间发生的更严重的错误。一些错误可能会更微妙。也许应用程序不能给数据库添加一个记录，因为遗漏了一个请求的字段，或者在其他有限制的环境中把错误的数据添加到记录中。应用程序的逻辑在某些方面有瑕疵时，就会产生这样的错误，此类错误称为语义错误(或逻辑错误)。

当应用程序的用户抱怨说程序不能正常工作时，就出现了比较难以处理的错误。此时需要跟踪代码，确定发生了什么问题，并修改代码，使之按照希望的那样工作。在此类情况下，VS 和 VCE 的调试功能就可以大显身手了。本章的第一部分就介绍一些调试技巧，并把它们应用到一些常见问题上。

接着，讨论 C#中的错误处理技术。利用它们，可以对可能发生错误的地方采取预防措施，并编写弹性代码来处理可能会致命的错误。这些是 C#语言的一部分，而不是调试功能，但 IDE 也提供了一些工具来帮助我们处理错误。

7.1 VS 和 VCE 中的调试

前面提到，可以采用两种方式执行应用程序：调试模式或非调试模式。在 VS 或 VCE 中执行应用程序时，它默认在调试模式下执行。例如，按下 F5 键或单击工具栏中的绿色 Play 按钮时，

就是在调试模式下执行应用程序。要在非调试模式下执行应用程序，应选择 Debug | Start Without Debugging，或者按下 Ctrl+F5 键。

VS 和 VCE 都允许在两种配置下创建应用程序：调试(默认)和发布(实际上，还可以定义其他配置，但这是一种高级技术，本书不涉及)。使用标准工具栏中的 Solution Configurations 下拉框可以在这两种配置之间切换。

在VCE 中，默认情况下不激活这个下拉列表。为了阅读本章，应启用它，方法是选择 Tools | Options，在Options 对话框中选择 Show All Settings，再选择Projects and Solutions 类别中的 General 子类别，启用 Show Advanced Build Configurations 选项。

在调试配置下生成应用程序，在调试模式下运行程序时，并不仅仅是运行编写好的代码。调试程序包含了应用程序的符号信息，所以 IDE 知道执行每行代码时发生了什么。符号信息意味着跟踪(例如)未编译代码中使用的变量名，这样，它们就可以匹配已编译的机器码应用程序中现有的值，而机器码程序不包含人们易于读取的信息。此类信息包含在.pdb 文件中，这些文件位于计算机的 Debug 目录下。它们可以执行许多有用的操作，包括：

- 向 IDE 输出调试信息
- 在执行应用程序期间查看和编辑变量的值
- 暂停程序和重启程序
- 在代码的某个位置自动暂停程序的执行
- 一次执行程序中的一行代码
- 在应用程序的执行期间监视变量内容的变化
- 在运行期间修改变量内容
- 测试函数的调用

在发布配置中，优化应用程序代码，但我们不能执行这些操作。但发布版本运行得比较快，完成了应用程序的开发时，一般应给用户提供发布版本，因为发布版本不需要调试版本所包含的符号信息。

本节介绍调试技巧，以及如何使用它们确定未按预期方式执行的那些代码，并修改它们，这个过程称为调试。按照这些技术的使用方法把它们分为两个部分。一般情况下，可以先中断程序的执行，再进行调试，或者注上标记，以便以后加以分析。在 VS 和 VCE 术语中，应用程序可以处于运行状态，也可以处于中断模式，即暂停正常的执行。下面首先介绍非中断模式(运行期间或正常执行)技术。

7.1.1 非中断(正常)模式下的调试

本书常常使用的一个命令是 Console.WriteLine()函数，它可以把文本输出到控制台上。在开发应用程序时，这个函数可以方便地获得操作的额外反馈，例如：

```
Console.WriteLine("MyFunc() Function about to be called.");
MyFunc ("Do something.");
Console.WriteLine("MyFunc() Function execution completed.");
```

这段代码说明了如何获取 MyFunc()函数的额外信息。这么做完全正确，但控制台的输出结果会

比较混乱。在开发其他类型的应用程序时，如 Windows 窗体应用程序，没有用于输出信息的控制台。作为一种替代方法，可以把文本输出到另一个位置上——IDE 中的 Output 窗口。

第 2 章简要介绍了 Error List 窗口，其他窗口也可以显示在这个位置上。其中一个窗口就是 Output 窗口，在调试时这个窗口非常有用。要显示这个窗口，可以选择 View | Output。在这个窗口中，可以查看与代码的编译和执行相关的信息，包括在编译过程中遇到的错误等，还可以将自定义的诊断信息直接写到窗口中，来使用这个窗口显示自定义信息。该窗口如图 7-1 所示。

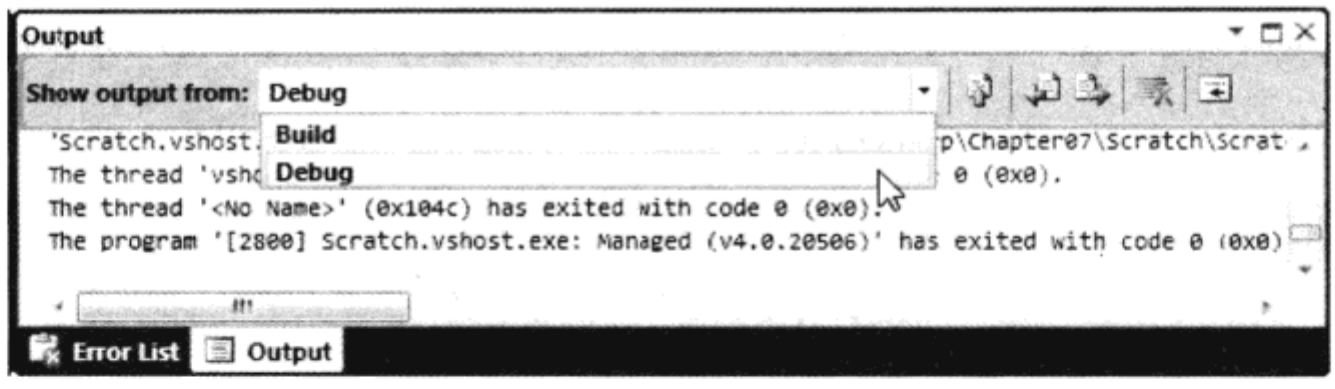


图　7-1

Output 窗口有两种模式 Build 和 Debug，使用其中的下拉菜单可以选择这些模式。Build 和 Debug 模式分别显示编译和运行期间的信息。本节提到“写入 Output 窗口”时，实际上是指“写入 Output 窗口的 Debug 模式视图”。

另外，还可以创建一个日志文件，在运行应用程序时，会把信息添加到该日志文件中。把信息写入日志文件所用的技巧与把文本写到 Output 窗口上所用的技巧相同，但需要理解如何从 C#应用程序中访问文件系统。我们把这个功能放在后面的章节中，因为目前不必了解文件访问技巧也可以完成很多工作。

1. 输出调试信息

在运行期间把文本写入 Output 窗口是非常简单的。只要用需要的调用替代 Console. WriteLine() 调用，就可以把文本写到希望的地方。此时可以使用如下两个命令：

- Debug.WriteLine()
- Trace.WriteLine()

这两个命令函数的用法几乎完全相同，但有一个重要区别。第一个命令仅在调试模式下运行，而第二个命令还可用于发布程序。实际上，Debug.WriteLine()命令甚至不能编译为可发布的程序，在发布版本中，该命令会消失，这肯定有其优点(首先，编译好的代码文件比较小)。实际上，一个源文件可以创建出两个版本的应用程序。调试版本显示所有的额外诊断信息，而发布版本没有这个开销，也不向用户显示信息，否则会引起用户的反感。

这两个函数的用法与 Console.WriteLine()是不同的。其唯一的字符串参数用于输出消息，而不需要使用{X}语法插入变量值。这意味着必须使用+等串联运算符在字符串中插入变量值。它们还可以有第二个字符串参数，用于显示输出文本的类别，这样，如果应用程序的不同地方输出了类似的消息，我们就可以马上确定 Output 窗口中显示的是哪些输出信息。

这些函数的一般输出如下所示：

```
<category>: <message>
```

例如，下面的语句把 MyFunc 作为可选的类别参数：

```
Debug.WriteLine("Added 1 to i", "MyFunc");
```

其结果为：

```
MyFunc: Added 1 to i
```

下面的示例按这种方式输出调试信息。

试一试：把文本输出到 Output 窗口

(1) 在 C:\BegVCSharp\Chapter07 目录中创建一个新的控制台应用程序 Ch07Ex01。

(2) 修改代码，如下所示：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace Ch07Ex01
{
   class Program
   {
      static void Main(string[] args)
      {
         int[] testArray = {4, 7, 4, 2, 7, 3, 7, 8, 3, 9, 1, 9};
         int[] maxValIndices;
         int maxVal = Maxima(testArray, out maxValIndices);
         Console.WriteLine("Maximum value {0} found at element indices:",
                           maxVal);
         foreach (int index in maxValIndices)
         {
            Console.WriteLine(index);
         }
         Console.ReadKey();
      }
      static int Maxima(int[] integers, out int[] indices)
      {
         Debug.WriteLine("Maximum value search started.");
         indices = new int[1];
         int maxVal = integers[0];
         indices[0] = 0;
         int count = 1;
         Debug.WriteLine(string.Format(
             "Maximum value initialized to {0},at element index 0." ,maxVal))
         for (int i = 1; i < integers.Length; i++)
         {
            Debug.WriteLine(string.Format(
```

```
                "Now looking at element at index {0}.",i));
            if (integers[i] > maxVal)
            {
               maxVal = integers[i];
               count = 1;
               indices = new int[1];
               indices[0] = i;
               Debug.WriteLine(string.Format(
                  "New maximum found. New value is{0},at element index{1}. " ,
                  maxVal, i));
            }
            else
            {
               if (integers[i] == maxVal)
               {
                  count++;
                  int[] oldIndices = indices;
                  indices = new int[count];
                  oldIndices.CopyTo(indices, 0);
                  indices[count - 1] = i;
                  Debug.WriteLine(string.Format(
                     "Duplicate maximum found at element index {0}.", i));
               }
            }
         }
Trace.WriteLine(string.Format(
             "Maximum value {0} found, with {1} " occurrences.", maxVal,count));
         Debug.WriteLine("Maximum value search completed.");
         return maxVal;
      }
   }
}
```

代码段 Ch07Ex01\Program.cs

(3) 在 Debug 模式下执行代码，结果如图 7-2 所示。

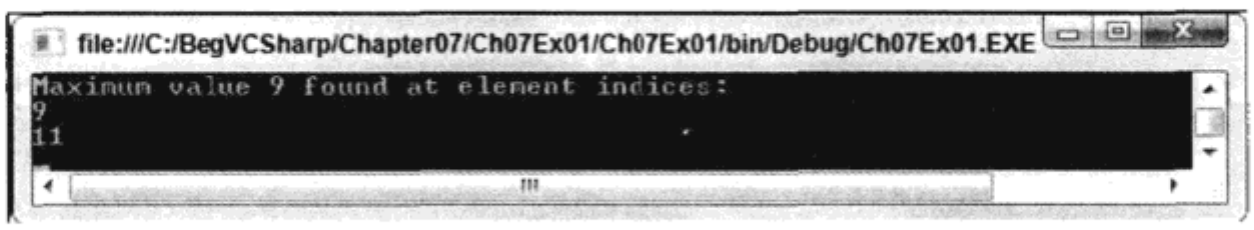


图 7-2

(4) 中断应用程序的执行，查看 Output 窗口中的内容(在 Debug 模式下)，如下所示(有删节)：

```
...
Maximum value search started.
Maximum value initialized to 4, at element index 0.
Now looking at element at index 1.
New maximum found. New value is 7, at element index 1.
Now looking at element at index 2.
Now looking at element at index 3.
Now looking at element at index 4.
```

```
Duplicate maximum found at element index 4.
Now looking at element at index 5.
Now looking at element at index 6.
Duplicate maximum found at element index 6.
Now looking at element at index 7.
New maximum found. New value is 8, at element index 7.
Now looking at element at index 8.
Now looking at element at index 9.
New maximum found. New value is 9, at element index 9.
Now looking at element at index 10.
Now looking at element at index 11.
Duplicate maximum found at element index 11.
Maximum value 9 found, with 2 occurrences.
Maximum value search completed.
The thread 'vshost.RunParkingWindow'(0x110c) has exited with code 0 (0x0).
The thread '<No Name>' (0x688) has exited with code 0 (0x0).
The program '[4568] Ch07Ex01.vshost.exe: Managed' (v4.0.20506) ' has exited with code 0 (0x0).
```

(5) 使用标准工具栏上的下拉菜单，切换到 Release 模式，如图 7-3 所示。

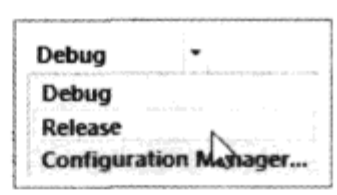


图 7-3

(6) 再次运行程序，这次是在 Release 模式下运行，并在执行中止时，再查看一下 Output 窗口。结果如下所示(有删节)：

```
...
Maximum value 9 found, with 2 occurrences.
The thread 'Vshost.RunParkingWindow' (0xa78) has exited with code 0 (0x0).
The thread '<No Name>' (0x130c) has exited with code 0 (0x0).
The program '[4348] Ch07Ex01.vshost.exe: Managed (v4.0.20506)' has exited with code 0 (0x0).
```

示例的说明

这个应用程序是第 6 章中一个示例的扩展版本，它使用一个函数计算整数数组中的最大值。这个版本也返回一个索引数组，表示最大值在数组中的位置，以便调用代码处理这些元素。

首先在代码开头使用了一个额外的 using 指令：

```
using System.Diagnostics;
```

这简化了本例前面讨论的函数访问，因为它们包含在 System.Diagnostics 名称空间中，没有这个 using 语句，下面的代码：

```
Debug.WriteLine("Bananas");
```

就需要进一步加以限定，重新编写这行语句，如下所示：

```
System.Diagnostics.Debug.WriteLine("Bananas");
```

using 语句使代码更简单，缩短了代码的长度。

Main()中的代码仅初始化一个测试用的整数数组 testArray，并声明了另一个整数数组 maxValIndices，以存储 Maxima()的索引输出结果(执行计算的函数)，接着调用这个函数。函数返回后，代码就会输出结果。

Maxima()稍复杂一些，但没有使用前面介绍的那么多代码。在数组中进行搜索的方式与第 6 章

的 MaxVal()函数类似，但要用一个记录存储最大值的索引。

特别需要注意用来跟踪索引的函数(而不是输出调试信息的那些代码行)。Maxima()并没有返回一个足以存储源数组中每个索引的数组(需要与源数组有相同的维数)，而是返回一个正好能容纳搜索到的索引的数组。这可以在搜索过程中连续重建不同长度的数组来实现。这是必要的，因为一旦创建好数组，就不能重新设置长度。

开始搜索时，假定源数组(本地称为 integers)中的第一个元素就是最大值，且数组中只有一个最大值。因此可以为 maxVal(函数的返回值，即搜索到的最大值)和 indices(out 参数数组，存储搜索到的最大值的索引)设置值。MaxVal 被赋予 integers 中第一个元素的值，indices 被赋予一个值 0，即数组中第一个元素的索引。在变量 count 中存储搜索到的最大值的个数，以跟踪 indices 数组。

函数的主体是一个循环，它迭代 integers 数组中的各个值，但忽略第一个值，因为它已经处理过了。每个值都与 maxVal 的当前值进行比较，如果 maxVal 更大，就忽略该值。如果当前处理的值比 maxVal 大，就修改 maxVal 和 indices，以反映这种情况。如果当前处理的值与 maxVal 相等，就递增 count，用一个新数组替代 indices。这个新数组比旧 indices 数组多一个元素，包含搜索到的新索引。

最后一个功能的代码如下所示：

```
if (integers[i] == maxVal)
 {
   count++;
   int[] oldIndices = indices;
   indices = new int[count];
   oldIndices.CopyTo(indices, 0);
   indices[count - 1] = i;
   Debug.WriteLine(string.Format(
      "Duplicate maximum found at element index {0}.", i));
 }
```

这段代码把旧 indices 数组备份到 if 代码块的 oldIndices 局部整型数组中。注意使用<array>.CopyTo()函数把 oldIndices 中的值复制到新的 indices 数组中。这个函数的参数是一个目标数组和一个用于复制第一个元素的索引，并把所有的值都粘贴到目标数组中。

在代码中，各个文本部分都使用 Debug.WriteLine()和 Trace.WriteLine()函数来进行输出，这些函数使用 string.Format()函数把变量值嵌套在字符串中，其方式与 Console.WriteLine()相同。这比使用+串联运算符更加高效。

在 Debug 模式下运行应用程序时，其最终结果是一个完整的记录，它记述了在循环中计算出结果所采取的步骤。在 Release 模式下，仅能看到计算的最终结果，因为没有调用 Debug.WriteLine()函数。

除了这些 WriteLine()函数外，还需要注意其他一些问题。首先是 Console.Write()的等价函数：

- Debug.Write()
- Trace.Write()

这两个函数使用的语法与 WriteLine()函数相同(一个或两个参数，即一个消息和可选的类别)，但它们是有区别的，因为它们没有添加行尾字符。

还有下列命令：

- Debug.WriteLineIf()
- Trace.WriteLineIf()
- Debug.WriteIf()

- Trace.WriteIf()

这些函数的参数都与没有 if 的对应函数相同，但增加了一个必选参数，且该参数放在列表参数的最前面。这个参数的值为布尔值(或者计算结果为布尔值的表达式)，只有这个值为 true 时，函数才会输出文本。使用这些函数可以有条件地把文本输出到 Output 窗口中。

例如，只需在某些情况下输出调试信息，所以代码中有许多 Debug.WriteLineIf()语句，它们都取决于具体的条件。如果没有这个条件，就不显示它们，以防 Output 窗口显示多余的信息。

2. 跟踪点

另一种把信息输出到 Output 窗口中的方法是使用跟踪点。这是 VS 的一个功能，而不是 C#的功能，但其作用与使用 Debug.WriteLine()相同。它实际上是输出调试信息且不修改代码的一种方式。

只能在 VS 中使用跟踪点，不能在 VCE 中使用。如果读者使用的是 VCE，就可以跳过本节。

为了演示跟踪点，可以使用它们替代上一个示例中的调试命令(请参阅本章的下载代码中的 Ch07Ex01TracePoints 文件)。添加跟踪点的过程如下所示：

(1) 把光标放在要插入跟踪点的代码行上。注意，跟踪点会在执行这行代码之前被处理。

(2) 右击该行代码，选择 Breakpoint | Insert Tracepoint。

(3) 在打开的 When Breakpoint Is Hit 对话框中，在 Print a message:文本框中键入要输出的字符串。如果要输出变量值，应把变量名放在花括号中。

(4) 单击 OK 按钮。在包含跟踪点的代码行左边会出现一个红色的菱形，该行代码也会突出显示为红色。

看一下添加跟踪点的对话框标题和所需要的菜单选项，显然，跟踪点是断点的一种形式(可以暂停应用程序的执行，就像断点一样)。断点一般用于更高级的调试目的，本章稍后将介绍断点。

图 7-4 显示了 Ch07Ex01TracePoints 中第 31 行所需的跟踪点。在删除已有的 Debug.WriteLine()语句后，对代码行编号。

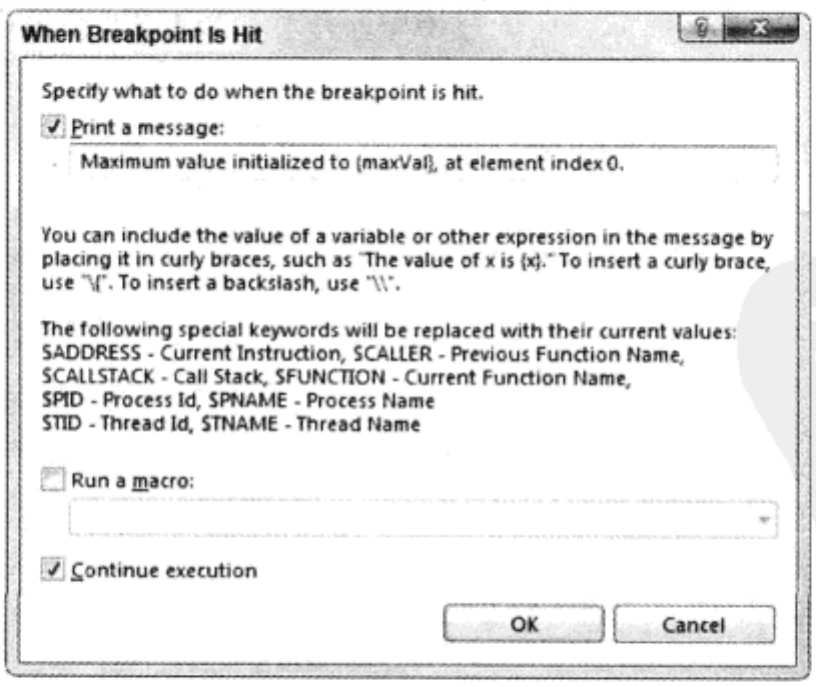


图　7-4

如图 7-4 中的文本所示，跟踪点允许插入与跟踪点的位置和上下文相关的其他有用信息。用户应试着使用这些值，尤其是$FUNCTION 和$CALLER，看看可以得到什么额外信息。还可以看出，跟踪点可以执行宏，但这是一个高级功能，这里不予以介绍。

还有一个窗口(只能在 VS 中使用)可用于快速查看应用程序中的跟踪点。要显示这个窗口，可以从 VS 菜单中选择 Debug | Windows | Breakpoints。这是显示断点的通用窗口(如前所述，跟踪点是断点的一种形式)。可以定制显示的内容，从这个窗口的 Columns 下拉框中添加 When Hit 列，显示与跟踪点关系更密切的信息。图 7-5 显示的窗口配置了这个列，还显示了添加到 Ch07Ex01TracePoints 中的所有跟踪点。

Program.cs

Ch07Ex01Tracepoints.Program　　Maxima(int[] integers, out int[] indices)

```
25      static int Maxima(int[] integers, out int[] indices)
26      {
27          indices = new int[1];
28          int maxVal = integers[0];
29          indices[0] = 0;
30          int count = 1;
31          for (int i = 1; i < integers.Length; i++)
32          {
33              if (integers[i] > maxVal)
34              {
35                  maxVal = integers[i];
36                  count = 1;
37                  indices = new int[1];
38                  indices[0] = i;
39              }
40              else
41              {
42                  if (integers[i] == maxVal)
43                  {
44                      count++;
45                      int[] oldIndices = indices;
46                      indices = new int[count];
47                      oldIndices.CopyTo(indices, 0);
48                      indices[count - 1] = i;
49                  }
50              }
51          }
52          Trace.WriteLine(string.Format(
53              "Maximum value {0} found, with {1} occurrences.", maxVal, count));
54          return maxVal;
55      }
```

Breakpoints

New　Columns　Search:

Name	Labels	Condition	Hit Count
Program.cs, line 27 character 10		(no condition)	break always
Program.cs, line 31 character 15		(no condition)	break always
Program.cs, line 33 character 13		(no condition)	break always
Program.cs, line 39 character 13		(no condition)	break always
Program.cs, line 49 character 16		(no condition)	break always
Program.cs, line 54 character 10		(no condition)	break always

Error List　Output　Breakpoints

图　7-5

在调试模式下执行这个应用程序，会得到与前面完全相同的结果。在代码窗口中右击跟踪点，或者利用 Breakpoints 窗口，就可以删除或临时禁用跟踪点。在 Breakpoints 窗口中，跟踪点左边的复选框确定是否启用跟踪点；禁用的跟踪点未被选中，在代码窗口中显示为菱形框，而不是实心菱形。

3. 诊断输出与跟踪点

前面介绍了两种输出相同信息的方法，下面看看它们的优缺点。首先，跟踪点与 Trace 命令并不等价，也就是说，不能使用跟踪点在发布版本中输出信息。这是因为跟踪点并没有包含在应用程序中。跟踪点由 VS 处理，在应用程序的已编译版本中，跟踪点是不存在的。只有应用程序运行在 VS 调试器中时，跟踪点才起作用。

跟踪点的主要缺点也是其优点，即它们存储在 VS 中，因此可以在需要时快速、方便地添加到应用程序中，而且也非常容易删除。如果输出非常复杂的信息字符串，觉得跟踪点非常讨厌，只需单击表示其位置的红色菱形，就可以删除跟踪点。

跟踪点的一个优点是允许方便地添加额外的信息，如上一节提到的$FUNCTION。这个信息可以用 Debug 和 Trace 命令来编写，但比较难。总之，输出调试信息的两种方法是：

- **诊断输出**：总是要从应用程序中输出调试结果时使用这种方法，尤其是在要输出的字符串比较复杂，涉及几个变量或许多信息的情况下，使用该方法比较好。另外，如果要在发布模式下获得执行应用程序的调试结果，Trace 命令常常是唯一的选择。
- **跟踪点**：调试应用程序时，希望快速输出重要信息，以便解决语义错误，应使用跟踪点。

另一个明显的区别是跟踪点只能在 VS 中使用，而诊断输出可以在 VS 和 VCE 中使用。

7.1.2　中断模式下的调试

调试技术的剩余内容是在中断模式下工作。可以通过几种方式进入这种模式，这些方式都可以暂停程序的执行。

1. 进入中断模式

进入中断模式的最简单方式是在运行应用程序时，单击 IDE 中的 Pause 按钮。这个 Pause 按钮在 Debug 工具栏上，应把该工具栏添加到 VS 默认显示的工具栏中。为此，右击工具栏区域，并选择 Debug，这个工具栏如图 7-6 所示。

图　7-6

在这个工具栏上，前 4 个按钮可以手工控制中断。在图 7-6 上，其中的 3 个按钮显示为灰色，因为在程序没有运行时，它们是不能工作的。还有一个按钮 Start 是可以使用的，这个按钮与标准工具栏上的 Start 按钮相同。在后面的章节需要其他的按钮时，再介绍它们。

运行一个应用程序时，工具栏就如图 7-7 所示。

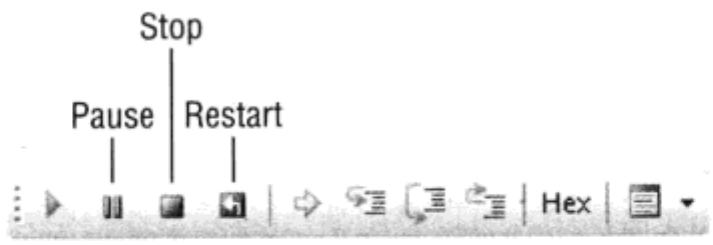


图　7-7

现在，就可以使用之前显示为灰色的 3 个按钮了。它们可以：

- 暂停应用程序的执行，进入中断模式
- 完全停止应用程序的执行(不进入中断模式，而是退出应用程序)
- 重新启动应用程序

暂停应用程序是进入中断模式的最简单方式，但这并不能更好地控制停止程序运行的位置。我们可能会很自然地停止运行应用程序，例如，要求用户输入信息。还可以在长时间的操作或循环过程中进入中断模式，但停止的位置可能相当随机。一般情况下，最好使用断点。

断点

断点是源代码中自动进入中断模式的一个标记，可以在 VS 和 VCE 中使用，但断点在 VS 中更加灵活。它们可以配置为：

- 在遇到断点时，立即进入中断模式
- (只用于 VS)在遇到断点时，如果布尔表达式的值为 true，就进入中断模式
- (只用于 VS)遇到某断点一定的次数后，进入中断模式
- (只用于 VS)在遇到断点时，如果自从上次遇到断点以来变量的值发生了变化，就进入中断模式
- (只用于 VS)把文本输出到 Output 窗口中，或者执行一个宏(参见本章上一节)

注意，上述功能仅能用于调试程序。如果编译发布程序，将会忽略所有断点。

添加断点有几种方法。要添加简单断点，当遇到该断点所在的代码行时，就中断执行，可以单击该代码行左边的灰色区域，右击该代码行，选择 Breakpoint | Insert Breakpoint 菜单项；选择 Debug | Toggle Breakpoint；或者按下 F9 键。

断点在该代码行的旁边显示为一个红色的圆圈，而该行代码也突出显示，如图 7-8 所示。

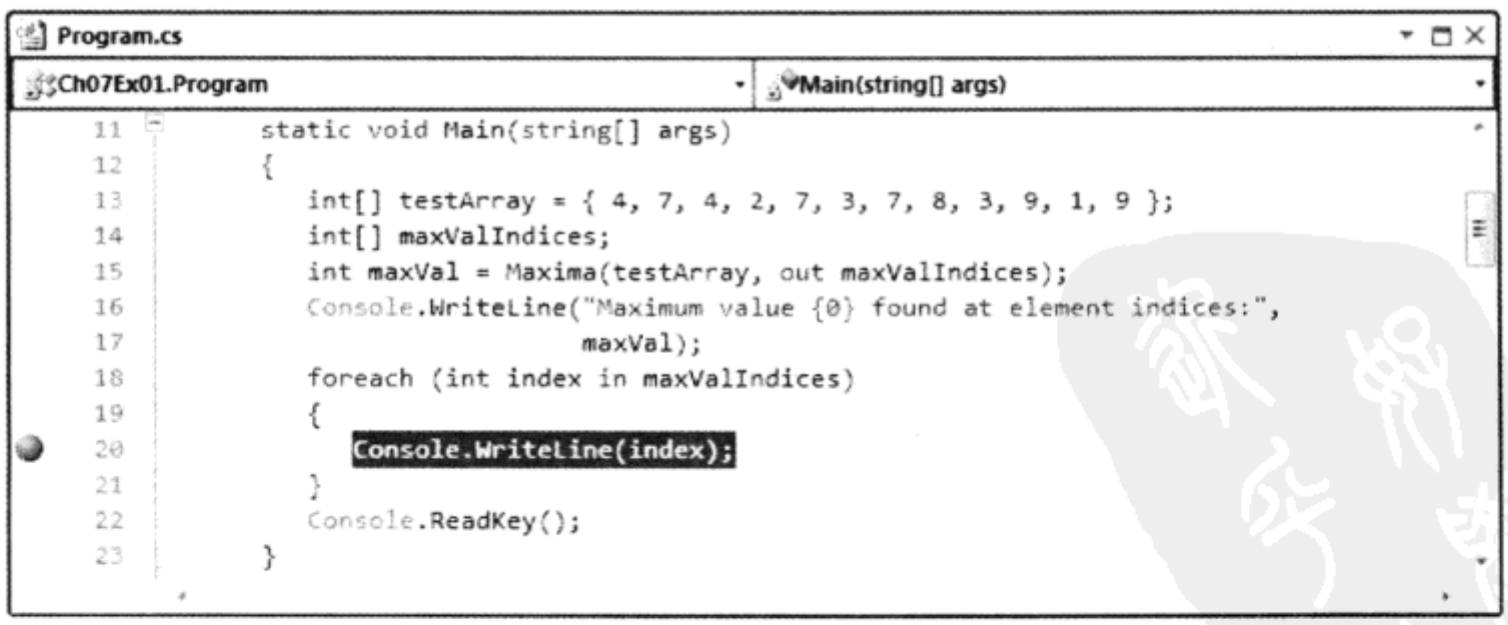


图　7-8

本节的剩余内容仅适用于 VS，不适用于 VCE。如果使用的是 VCE，可以跳到“进入中断模式的其他方式”一节。

在 VS 中，使用 Breakpoints 窗口还可以查看文件中的断点信息(在“跟踪点”一节中介绍过启用该窗口的方法)。在 Breakpoints 窗口中，可以禁用断点(删除描述信息左边的记号；禁用的断点用未填充的红色圆圈来表示)，删除断点，编辑断点的属性。

这个窗口中显示的 Condition 和 Hit Count 列是唯一的两个可用列，它们是非常有用的。右击断点(在代码或这个窗口中)，选择 Condition 或 Hit Count 菜单项，就可以编辑它们。

选择 Condition 按钮，将弹出一个对话框。在该对话框中可以键入任意布尔表达式，该表达式可以包含断点涉及的任何变量。例如，可以配置一个断点，输入表达式 maxVal>4，选择 Is true 选项，则在遇到这个断点，且 maxVal 的值大于 4 时，就会触发该断点。还可以检查这个表达式是否有变化，仅当发生变化时，断点才会被触发(例如，如果在遇到断点时，maxVal 的值从 2 改为 6，就会触发该断点)。

选择 Hit Count 按钮，将弹出一个对话框。在这个对话框中可以指定在触发前，要遇到该断点多少次。下拉列表提供了如下选项：

- 总是中断
- 在 Hit Count 等于多少次时中断
- 在 Hit Count 是某个数的倍数时中断
- 在 Hit Count 大于等于多少次时中断

所选的选项与在旁边的文本框中输入的值共同确定断点的行为。这个 Hit Count 按钮在比较长的循环中很有用，例如，在执行了前 5000 次循环后需要中断。如果不这么做，中断并再重启动 5000 次是很痛苦的。

带有附加属性集(例如，条件或遇到断点的次数)的断点，在显示时略有区别。已配置的断点不是显示一个简单的红色圆圈，而是在红色的圆圈中有一个白色的加号。这是很有用的，因为它允许很快辨认出哪个断点总是进入中断模式，哪个断点只在某种情况下才进入中断模式。

进入中断模式的其他方式

进入中断模式还有两种方式。一种是在抛出一个未处理的异常时选择进入该模式。这种方式在本章后面讨论到错误处理时论述。另一种方式是生成一个判定语句(assertion)时中断。

判定语句是可以用用户定义的消息中断应用程序的指令。它们常常用于应用程序的开发过程，作为测试程序是否能平滑运行的一种方式。例如，在应用程序的某一处要求给定的变量值小于 10，此时就可以使用一个判定语句，确定它是否为 true，如果不是，就中断程序的执行。当遇到判定语句时，可以选择 Abort，中断应用程序的执行，也可以选择 Retry，进入中断模式，还可以选择 Ignore，让应用程序像往常一样继续执行。

与前面的调试输出函数一样，判定函数也有两个版本：

- Debug.Assert()
- Trace.Assert()

其调试版本也是仅用于编译调试程序。

这两个函数带 3 个参数。第一个参数是一个布尔值，其值为 false 会触发判定语句。第二、三个参数是两个字符串，分别把信息写到弹出的对话框和 Output 窗口中。上面的示例需要一个函数调用，如下所示：

```
Debug.Assert(myVar < 10, "myVar is 10 or greater.",
             "Assertion occurred in Main().");
```

判定语句通常在应用程序的早期使用比较有效。可以分发应用程序的一个发布程序，其中包含 Trace.Assert()函数，以列出各种信息。如果触发了判定语句，用户就会收到通知，把这些消息传递给开发人员。这样，即使开发人员不知道错误是如何发生的，也可以改正这个错误。

例如，在第一个字符串中提供错误的简短描述，在第二个字符串中提供下一步该如何操作的指示：

```
Trace.Assert(myVar < 10, "Variable out of bounds.",
             "Please contact vendor with the error code KCW001.");
```

如果触发了这个判定语句，用户就会看到如图 7-9 所示的对话框。

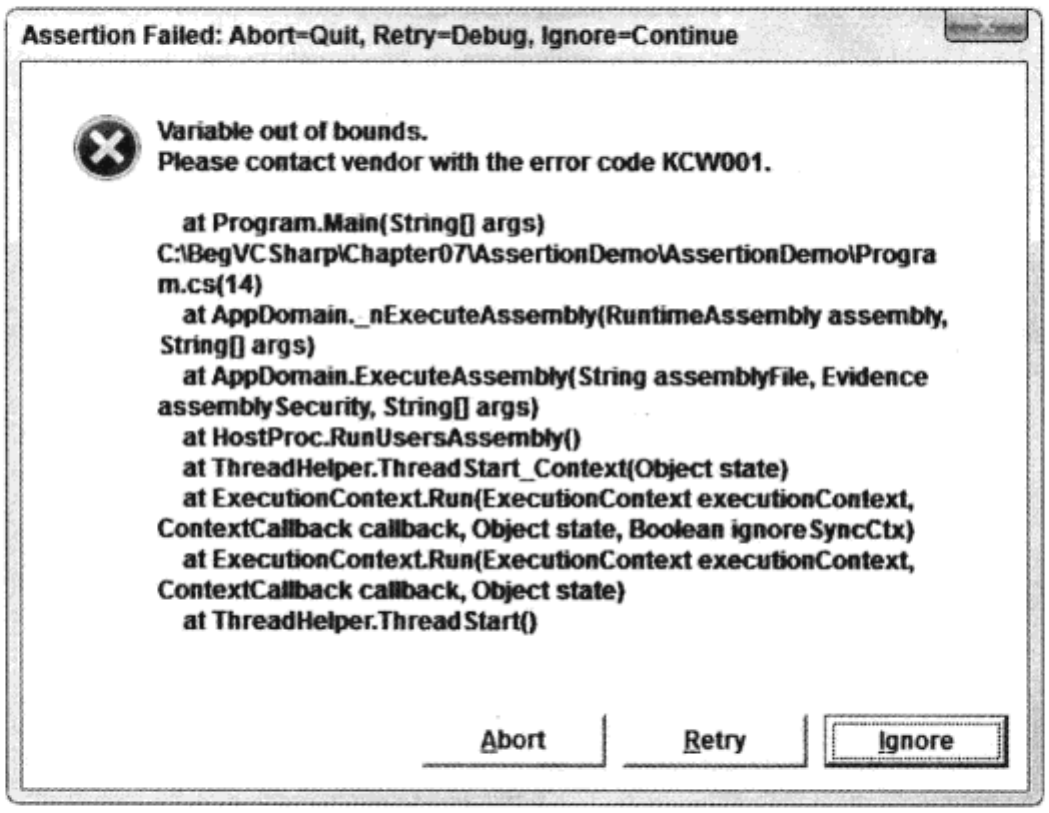


图　7-9

诚然，这并不是最友好的对话框，因为它包含了许多令人迷惑的信息，但如果用户给开发人员发送了错误的屏幕图，开发人员就可以很快找出问题所在。

下一个要论述的主题是应用程序中断，以及进入中断模式后，我们可以做什么。一般情况下，进入中断模式的目的是找出代码中的错误(或确保程序工作正常)。一旦进入中断模式，就可以使用各种技巧分析代码，分析应用程序在暂停处的确切状态。

2. 监视变量的内容

监视变量的内容是 VS 和 VCE 帮助我们使工作变得简单的一个方面。查看变量值的最简单方式是在中断模式下，使鼠标指向源代码中的变量名，此时就会出现一个黄色的工具提示，显示该变量

的信息，其中包括该变量的当前值。

还可以高亮显示整个表达式，以相同的方式得到该表达式的结果。对于比较复杂的值，例如，数组，甚至可以扩展工具提示中的值，查看各个数组元素项。

注意，在运行应用程序时，IDE 中各个窗口的布局发生了变化，在默认情况下，运行期间会发生如下变化(变化的情况会根据具体的安装略有区别)：

- Properties 窗口消失，其他一些窗口也会消失，包括 Solution Explorer 窗口
- Error List 窗口会被屏幕底部的两个新窗口代替
- 新窗口中会出现几个新的选项卡

新的屏幕布局如图 7-10 所示。这可能与读者的显示情况不完全相同，一些选项卡和窗口可能不完全匹配。但是，这些窗口的功能(后面将讨论)是相同的，这个显示完全可以通过 View 和 Debug | Windows 菜单来定制(在中断模式下)，也可以在屏幕上拖动窗口，重新设定它们的位置。

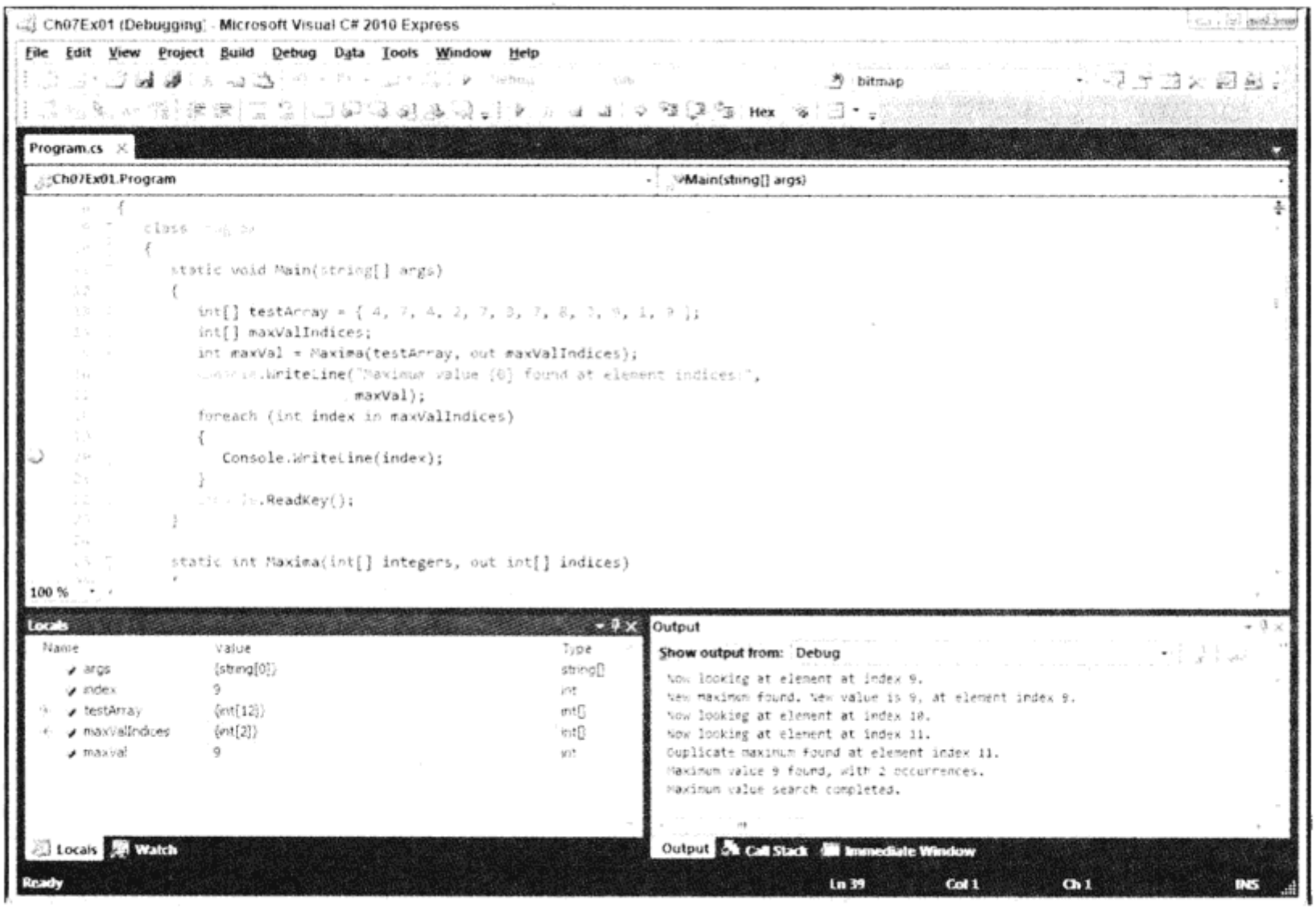


图　7-10

底部左角边的新窗口在调试时非常有用，它允许在中断模式下，在应用程序的变量值上保留标签，它包含 3 个选项卡，如下所示(在 VS 和 VCE 中有所不同)：

- Autos(只在 VS 中有)—— 当前和前面的语句使用的变量(Ctrl+D, A)
- Locals——作用域内的所有变量(Ctrl+D, L)
- Watch N——可定制的变量和表达式显示(其中 N 从 1～4，在 Debug Windows Watch 上)

这些选项卡的工作方式或多或少有些类似，并根据它们的特定功能添加了各种附加特性。一般情况下，每个选项卡都包含一个变量列表，其中包括变量的名称、值和类型等信息。更复杂的变量(如

数组)可以使用变量名左边的＋和－(展开/折叠)符号进一步查看，它们的内容可以以树状视图的方式显示。例如，在前面的示例中，在代码中放置了一个断点，得到的 Locals 选项卡如图 7-11 所示，其中显示了数组变量 maxValIndices 的展开视图。

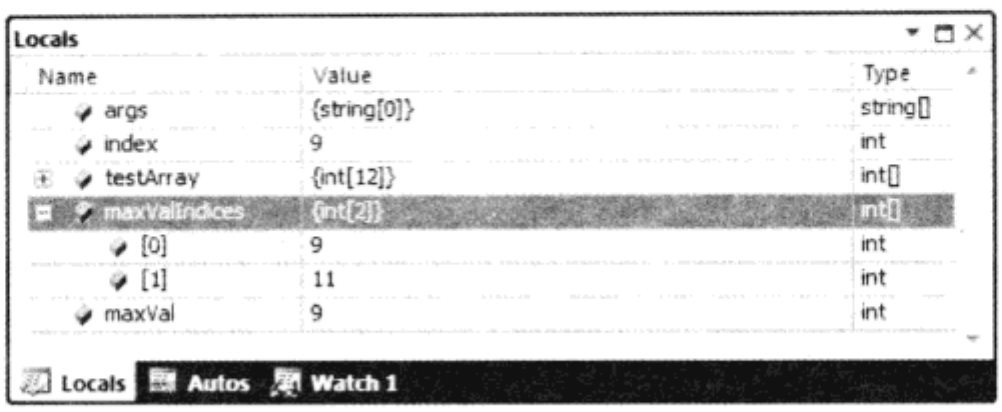


图　7-11

在这个视图中，还可以编辑变量的内容。它有效地绕过了前面代码中的其他变量赋值。为此，只需在 Value 列中为要编辑的变量输入一个新值即可。也可以把这种技巧用于其他情况，例如，需要修改代码才能编辑变量值的情况。

可以通过 Watch 窗口(或 VS 中的 Watch 窗口，至多可以显示 4 个)监视特定变量或涉及特定变量的表达式。要使用这个窗口，只需在 Name 列中键入变量名或表达式，就可以查看它们的结果，注意，并不是应用程序中的所有变量在任何时候都在作用域内，并在 Watch 窗口中对变量做出标记。例如，图 7-12 显示了一个 Watch 窗口，其中包含几个示例变量和表达式，在遇到 Maxima()函数末尾前面的一个断点时，会显示这个 Watch 窗口。

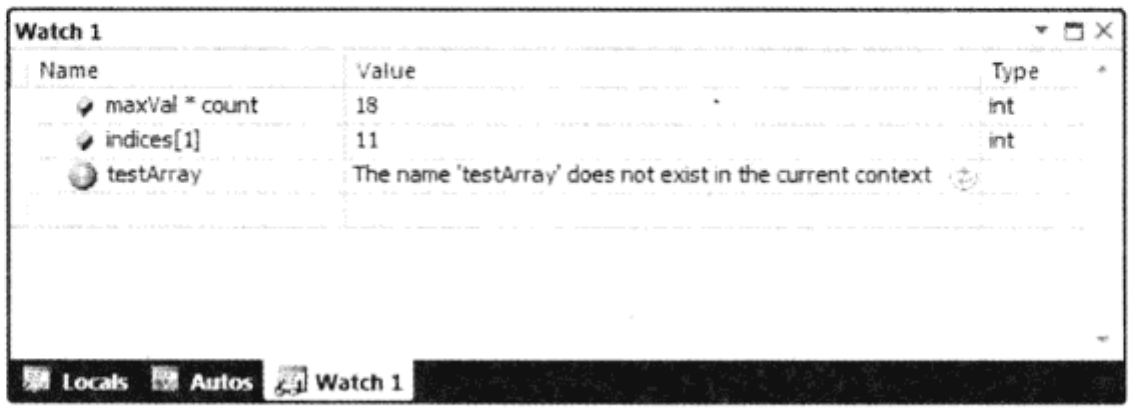


图　7-12

testArray 数组对于 Main()来说是局部数组，所以在该图中没有值，而是显示了一个信息，告诉我们这个变量不在作用域内。

要在 Watch 窗口中添加变量，还可以把变量从源代码拖动到该窗口中。

在这个窗口中可以访问变量的各种显示结果，一个优点是它们可以显示变量在断点之间的变化情况。新值显示为红色而不是黑色，所以很容易看出哪个值发生了变化。

如前所述，要在 VS 中添加更多的 Watch 窗口，可以在中断模式下，使用 Debug | Windows | Watch | Watch N 菜单选项打开或关闭 Watch 的 4 个窗口。每个窗口都可以包含变量和表达式的一组观察结果，所以可以把相关的变量组合在一起，以便于访问。

除了这些 Watch 窗口外，VS 还有一个 QuickWatch 窗口，它能快速提供源代码中某个变量的详细信息。要使用这个窗口，可以右击要查看的变量，选择 QuickWatch 菜单选项。但在大多数情况下，使用标准的 Watch 窗口就足够了。

Watch 窗口可以在应用程序的各个执行过程之间保留下来。如果中断应用程序，再重新运行，就不必再次添加 Watch 窗口了，IDE 会记住上次使用的 Watch 窗口。

3. 单步执行代码

前面介绍了如何在中断模式下查看应用程序的运行情况，下面论述如何在中断模式下使用 IDE 单步执行代码，查看代码的执行结果，人们的思维速度不会比计算机运行得更快，所以这是一个极有价值的技巧。

进入中断模式后，在代码视图的左边，正在执行的代码旁边会出现一个光标(如果使用断点进入中断模式，该光标最初应显示在断点的红色圆圈中)，如图 7-13 所示。

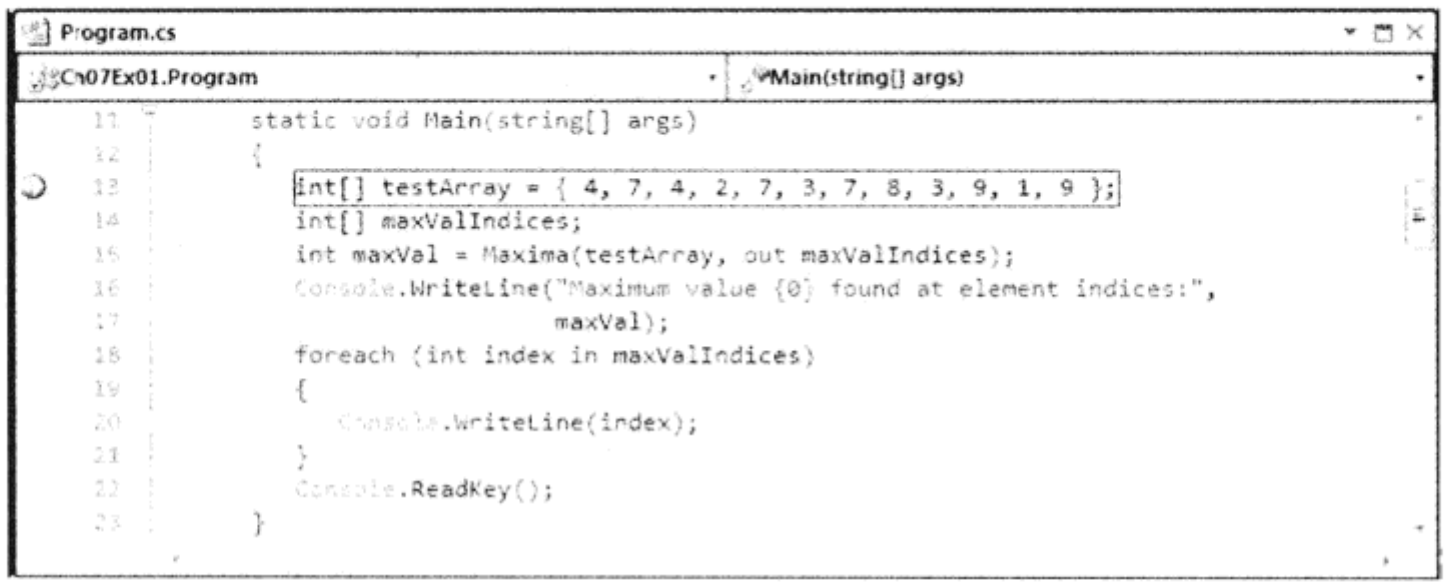


图　7-13

这显示了在进入中断模式时程序执行到的位置。在这个位置上，可以选择逐行执行。为此，使用前面看到的其他一些 Debug 工具栏按钮，如图 7-14 所示。

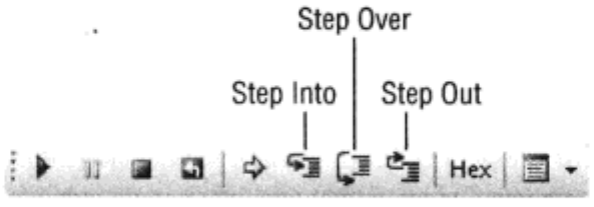


图　7-14

第 6、7、8 个图标控制了中断模式下的程序流。它们依次是：

- Step Into—— 执行并移动到下一个要执行的语句上
- Step Over——同上，但不进入嵌套的代码块，包括函数
- Step Out——执行到代码块的末尾，在执行完该语句块后，重新进入中断模式

如果要查看应用程序执行的每个操作，可以使用 Step Into 按顺序执行指令，这包括在函数中执行，例如，上面示例中的 Maxima()。当光标到达第 15 行，调用 Maxima()时，单击这个图标，会使光标移动到 Maxima()函数内部的第一行代码上。而如果光标移到第 15 行时单击 Step Over，就会使光标移动到第 16 行，不必进入 Maxima()中的代码(但仍执行这段代码)。如果单步执行到不感兴趣的

函数，可以单击 Step Out，返回到调用该函数的代码。在单步执行代码时，变量的值可能会发生变化。注意观察上一节讨论的 Watch 窗口，可以看到变量值的变化情况。

在存在语义错误的代码中，这个技巧也许是最有效的。可以单步执行代码，当执行到有错误的代码时，错误会像正常运行程序那样发生。在这个过程中，可以监视数据，看看什么地方出错。本章后面将使用这个技巧查看示例应用程序的执行情况。

4. Immediate 和 Command 窗口

Command(只有 VS 中有)和 Immediate 窗口(选择 Debug | Windows 菜单)可以在运行应用程序的过程中执行命令。通过 Command 窗口可以手动执行 VS 操作(例如，菜单和工具栏操作)，Immediate 窗口可以执行源代码，计算表达式，还可以执行其他代码。

VS 中的这些窗口在内部是链接在一起的(实际上，VS 的早期版本把它们当作同一个窗口)。甚至可以在它们之间切换：输入命令 immed，可以从 Command 窗口切换到 Immediate 窗口；输入>cmd 可以从 Immediate 窗口切换到 Command 窗口。

下面详细讨论 Immediate 窗口，因为 Command 窗口仅适用于复杂的操作，只能在 VS 中使用，而 Immediate 窗口可以在 VS 和 VCE 中使用。Immediate 窗口最简单的用法是计算表达式，有点像 Watch 窗口中的一次性使用。为此，只需键入一个表达式，并按回车键即可。接着就会显示请求的信息，如图 7-15 所示。

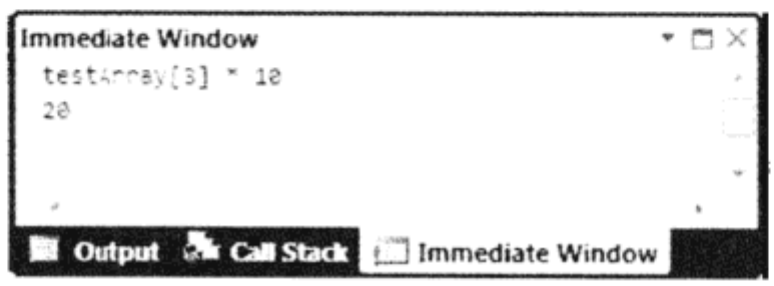


图　7-15

也可以在这里修改变量的内容，如图 7-16 所示。

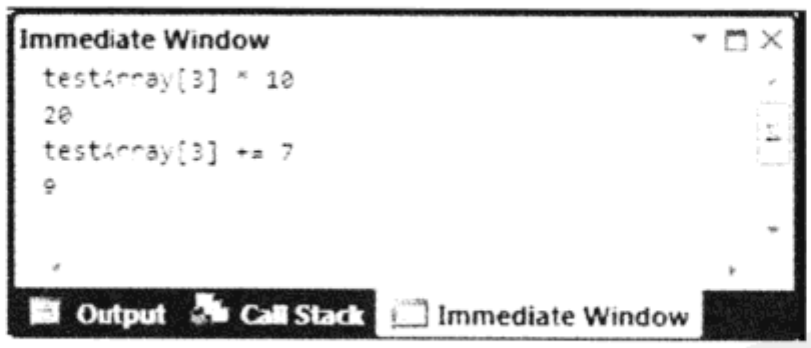


图　7-16

在大多数情况下，使用前面介绍的变量监视窗口更容易得到相同的效果，但这个技巧对于常常发生变化的变量值仍很方便，也适合于测试以后不感兴趣的表达式。

5. Call Stack 窗口

这是最后一个要讨论的窗口，它描述了程序是如何执行到当前位置的。简言之，该窗口显示了当前函数、调用它的函数以及调用函数的函数(即一个嵌套的函数调用列表)。调用的位置也被记录下来。

在前面的示例中，在执行到 Maxima()时进入中断模式，或者使用代码单步执行功能移动到这个函数的内部，得到如图 7-17 所示的信息。

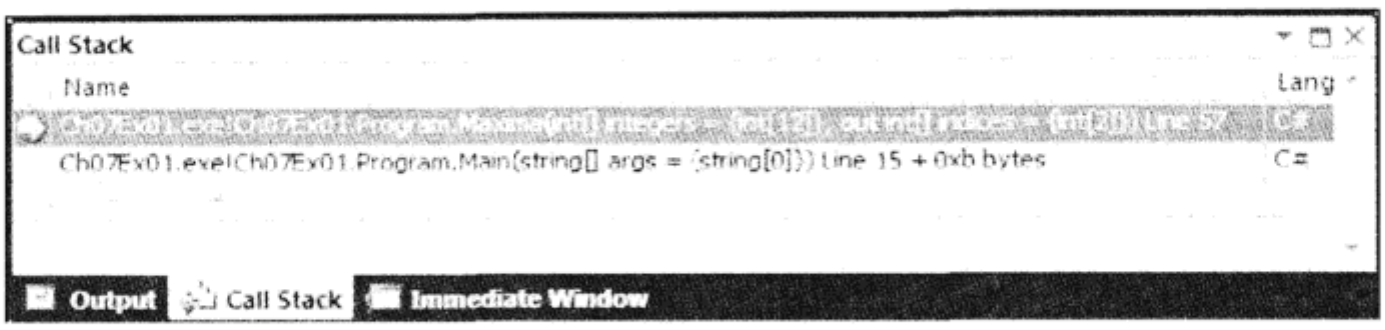


图　7-17

如果双击某一项，就会移动到相应的位置，跟踪代码执行到当前位置的过程。在第一次检测错误时，这个窗口非常有用，因为它们可以查看临近错误发生时的情况。对于常用函数中出现的错误，则有助于找到错误的源头。

有时 Call Stack 窗口会显示一些非常杂乱的信息，例如，有时因为以错误的方式使用了外部函数，错误在应用程序的外部发生，就会出现这种情况。此时，这个窗口中会列出一个很长的列表，其中只有一两个选项是我们熟悉的。如果需要，可以右击该窗口，选择 Show External Code，查看外部引用。

7.2　错误处理

本章的第一部分讨论如何在应用程序的开发过程中查找和改正错误，使这些错误不会在发布的代码中出现。但有时，我们知道可能会有错误发生，但不能100%地肯定它们不会发生。此时，最好能预料到错误的发生，编写足够强壮的代码以处理这些错误，而不必中断程序的执行。

错误处理就是用于这个目的，下面还将介绍异常和处理它们的方式。异常是在运行期间代码中产生的错误，或者由代码调用的函数产生的错误。这里的“错误”定义要比以前更含糊，因为异常可能是在函数等结构中手工产生。例如，如果函数的一个字符串参数不是以 a 开头，就产生一个异常。这并不是严格意义上的函数外部错误，但调用该函数的代码把它看作函数外部错误。

您已在本书前面已经遇到几次异常了。最简单的示例是试图定位一个超出范围的数组元素，例如：

```
int[] myArray = {1, 2, 3, 4};
int myElem = myArray[4];
```

这会产生如下异常信息，并中断应用程序的执行：

```
Index was outside the bounds of the array.
```

前面介绍了异常辅助信息窗口的一些示例。该窗口中的一行把它与出错的代码连接起来，还包含.NET 帮助文件中相关主题的链接和一个 View Detail 链接，利用该链接可以找到所发生异常的更多信息。

异常在命名空间中定义，大多数异常的名称清晰地说明了它们的用途。在这个示例中，产生的异常叫做 System.IndexOutOfRangeException，说明我们提供的 myArray 数组索引不在允许使用的索引范围内。在异常未处理时，这个信息才会显示出来，应用程序也才会中断执行。下一节将讨论如何处理异常。

7.2.1　try...catch...finally

C#语言包含结构化异常处理(Structured Exception Handling，SEH)的语法。用 3 个关键字可以标记出能处理异常的代码和指令，如果发生异常，就使用这些指令处理异常。用于这个目的的 3 个关键字是 try、catch 和 finally。它们都有一个关联的代码块，必须在连续的代码行中使用。其基本结构如下：

```
try
{
   ...
}
catch (<exceptionType> e)
{
   ...
}
finally
{
   ...
}
```

也可以只有 try 块和 finally 块，而没有 catch 块，或者有一个 try 块和好几个 catch 块。如果有一个或多个 catch 块，finally 块就是可选的，否则就是必需的。这些代码块的用法如下：

- **try**——包含抛出异常的代码(在谈到异常时，其中的“抛出”在 C#语言中也可以是“生成”或“导致”)。
- **catch**——包含抛出异常时要执行的代码。catch 块可以使用<*exceptionType*>，设置为只响应特定的异常类型(如 System.IndexOutOfRangeException)，以便提供多个 catch 块。还可以完全省略这个参数，让一般的 catch 块响应所有异常。
- **finally**——包含总是会执行的代码，如果没有产生异常，则在 try 块之后执行，如果处理了异常，就在 catch 块后执行，或者在未处理的异常上移到调用堆栈之前执行。“上移到调用堆栈”表示，SEH 允许嵌套 try...catch...finally 块，可以直接嵌套，也可以在 try 块包含的函数调用中嵌套。例如，如果在被调用的函数中没有 catch 块能处理某个异常，就由调用代码中的 catch 块处理。如果始终没有匹配的 catch 块，就终止应用程序。finally 块在此之前处理，是因为存在这个块，否则也可以在 try...catch...finally 结构的外部放置代码。这个嵌套功能将在后面的“异常处理的注意事项”一节中进一步讨论，所以如果对这个功能感到迷惑，请不必担心。

在 try 块的代码中出现异常后，发生的事件依次是：

- try 块在发生异常的地方中断程序的执行。
- 如果有 catch 块，就检查该块是否匹配已抛出的异常类型。如果没有 catch 块，就执行 finally 块(如果没有 catch 块，就一定要有 finally 块)。
- 如果有 catch 块，但它与已发生的异常类型不匹配，就检查是否有其他 catch 块。
- 如果有 catch 块匹配已发生的异常类型，就执行它包含的代码，再执行 finally 块(如果有)。

- 如果 catch 块都不匹配已发生的异常类型，就执行 finally 块(如果有)。

下面用一个“试一试”示例来说明异常的处理。这个示例以几种方式抛出和处理异常，以便读者了解其工作情况。

试一试：异常处理

(1) 在 C:\BegVCSharp\Chapter07 目录中创建一个新控制台应用程序 Ch07Ex02。

(2) 修改代码，如下所示(这里显示的行号注释有助于将代码与后面的讨论相匹配，在本章的可下载代码中复制了它们，以便使用)：


```
class Program
{
   static string[] eTypes = {"none", "simple", "index", "nested index"};

   static void Main(string[] args)
   {
      foreach (string eType in eTypes)
      {
         try
         {
            Console.WriteLine("Main() try block reached.");        // Line 23
            Console.WriteLine("ThrowException(\"{0}\") called.", eType);
                                                                   // Line 24
            ThrowException(eType);
            Console.WriteLine("Main() try block continues.");      // Line 26
         }
         catch (System.IndexOutOfRangeException e)                 // Line 28
         {
            Console.WriteLine("Main() System.IndexOutOfRangeException catch"
                              + " block reached. Message:\n\"{0}\"",
                              e.Message);
         }
         catch                                                     // Line 34
         {
            Console.WriteLine("Main() general catch block reached.");
         }
         finally
         {
            Console.WriteLine("Main() finally block reached.");
         }
         Console.WriteLine();
      }
      Console.ReadKey();
   }

   static void ThrowException(string exceptionType)
   {
                                                                   // Line 49
      Console.WriteLine("ThrowException(\"{0}\") reached.", exceptionType);
      switch (exceptionType)
      {
         case "none" :
            Console.WriteLine("Not throwing an exception.");
```

```
            break;                                                    // Line 54
         case "simple" :
            Console.WriteLine("Throwing System.Exception.");
            throw (new System.Exception());                           // Line 57
            break;
         case "index" :
            Console.WriteLine("Throwing System.IndexOutOfRangeException.");
          eTypes[4] = "error";                                        // Line 60
            break;
         case "nested index" :
            try                                                       // Line 63
            {
               Console.WriteLine("ThrowException(\"nested index\") " +
                                 "try block reached.");
               Console.WriteLine("ThrowException(\"index\") called.");
               ThrowException("index");                               // Line 68
            }
            catch                                                     // Line 70
            {
               Console.WriteLine("ThrowException(\"nested index\") general"
                                 + " catch block reached.");
            }
            finally
            {
               Console.WriteLine("ThrowException(\"nested index\") finally"
                                 + " block reached.");
            }
            break;
      }
   }
}
```

代码段 Ch07Ex02\Program.cs

(3) 运行应用程序，结果如图 7-18 所示。

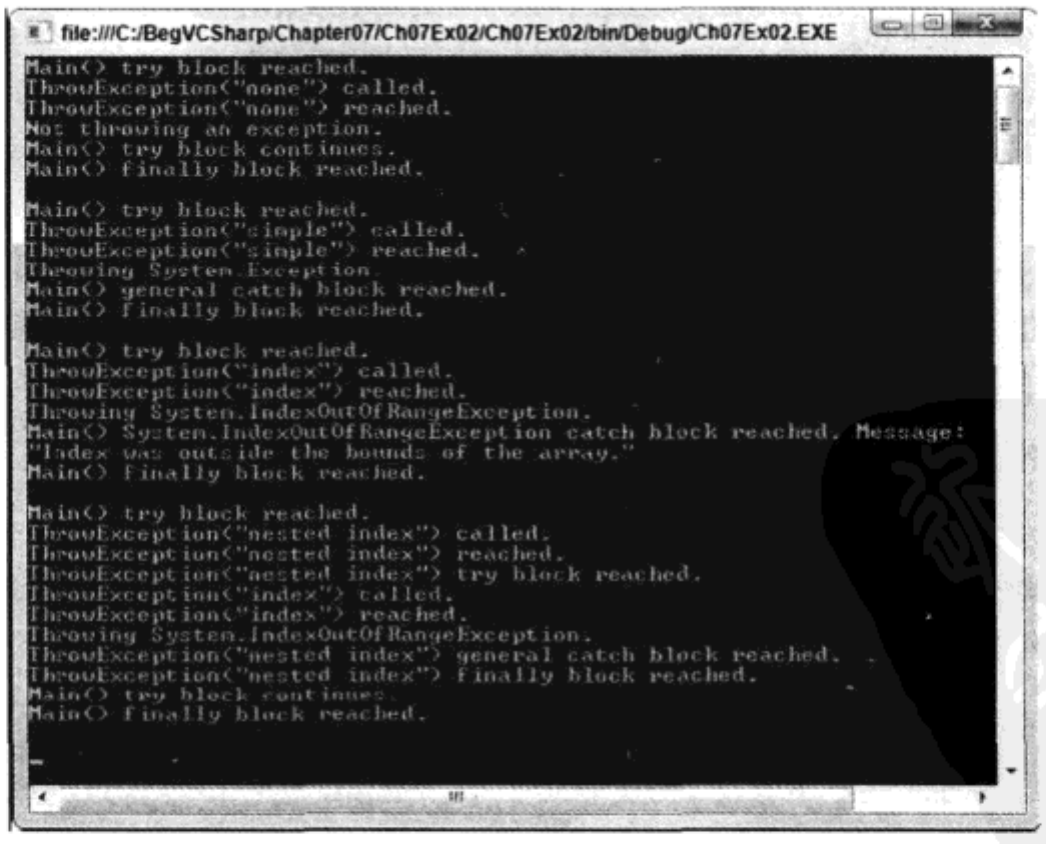


图　7-18

示例的说明

这个应用程序在 Main()中有一个 try 块，它调用函数 ThrowException()。这个函数会根据调用时使用的参数抛出异常：

- ThrowException("none")——不抛出异常。
- ThrowException("simple")——生成一般异常。
- ThrowException("index")——生成 System.IndexOutOfRangeException 异常。
- ThrowException("nested index")——包含它自己的try 块，其中的代码调用 ThrowException("index")，生成一个 System.IndexOutOfRangeException 异常。

其中的每个 string 参数都存储在全局数组 eTypes 中，在 Main()函数中迭代，用每个可能的参数调用 ThrowException()。在迭代过程中，会把各种信息写到控制台上，说明发生了什么情况。这段代码可以使用本章前面介绍的代码单步执行技巧。在执行代码的过程中，一次执行一行代码可以确切地了解代码的执行进度。

在代码的第 23 行添加一个新断点(用默认的属性)，该行代码如下：

```
Console.WriteLine("Main() try block reached.");
```

这里使用了行号，因为它们显示在这段代码的下载版本中。如果关闭了行号，可以在 Text Editor | C# | General 选项区域中，选择 Tools | Options 菜单选项打开它们。上述代码包含注释，这样读者可以阅读文本，而无需打开文件。

在调试模式下运行应用程序。程序立即进入中断模式，此时光标停在第 23 行上。如果选择变量监视窗口中的 Locals 选项卡，就会看到 eType 当前是 none。使用 Step Into 按钮处理第 23 和 24 行，看看第一行文本是否已经写到控制台上。接着使用 Step Into 按钮单步执行第 25 行的 ThrowException()函数。

执行到 ThrowException()函数(第 49 行)后，Locals 窗口会发生变化。eType 和 args 不再能访问(因为它们是 Main()的局部变量)，我们看到的是 exceptionType 局部参数，它当然是 none。继续单击 Step Into，到达 switch 语句，检查 exceptionType 的值，执行代码，把字符串 Not throwing an exception 写到屏幕上。在执行第 54 行上的 break 语句时，将退出函数，继续处理 Main()中的第 26 行代码。因为没有抛出异常，所以继续执行 try 块。

接着处理 finally 块。再单击 Step Into 几次，执行完 finally 块和 foreach 的第一次循环。下次执行到第 25 行时，使用另一个参数 simple 调用 ThrowException()。

继续使用 Step Into 单步执行 ThrowException()，最终会执行到第 57 行：

```
throw (new System.Exception());
```

这里使用C# throw 关键字生成一个异常，需要为这个关键字提供新初始化的异常作为其参数，抛出一个异常，这里使用名称空间 System 中的另一个异常 System.Exception。

在这个 case 块中不需要 break 语句，使用 throw 就可以结束该块的执行。

在使用 Step Into 执行这个语句时，将从第 34 行开始执行一般的 catch 块。因为与第 28 行开始的 catch 块都不匹配，所以执行这个一般的 catch 块。单步执行这段代码，然后执行 finally 块，最后返回另一个循环周期，该循环在第 25 行用一个新参数调用 ThrowException()，这次的参数是 index。

这次 ThrowException()在第 60 行生成一个异常：

```
eTypes[4] = "error";
```

eTypes 是一个全局数组，所以可以在这里访问它。但是这里试图访问数组中的第5 个元素(其索引从 0 开始计数)，这会生成一个 System.IndexOutOfRangeException 异常。

这次 Main()中有一个匹配的 catch 块，单步执行该 catch 块，从第 28 行开始。这个块中调用的 Console.WriteLine()使用 e.Message，输出存储在异常中的信息(可以通过 catch 块的参数访问异常)。之后再次单步执行 finally 块(而不是第二个 catch 块，因为异常已经处理完了)。返回循环，再次调用第 25 行的 ThrowException()。

在执行到 ThrowException()中的 switch 结构时，进入一个新的 try 块，从第 63 行开始。在执行到第 68 行时，将遇到 ThrowException()的一个嵌套调用，这次使用 index 参数。可以使用 Step Over 按钮跳过其中的代码行，因为前面已经单步执行过了。与前面一样，这个调用生成一个 System.IndexOutOfRangeException 异常。但这个异常在 ThrowException()中的嵌套 try...catch...finally 结构中处理。这个结构没有明确匹配这种异常的 catch 块，所以执行一般的 catch 块(从第 70 行开始)。

与前面的异常处理一样，现在单步执行这个 catch 块，以及关联的 finally 块，最后返回到函数调用的末尾。但是它们有一个重大的区别：抛出的异常是由 ThrowException()中的代码处理的。这就是说，异常并没有留给 Main()处理，所以直接进入 finally 块，之后应用程序中断执行。

7.2.2　列出和配置异常

.NET Framework 包含许多异常类型，可以在代码中自由抛出和处理这些类型的异常，甚至可以在代码中抛出异常，让它们在比较复杂的应用程序中被捕获。IDE 提供了一个对话框，可以检查和编辑可用的异常，可以使用 Debug | Exceptions 菜单选项(或按下 Ctrl+D，E)打开该对话框，如图 7-19 所示(如果使用 VCE，则列表中的项会不同，只包含图 7-19 中的第二、三项)。

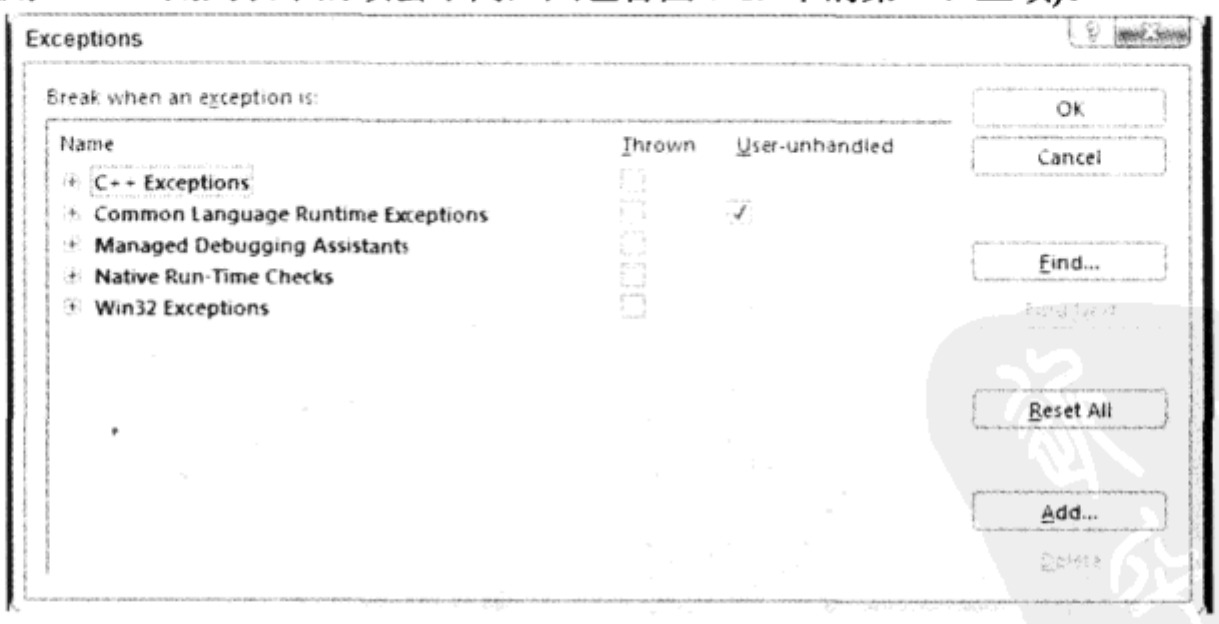


图　7-19

按照类别和.NET 库名称空间列出异常。展开 Common Language Runtime Exceptions 选项，再展开 System 选项，就可以看到 System 名称空间中的异常，这个列表包括上面使用的异常

System.IndexOutOfRangeException。

每个异常都可以使用右边的复选框来配置。可以使用第一个选项(break when)Thrown 中断调试器，即使是对于已处理的异常，也是这样。第二个选项可以忽略未处理的异常，这样做会对结果有影响。在大多数情况下，这会进入中断模式，所以只需在异常环境下这么做。

在大多数情况下，使用默认设置就足够了。

7.2.3　异常处理的注意事项

注意，必须在更一般的异常捕获之前为比较特殊的异常提供 catch 块。如果 catch 块的顺序错误，应用程序就会编译失败。还要注意可以在 catch 块中抛出异常，方法是使用上一个示例中的方式，或使用下述表达式：

```
throw;
```

这个表达式会再次抛出 catch 块处理过的异常。如果以这种方式抛出异常，该异常就不会由当前的 try...catch...finally 块处理，而是由上一级的代码处理(但嵌套结构中的 finally 块仍会执行)。

例如，如果修改 ThrowException()中的 try...catch...finally 块，如下所示：

```
try
 {
    Console.WriteLine("ThrowException(\"nested index\") " +
                      "try block reached.");
    Console.WriteLine("ThrowException(\"index\") called.");
    ThrowException("index");
 }
 catch
 {
    Console.WriteLine("ThrowException(\"nested index\") general"
       + " catch block reached.");
    throw;
 }
 finally
 {
    Console.WriteLine("ThrowException(\"nested index\") finally"
       + " block reached.");
 }
```

则首先执行其中的 finally 块，再执行 Main()中匹配的 catch 块，得到的控制台输出如图 7-20 所示。

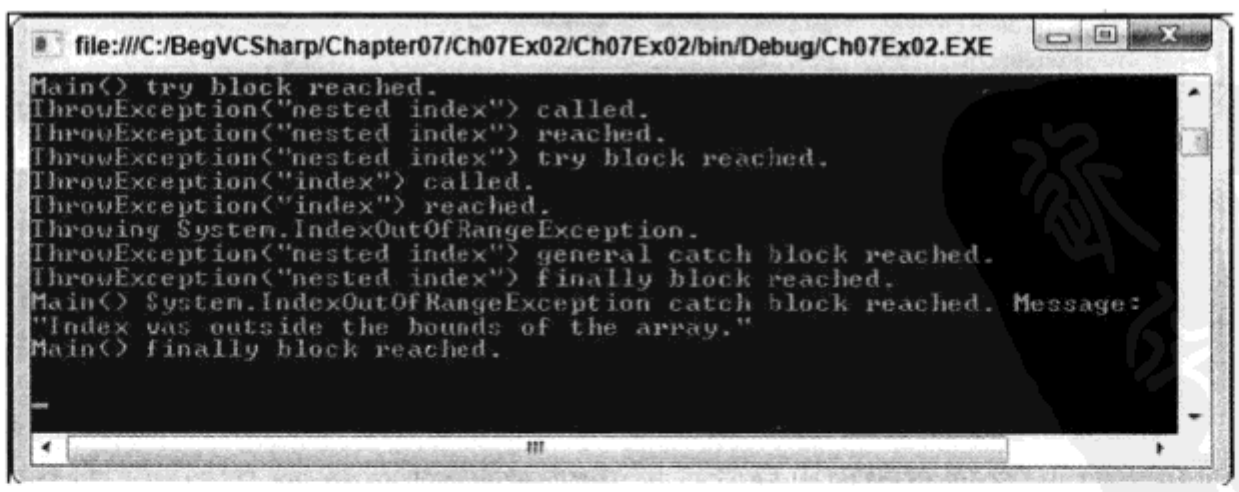


图　7-20

在这个屏幕图中，Main()函数输出了额外的几行，因为这个函数捕获了 System.IndexOutOfRangeException 异常。

7.3　小结

本章主要论述了调试应用程序所使用的技巧。有许多这方面的技巧，其中的大部分可用于各类项目，而不仅仅是控制台应用程序。

前面介绍了创建简单控制台应用程序的所有内容，以及调试它们的各种方法。本书的下一部分将讨论强大的面向对象编程技术。

7.4　练习

(1) “使用 Trace.WriteLine() 要优于使用 Debug.WriteLine()，因为调试版本仅能用于调试程序。”这个观点正确吗？为什么？

(2) 为一个简单的应用程序编写代码，其中包含一个循环，该循环在运行 5000 次后产生一个错误。使用断点在第 5000 次循环出现错误前进入中断模式(注意产生错误的一种简单方式是试图访问一个不存在的数组元素，例如在一个有 100 个元素的数组中，访问 myArray[1000])。

(3) “只有在不执行 catch 块的情况下，才执行 finally 代码块”，对吗？

(4) 下面定义了一个枚举数据类型 orientation。编写一个应用程序，使用结构化异常处理(SEH)把 byte 类型的变量安全地强制转换为 orientation 类型变量。注意，可以使用 checked 关键字强制抛出异常，下面是一个示例。可以在应用程序中使用这段代码：

```
enum orientation : byte
{
   north = 1,
   south = 2,
   east  = 3,
   west  = 4
}
myDirection = checked((orientation)myByte);
```

附录 A 给出了练习答案。

7.5　本章要点

主　题	重 要 概 念
错误类型	在编译期间和运行期间，致命错误(语法错误)都会使应用程序完全失败，语义错误或逻辑错误比较微妙，可能会使应用程序执行不正确，或者以未预料到的方式执行
输出调试信息	我们可以编写代码，把有帮助的信息输出到 Output 窗口中，以帮助在 IDE 中进行调试。为此需要使用 Debug 和 Trace 系列函数，其中 Debug 函数在发布版本中会被忽略。对于投入生产的应用程序，应把调试输出写入日志文件。在 VS 中，还可以使用跟踪点输出调试信息

(续表)

主　题	重 要 概 念
中断模式	可以通过断点、判定语句，或者在发生未处理的异常时，手工进入中断模式(暂停应用程序的状态)。可以在代码的任意位置添加断点，在 VS 中，还可以把断点配置为仅在特定条件下中断执行。在中断模式下，可以检查变量的内容(使用各种调试信息窗口)，每次执行一行代码，以帮助确定哪里出现了错误
异常	异常是运行期间发生的错误，可以通过编程方式捕获和处理这种错误，以防应用程序终止。调用函数或处理变量时，可能会发生许多不同类型的异常。还可以使用 throw 关键字生成异常
异常处理	代码中未处理的异常会使应用程序终止。使用 try、catch 和 finally 代码块处理异常。Try 块标记了一个启用异常处理的代码段，catch 块包含的代码仅在异常发生时执行，它可以匹配特定类型的异常，还可以包含多个 catch 块。Finally 块指定异常处理完毕后执行的代码，如果没有发生异常，Finally 块就指定在 try 块执行完毕后执行的代码。只能包含一个 Finally 块，如果包含了 catch 块，Finally 块就是可选的

第 8 章

面向对象编程简介

本章的主要内容：

- 什么是面向对象编程
- OOP 技术
- Windows Forms 应用程序对 OOP 的依赖关系

本书前面介绍了 C#语法和编程的所有基础知识，以及调试应用程序的方法。现在我们已经可以编写出能使用的控制台应用程序了。但是，要了解 C#语言和.NET Framework 的强大功能，还需要使用面向对象编程(Object-Oriented Programming，OOP)技术。实际上，前面已经使用了这些技术，但为了使学习任务简单一些，在列出代码示例时没有重点讲述该技术。

本章先不考虑代码，而主要探讨 OOP 的原理。OOP 会很快把我们领回 C#语言，因为它与 OOP 是一种共生关系。本章介绍的所有概念在后面的章节中都会再次讨论，并用演示性的代码来说明。所以，如果您在第一次阅读本章时没有掌握所有的内容，不必惊慌。

首先介绍 OOP 的基础知识，包括回答最基本的问题“什么是对象？”。很快您就会发现有许多术语与 OOP 有关，这些术语最初很容易混淆，但本章提供了大量的解释。使用 OOP 需要以另一种方式来看待编程。

除了讨论 OOP 的一般原理外，本章还将进入一个对于全面理解 OOP 非常重要的领域：Windows Forms 应用程序。此类应用程序(它们使用 Windows 环境和诸如菜单、按钮等特性)有许多描述性的区域，在 Windows Forms 环境中可以有效地说明 OOP 要点。

本章中的 OOP 实际上是.NET OOP，这里讲述的一些技术不能应用于其他 OOP 环境。在编写 C#程序时，使用的是.NET 特有的 OOP，因此专注于这些方面是明智之举。

8.1　面向对象编程的含义

面向对象编程是创建计算机应用程序的一种相当新的方法，它解决了传统编程技巧带来的许多问题。前面介绍的编程方法称为函数(或过程)化编程，常常会导致所谓的单一应用程序，即所有的功能都包含在几个代码模块中(常常是一个代码模块)。而使用OOP 技术，常常要使用许多代码模块，每个模块都提供特定的功能，每个模块都是孤立的，甚至与其他模块完全独立。这种模块化编程方法提供了非常大的多样性，大大增加了重用代码的机会。

要进一步说明这个问题，假定计算机上的一个高性能应用程序是一辆一流赛车。如果使用传统的编程技巧，这辆赛车就是一个单元。如果要改进该车，就必须替换整车，把它送回厂商那里，让汽车专家升级它，或者购买一辆新车。如果使用OOP 技术，就只需从厂商处购买新的引擎，自己按照其说明替换它，而不必用钢锯切割车体。

在传统的应用程序中，执行流常常是简单的、线性的。把应用程序加载到内存中，从 A 点开始执行，在 B 点结束，然后从内存中卸载，在这个过程中可能用到其他各种实体，例如存储介质上的文件或显卡的功能，但处理的主体总是位于一个地方。此时的代码一般与使用各种数学和逻辑方式处理数据相关。处理方法通常比较简单，使用基本的数据类型，例如整型和布尔值，建立比较复杂的数据表达方式。

而使用OOP，事情就不是这么直接了。尽管可以获得相同的效果，但其实现方式是完全不同的。OOP 技术以结构、数据的含义以及数据和数据之间的交互操作为基础。这通常意味着要把更多的精力放在项目的设计阶段，但项目的可扩展性比较高。一旦对某种类型的数据的表达方式达成一致，这种表达方式就会应用到应用程序以后的版本中，甚至是全新的应用程序中。这种一致的表达方式可以大大减少开发时间。这就是上述赛车示例的工作原理。这里的一致是“引擎”的代码是结构化的，这样就可以很容易地替换成新代码(即新引擎)，而不需要找厂商帮忙。这也表示，引擎创建出来后可以用于其他目的，可以把它安装到另一辆车上，或者用它驱动潜艇。

除了数据表达方式的一致性外，OOP 编程还常常可以简化任务，因为较抽象实体的结构和用法也是一致的。例如，不仅把输出结果发送给设备(如打印机)所使用的数据格式是一致的，而且与该设备交换数据的方法也是一致的，这包括它理解的指令等等。回到赛车的示例上，要达成的一致包括引擎如何连接到油箱上，如何把驱动力传送给车轮等。

顾名思义，OOP 技术要使用对象。

8.1.1　对象的含义

对象就是OOP 应用程序的一个组成部件。这个组成部件封装了部分应用程序，这部分程序可以是一个过程、一些数据或一些更抽象的实体。

简单地说，对象非常类似于本书前面讨论的结构类型，包含变量成员和函数类型。它所包含的变量组成了存储在对象中的数据，其中包含的函数可以访问对象的功能。略为复杂的对象可能不包含任何数据，而只包含函数，表示一个过程。例如，可以使用表示打印机的对象，其中的函数可以控制打印机(允许打印文档、测试页等)。

C#中的对象是从类型中创建的，就像前面的变量一样。对象的类型在 OOP 中有一个特殊的名称：类。可以使用类的定义实例化对象，这表示创建该类的一个实例。“类的实例”和对象含义相同，注

意“类”和“对象”是完全不同的概念。

术语“类”和“对象”常常混淆，从一开始就正确区分它们是非常重要的，使用前面的赛车示例有助于区分这两个术语。在这个示例中，类是指汽车的模板，或者用于构建汽车的规划。汽车本身是这些规划的实例，所以可以看作对象。

本章将使用统一建模语言(Unified Modeling Language，UML)语法研究类和对象。UML 是为应用程序建模而设计的，从组成应用程序的对象，到它们执行的操作，到我们希望有的用例，应有尽有。这里只使用这个语言的基本部分，在使用它们的过程中进行解释，但不考虑比较复杂的部分，因为 UML 是一个很专业的主题，所以需要整本书的篇幅来讨论。

VS 有一个类查看器，它是一个功能很强大的工具，可用于以类似的方式显示类。但为了简单起见，本章的图是手绘的。

图 8-1 是打印机类 Printer 的 UML 表示方法。类名显示在这个框的顶部(后面将论述下面两个区域)。图 8-2 是这个 Printer 类的一个实例 myPrinter。

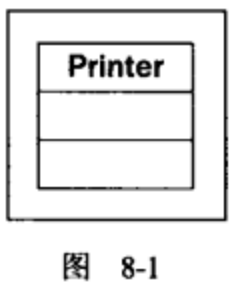


图　8-1

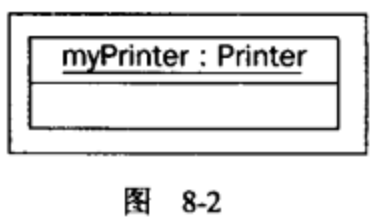


图　8-2

在顶部，实例名显示在前面，后面是类名。这两个名称用一个冒号分隔。

1. 属性和字段

可以通过属性和字段访问对象中包含的数据。这个对象数据可以用于区分不同的对象，因为同一个类的不同对象在属性和字段中存储了不同的值。

包含在对象中的不同数据构成了对象的状态。假定一个对象类表示一杯咖啡，叫作 CupOfCoffee。在实例化这个类(即创建这个类的对象)时，必须提供对类有意义的状态。此时可以使用属性和字段，让代码能通过该对象设置要使用的咖啡品牌，咖啡中是否加牛奶或方糖，咖啡是否即溶等。于是，给定的这杯咖啡对象就有了指定的状态，例如，加牛奶和两块方糖的哥伦比亚滴滤咖啡。

字段和属性都可以键入，所以可以把信息存储在字段和属性中，作为 string 值、int 值等。但是，属性与字段是不同的，因为属性不提供对数据的直接访问。对象能让用户不考虑数据的细节，不需要在属性中用一对一的方式表示。如果在 CupOfCoffee 实例中使用一个字段表示方糖的数量，用户就可以在该字段中放置自己喜欢的值，其取值范围仅由存储该信息的类型来限制。例如，如果使用 int 来存储这个数据，用户就可以使用-2 147 483 648～2 147 483 647 之间的任意值，如第 3 章所述。显然，并不是所有的值都有意义，尤其是负值，一些较大的正值将需要非常大的咖啡杯。但如果使

用一个属性来表示，就可以限制这个值，例如为 0~2 之间的一个数字。

一般情况下，在访问状态时最好提供属性，而不是字段，因为这样可以更好地控制各种行为，这个选择不会影响使用对象实例的代码，因为使用属性和字段的语法是相同的。

对属性的读写访问也可以由对象来明确定义。某些属性是只读的，只能查看它们的值，而不能改变它们(至少不能直接改变)。这常常是同时读取几个状态的一个有效技巧。CupOfCoffee 类有一个只读属性 Description，在请求它时，就返回一个字符串，表示该类的一个实例的状态(例如前面给出的字符串)。也可以通过查看几个属性，把相同的数据组合起来，但这样的属性可以节省时间和精力。还可以有只写的属性，其操作方式是类似的。

除了对属性的读/写访问外，还可以为字段和属性指定另一种访问权限，称为可访问性。这种可访问性确定了什么代码可以访问这些成员，它们是可用于所有的代码(公共)，还是只能用于类中的代码(私有)，或者更复杂的模式(详见本章后面的内容)。常见的情况是把字段设置为私有，通过公共属性访问它们。这样，类中的代码就可以直接访问存储在字段中的数据，而公共属性禁止外部用户访问这些数据，以防他们在其中放置无效的内容。公共成员是类可以访问的成员。

要更清晰地阐明这个问题，可以把可访问性与变量的作用域等同起来。例如，私有字段和属性可以看作是拥有它们的对象的局部成员，而公共字段和属性的作用域也包括对象以外的代码。

在类的 UML 表示方法中，用第二部分显示属性和字段，如图 8-3 所示。

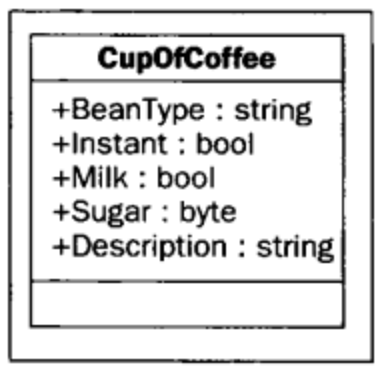


图 8-3

这是 CupOfCoffee 类的表示方式，前面为它定义了 5 个成员(属性或字段，在 UML 中，它们没有区别)。每个成员都包含下述信息：

- 可访问性：＋号表示公共成员，－号表示私有成员。但一般情况下，本章的图中不显示私有成员，因为这些信息是类内部的信息。至于读/写访问，则不提供任何信息。
- 成员名。
- 成员的类型。

冒号用于分隔成员名和类型。

2. 方法

“方法”这个术语用于表示对象中的函数。这些函数调用的方式与其他函数相同，使用返回值和参数的方式也相同(详见第 6 章)。

方法用于提供访问对象的功能。与字段和属性一样，方法也可以是公共的或私有的，按照需要限制外部代码的访问。它们常常使用对象状态影响它们的操作，在需要时访问私有成员，如私有字段。例如，CupOfCoffee 类定义了一个方法 AddSugar()，该方法对递增方糖数提供了比设置相应的 Sugar 属性更易读的语法。

在 UML 的对象框中，方法显示在第三部分，如图 8-4 所示。

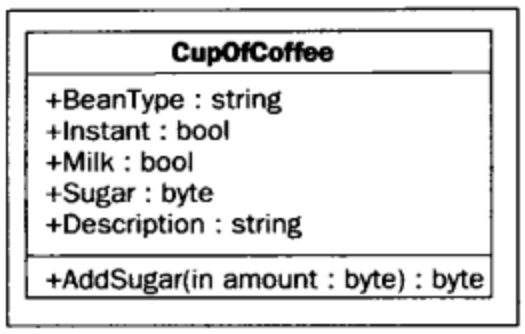


图　8-4

其语法类似于字段和属性，但最后显示的类型是返回类型，在这一部分，还显示了方法的参数。在 UML 中，每个参数都带有下述标识符之一：in、out 或 inout。它们用于表示数据流的方向，其中 out 和 inout 大致对应于第 6 章讨论的 C#关键字 out 和 ref。in 大致对应于 C#中不使用这两个关键字的情形。

8.1.2　一切皆对象

本书一直在使用对象、属性和方法。实际上，C#和.NET Framework 中的所有东西都是对象。控制台应用程序中的 Main()函数就是类的一个方法。前面介绍的每个变量类型都是一个类。前面使用的每个命令都是一个属性或方法，例如，<String>.Length 和 <String>.ToUpper()等。句点字符把对象实例名和属性或方法名分隔开来，方法名后面的()把方法与属性区分开来。

对象无处不在，使用它们的语法通常比较简单，这使我们可以集中精力讨论 C#中一些比较基础的方面。从现在开始详细介绍对象。这里讨论的概念都具有深远的影响。它们甚至可以应用到简单的 int 变量上。

8.1.3　对象的生命周期

每个对象都有一个明确定义的生命周期，除了“正在使用”的正常状态之外，还有两个重要的阶段：

- **构造阶段**：对象最初进行实例化的时期。这个初始化过程称为构造阶段，由构造函数完成。
- **析构阶段**：在删除一个对象时，常常需要执行一些清理工作，例如，释放内存，这由析构函数完成。

1. 构造函数

对象的初始化过程是自动完成的。我们不需要找一个适于存储新对象的内存空间。但是，在初始化对象的过程中，有时需要执行一些额外的工作。例如，需要初始化对象存储的数据。构造函数就是用于初始化数据的函数。

所有的类定义都至少包含一个构造函数。在这些构造函数中，可能有一个默认的构造函数，该函数没有参数，与类同名。类定义还可能包含几个带有参数的构造函数，称为非默认的构造函数。代码可以使用它们以许多方式实例化对象，例如给存储在对象中的数据提供初始值。

在 C#中，用 new 关键字来调用构造函数。例如，可以用下面的方式通过其默认的构造函数实例化一个 CupOfCoffee 对象：

```
CupOfCoffee myCup = new CupOfCoffee();
```

还可以用非默认的构造函数来创建对象。例如，CupOfCoffee 类有一个非默认的构造函数，它使用一个参数在初始化时设置咖啡豆的品牌：

```
CupOfCoffee myCup = new CupOfCoffee("Blue Mountain");
```

构造函数与字段、属性和方法一样，可以是公共或私有的。在类外部的代码不能使用私有构造函数实例化对象，而必须使用公共构造函数。这样，就可以要求类的用户使用非默认的构造函数(把默认构造函数设置为私有的)。

一些类没有公共的构造函数，外部的代码就不可能实例化它们，这些类称为不可创建的类，但如稍后所述，这些类并不是完全没有用的。

2. 析构函数

.NET Framework 使用析构函数清理对象。一般情况下，不需要提供析构函数的代码，而是由默认的析构函数自动执行操作。但是，如果在删除对象实例前，需要完成一些重要的操作，就应提供特定的析构函数。

例如，如果变量超出了范围，代码就不能访问它，但该变量仍存在于计算机内存的某个地方。只有在.NET 运行程序执行其垃圾回收，进行清理时，该实例才被彻底删除。

不应依赖析构函数释放对象实例使用的资源，因为在不再使用某个对象后，该资源会长时间被该对象占用。如果所使用的资源非常重要，这样做就有可能出问题。有一个解决方法，参阅本章后面的“可删除对象”一节。

8.1.4　静态和实例类成员

属性、方法和字段等成员是对象实例所特有的，此外，还有静态成员(也称为共享成员，尤其是 Visual Basic 用户常常使用这个术语)，例如静态方法、静态属性或静态字段。静态成员可以在类的实例之间共享，所以可以将它们看作是类的全局对象。静态属性和静态字段可以访问独立于任何对象实例的数据，静态方法可以执行与对象类型相关、但与对象实例无关的命令。在使用静态成员时，甚至不需要实例化对象。

例如，前面使用的 Console.WriteLine()和 Convert.ToString()方法就是静态的，根本不需要实例化 Console 或Convert 类(如果试着进行这样的实例化，操作会失败，因为这些类的构造函数不是可公共访问的，如前所述)。

许多情况下，静态属性和方法有很好的效果。例如，可以使用静态属性跟踪给类创建了多少个实例。在 UML 语法中，类的静态成员用下划线表示，如图 8-5 所示。

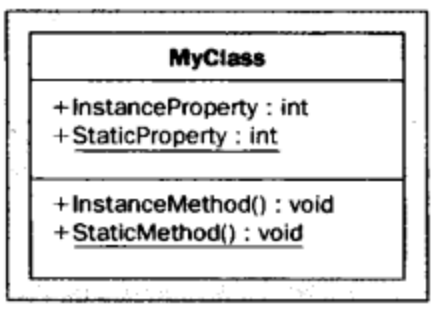


图　8-5

1. 静态构造函数

使用类中的静态成员时，需要预先初始化这些成员。在声明时，可以给静态成员提供一个初始值，但有时需要执行更复杂的初始化，或者在赋值、执行静态方法之前执行某些操作。

使用静态构造函数可以执行此类初始化任务。一个类只能有一个静态构造函数，该构造函数不能有访问修饰符，也不能带任何参数。静态构造函数不能直接调用，只能在下述情况下执行：

- 创建包含静态构造函数的类实例时
- 访问包含静态构造函数的类的静态成员时

在这两种情况下，会先调用静态构造函数，之后实例化类或访问静态成员。无论创建了多少个类实例，其静态构造函数都只调用一次。为了区分静态构造函数和本章前面介绍的构造函数，也将所有非静态构造函数称作实例构造函数。

2. 静态类

我们常常希望类只包含静态成员，且不能用于实例化对象(如Console)。为此，一种简单的方法是使用静态类，而不是把类的构造函数设置为私有。静态类只能包含静态成员，不需要实例构造函数，因为按照定义，它根本不能实例化。但静态类可以有一个静态构造函数，如上一节所述。

如果以前完全没有接触过 OOP，在阅读本章的其他内容之前，应该停下来将 OOP 研究一番。在学习更复杂的 OOP 内容之前，全面掌握基础知识是很重要的。

8.2　OOP 技术

前面介绍了一些基础知识，知道对象是什么，以及对象的工作原理，下面讨论对象的其他一些特性，包括：

- 接口
- 继承
- 多态性
- 对象之间的关系
- 运算符重载
- 事件
- 引用类型和值类型

8.2.1　接口

接口是把公共实例(非静态)方法和属性组合起来，以封装特定功能的一个集合。一旦定义了接口，就可以在类中实现它。这样，类就可以支持接口所指定的所有属性和成员。

注意，接口不能单独存在。不能像实例化一个类那样实例化接口。另外，接口不能包含实现其成员的任何代码，而只能定义成员本身。实现过程必须在实现接口的类中完成。

在前面的咖啡示例中，可以把通用属性和方法例如 AddSugar()、Milk、Sugar 和 Instant 组合到

一个接口中，这个接口称为 IHotDrink(接口的名称一般用大写字母 I 开头)。然后就可以在其他对象上使用该接口，例如CupOfTea 类的对象。所以可以用类似的方式处理这些对象，而对象仍保有自己的属性(例如 CupOfCoffee 仍有属性 BeanType，CupOfTea 仍有属性 LeafType)。

在UML 中，在对象上实现的接口用“棒棒糖”语法来表示。在图 8-6 中，用与类相似的语法把 IHotDrink 的成员放在一个单独的框中。

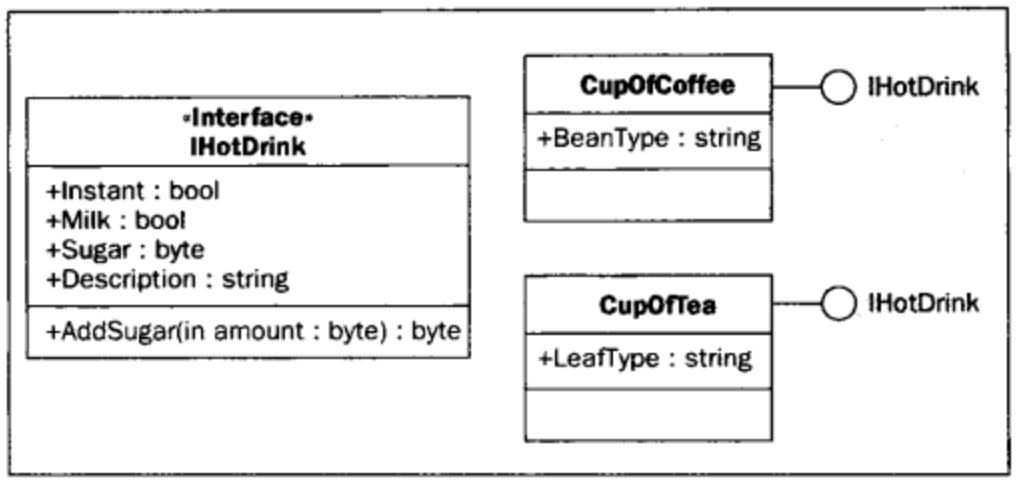


图　8-6

一个类可以支持多个接口，多个类也可以支持相同的接口。所以接口的概念让用户和其他开发人员更容易理解其他人的代码。例如，有一些代码使用一个带某接口的对象。假定不使用这个对象的其他属性和方法，就可以用另一个对象代替这个对象(例如，使用上述 IHotDrink 接口的代码可以处理 CupOfCoffee 和 CupOfTea 实例)。另外，该对象的开发人员可以提供该对象的更新版本，只要它支持已经在用的接口，就可以在代码中使用这个新版本。

在发布接口后，即接口可以用于其他开发人员或终端用户后，最好不要修改它。理解这一点的一种方式是把接口看作类的创建者和使用者之间的契约。“每个支持接口 X 的类都支持这些方法和属性”是有效的。如果以后修改了接口，也许是升级了底层的代码，该接口的使用者就不能正确运行接口，甚至失败。我们应创建一个新的接口，来扩展旧接口，例如包含一个版本号，如 X2。这是创建接口的标准方式，以后我们会常常遇到编了号的接口。

可删除的对象

IDisposable 接口特别有趣。支持IDisposable 接口的对象必须实现其Dispose()方法，即它们必须提供这个方法的代码。当不再需要某个对象(例如，在对象超出作用域之前)时，就调用这个方法，释放重要的资源，否则，该资源会等到对垃圾回收调用析构方法时才释放。这样可以更好地控制对象所使用的资源。

C#允许使用一种可以优化使用这个方法的结构。using 关键字可以在代码块中初始化使用重要资源的对象，会在这个代码块的末尾自动调用 Dispose()方法，用法如下：

```
<ClassName> <VariableName> = new <ClassName>()

...

using (<VariableName>)
{
    ...
}
```

或者把初始化对象<VariableName>作为 using 语句的一部分：

```
using (<ClassName> <VariableName> = new <ClassName>())
{
    ...
}
```

在这两种情况下，可以在 using 代码块中使用变量<VariableName>，并在代码块的末尾自动删除(在代码块执行完毕后，调用 Dispose())。

8.2.2　继承

继承是OOP 最重要的特性之一。任何类都可以从另一个类中继承，这就是说，这个类拥有它继承的类的所有成员。在 OOP 中，被继承(也称为派生)的类称为父类(也称为基类)。注意，C#中的对象仅能直接派生于一个基类，当然基类也可以有自己的基类。

继承性可以从一个较一般的基类扩展或创建更多的特定类。例如，考虑一个代表农场家畜的类(80 多岁的一流开发人员 Old MacDonald 在他的家畜应用程序中使用)。这个类叫作 Animal，拥有 EatFood()或 Breed()等方法，我们可以创建一个派生类 Cow，支持所有这些方法，它也有自己的方法，如 Moo()和 SupplyMilk()。还可以创建另一个派生类 Chicken，该类有Cluck()和 LayEgg()方法。

在 UML 中，用箭头表示继承，如图 8-7 所示。

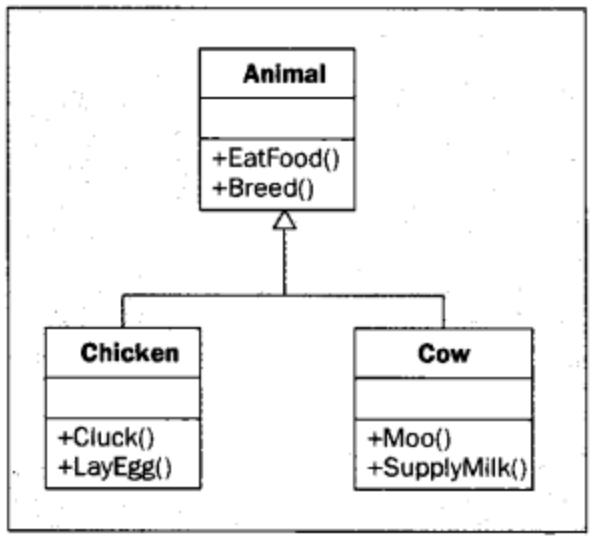


图　8-7

为了简洁起见，图 8-7 中省略了成员的返回类型。

在继承一个基类时，成员的可访问性就成了一个重要的问题。派生类不能访问基类的私有成员，但可以访问其公共成员。不过，派生类和外部的代码都可以访问公共成员。这就是说，只使用这两个可访问性，不能让一个成员可由基类和派生类访问，而不能由外部的代码访问。

为了解决这个问题，C#提供了第三种可访问性：protected，只有派生类才能访问 protected 成员。对于外部代码来说，这个可访问性与私有成员一样：外部代码不能访问 private 成员和 protected 成员。

除了定义成员的保护级别外，我们还可以为成员定义其继承行为。基类的成员可以是虚拟的，也就是说，成员可以由继承它的类重写。派生类可以提供成员的其他实现代码。这种实现代码不会删除原来的代码，仍可以在类中访问原来的代码，但外部代码不能访问它们。如果没有提供其他实现方式，通过派生类使用成员的外部代码就自动访问基类中成员的实现代码。

虚拟成员不能是私有成员，因为这样会自相矛盾——不能说成员可以由派生类重写，同时派生类又不能访问它。

在前面的家畜示例中，可以把 EatFood()变成虚拟成员，在派生类中为它提供新的实现代码，例如为 Cow 类提供新实现代码，如图 8-8 所示。这里显示了Animal 和 Cow 类的 EatFood()方法，说明它们有自己的实现代码。

基类还可以定义为抽象类。抽象类不能直接实例化。要使用抽象类，必须继承这个类，抽象类可以有抽象成员，这些成员在基类中没有实现代码，这些实现代码必须在派生类中提供。如果 Animal 是一个抽象类，UML 就会如图 8-9 所示。

抽象类名以斜体显示(有时它们的方框有一个短横线)。

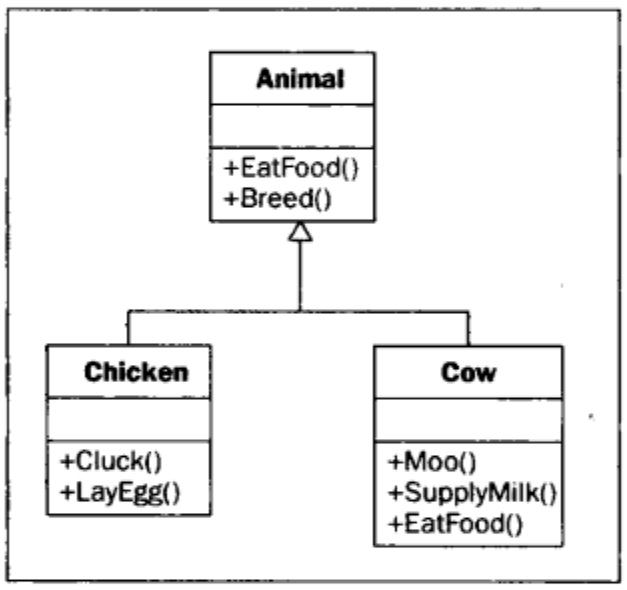


图　8-8

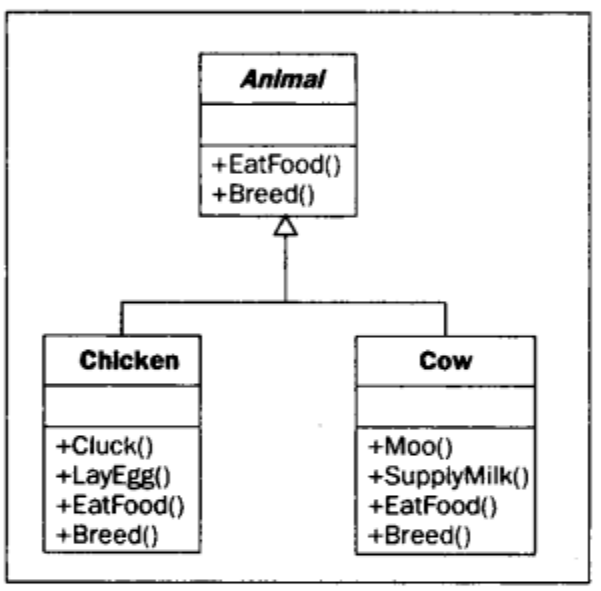


图　8-9

在图 8-9 中，EatFood()和 Breed()都显示在派生类 Chicken 和 Cow 中，这说明这些方法是抽象的(必须在派生类中重写)或者虚拟的(已经在 Chicken 和 Cow 中重写)。当然，抽象基类可以提供成员的实现代码，这是很常见的。不能实例化抽象类，并不意味着不能在抽象类中封装功能。

最后，类可以是密封(seal)的。密封的类不能用作基类，所以没有派生类。

在 C#中，所有的对象都有一个共同的基类 object(在.NET Framework 中，它是 System.Object 类的别名)。第 9 章将详细介绍这个类。

如本章前面所述，接口也可以继承自其他接口。与类不同的是，接口可以继承多个基接口(与类可以支持多个接口的方式类似)。

8.2.3　多态性

继承的一个结果是派生于基类的类在方法和属性上有一定的重叠，因此，可以使用相同的语法处理从同一个基类实例化的对象。例如，如果基类 Animal 有一个 EatFood()方法，则从派生于它的类 Cow 和Chicken 中调用这个方法，其语法是类似的：

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
myCow.EatFood();
myChicken.EatFood();
```

多态性则更推进了一步。可以把某个派生类型的变量赋给基本类型的变量，例如：

```
Animal myAnimal = myCow;
```

不需要进行强制类型转换，就可以通过这个变量调用基类的方法：

```
myAnimal.EatFood();
```

结果是调用派生类中的 EatFood()的实现代码。注意，不能以相同的方式调用派生类上定义的方法，下面的代码不能运行：

```
myAnimal.Moo();
```

但是，可以把基本类型的变量转换为派生类变量，调用派生类的方法，如下所示：

```
Cow myNewCow = (Cow)myAnimal;
myNewCow.Moo();
```

如果原始变量的类型不是Cow 或派生于Cow 的类型，这个强制类型转换就会引发一个异常。有许多方式说明对象的类型是什么，详见下一章。

在派生于同一个类的不同对象上执行任务时，多态性是一种极有效的技巧，其使用的代码最少。注意并不是只有共享同一个父类的类才能利用多态性。只要子类和孙子类在继承层次结构中有一个相同的类，它们就可以用同样的方式利用多态性。

还要注意，在 C#中，所有的类都派生于同一个类 object，object 是继承层次结构中的根。所以可以把所有对象看作是 object 类的实例。这就是在建立字符串时，Console.WriteLine()可以处理无数多种参数组合的原因。第一个参数后面的每个参数都可以看作是一个 object 实例，所以可以把任何对象的输出结果写到屏幕上。为此，需要调用方法 ToString()(object 的一个成员)，我们可以重写这个方法，为自己的类提供合适的实现代码，或者使用默认实现代码，返回类名(根据它所在的名称空间，返回类的修饰名)。

接口的多态性

尽管不能像对象那样实例化接口，但可以建立接口类型的变量，然后就可以在支持该接口的对象上，使用这个变量访问该接口提供的方法和属性。

例如，假定不使用基类 Animal 提供 EatFood()方法，而是把该方法放在 IConsume 接口上。Cow 和Chicken 类也支持这个接口，唯一的区别是它们必须提供 EatFood()方法的实现代码(因为接口不包含实现代码)，接着就可以使用下述代码访问该方法了：

```
Cow myCow = new Cow();
Chicken myChicken = new Chicken();
IConsume consumeInterface;
consumeInterface = myCow;
consumeInterface.EatFood();
consumeInterface = myChicken;
consumeInterface.EatFood();
```

这就提供了以相同方式访问多个对象的简单方式，且不依赖于一个公共的基类。例如，这个接口可以由派生于 Vegetable 的 VenusFlyTrap 类实现，而不是由 Animal 实现：

```
VenusFlyTrap myVenusFlyTrap = new VenusFlyTrap();
IConsume consumeInterface;
consumeInterface = myVenusFlyTrap;
consumeInterface.EatFood();
```

在这段代码中，调用 consumeInterface.EatFood()的结果是调用 Cow、Chicken 或 VenusFlyTrap 类的 EatFood()方法，这取决于哪个实例被赋予了接口类型的变量。

注意，派生类会继承其基类支持的接口。在上面的第一个示例中，要么是 Animal 支持 IConsume，要么是 Cow 和 Chicken 支持 IConsume。有共同基类的类不一定有共同的接口，反之亦然。

8.2.4　对象之间的关系

继承是对象之间的一种简单关系，可以让派生类完整地获得基类的特性，而派生类也可以访问基类内部的一些工作代码(通过受保护的成员)。对象之间还有其他一些重要关系。

本节简要讨论下述关系：

- **包含关系**：一个类包含另一个类。这类似于继承关系，但包含类可以控制对被包含类的成员的访问，甚至在使用被包含类的成员前进行其他处理。
- **集合关系**：一个类用作另一个类的多个实例的容器。这类似于对象数组，但集合有其他功能，包括索引、排序和重新设置大小等。

1. 包含关系

用一个成员字段包含对象实例，就可以实现包含(containment)关系。这个成员字段可以是公共字段，此时与继承关系一样，容器对象的用户就可以访问它的方法和属性，但不能像继承关系那样，通过派生类访问类的内部代码。

另外，可以让被包含的成员对象变成私有成员。如果这么做，用户就不能直接访问任何成员，即使这些成员是公共的，也不能访问。但可以使用包含类的成员访问这些私有成员。也就是说，可以完全控制被包含的类有什么成员，如果有成员，还可以在访问被包含类的成员前，在包含类的成员上进行其他处理。

例如，Cow 类包含一个 Udder 类，它有一个公共方法 Milk()。Cow 对象可以按照要求调用这个方法，作为其 SupplyMilk()方法的一部分，但 Cow 对象的用户看不到这些细节。

在 UML 中，被包含类可以用关联线条来表示。对于简单的包含关系，可以用带有 1 的线条说明一对一的关系(一个 Cow 实例包含一个 Udder 实例)。为清晰起见，也可以把被包含的 Udder 类实例表示为 Cow 类的私有字段，如图 8-10 所示。

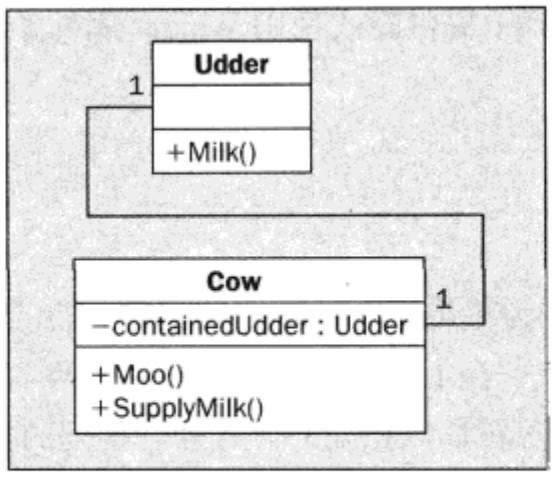


图　8-10

2. 集合关系

第 5 章讨论了如何使用数组存储多个同类变量。这也适用于对象(前面使用的变量类型实际上是对象)。例如：

```
Animal[] animals = new Animal[5];
```

集合基本上是数组，集合以与其他对象相同的方式实现为类。它们通常以所存储的对象名称的复数形式来命名，例如类 Animals 就包含 Animal 对象的一个集合。

数组与集合的主要区别是，集合通常实现额外的功能，例如 Add()和 Remove()方法可添加和删除集合中的项。而集合通常有一个 Item 属性，它根据对象的索引返回该对象。不仅如此，这个属性还允许以更复杂的访问方式来实现。例如，可以设计一个 Animals，让 Animal 对象根据其名称来访问。

在 UML 中，这用图 8-11 来表示。

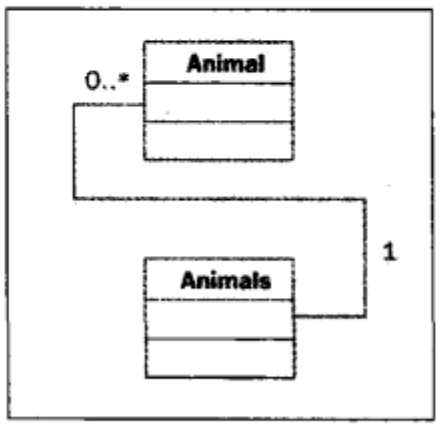


图　8-11

这里遗漏了成员，因为这里描述的是关系。连接线末尾的数字表示一个 Animals 对象可以包含 0 个或多个 Animal 对象。第 11 章将详细论述集合。

8.2.5　运算符重载

本书前面介绍了如何使用运算符处理简单的变量类型。有时也可以把运算符用于从类实例化而来的对象，因为类可以包含如何处理运算符的指令。

例如，给 Animal 添加一个新属性 Weight。接着使用下述代码比较家畜的体重：

```
if (cowA.Weight > cowB.Weight)
{
   ...
}
```

使用运算符重载，可以在代码中提供隐式使用 Weight 属性的逻辑，如下面的代码所示：

```
if (cowA > cowB)
{
   ...
}
```

大于运算符>被重载了。我们为重载运算符编写代码，执行上述操作，这段代码用作类定义的一部分，而该运算符作用于这个类。在上面的示例中，使用了两个 Cow 对象，所以运算符重载定义包含在 Cow 类中。也可以重载运算符，以相同的方式处理不同的类，其中一个(或两个)类定义包含达到这一目的的代码。

注意，只能采用这种方式重载现有的 C#运算符，不能创建新的运算符。但是，可以为一元和二元运算符(如+)提供实现代码。详见第 13 章。

8.2.6　事件

对象可以激活事件，作为它们处理的一部分。事件是非常重要的，可以在代码的其他部分起作用，类似于异常(但功能更强大)。例如，可以在把 Animal 对象添加到 Animals 集合中时，执行特定的代码，而这部分代码不是 Animals 类的一部分，也不是调用 Add()方法的代码的一部分。为此，需要给代码添加事件处理程序，这是一种特殊类型的函数，在事件发生时调用。还需要配置这个处理程序，以监听自己感兴趣的事件。

使用事件可以创建事件驱动的应用程序，这类应用程序比读者此时所能想到的多得多。例如，许多基于 Windows 的应用程序完全依赖于事件。每个按钮单击或滚动条拖动操作都是通过事件处理实现的，其中事件是通过鼠标或键盘触发的。

本章的后面将介绍在 Windows 应用程序中事件的工作原理，第 13 章将深入讨论事件。

8.2.7　引用类型和值类型

在 C#中，数据根据变量的类型以两种方式中的一种存储在一个变量中。变量的类型分为两种：引用类型和值类型，其区别如下：

- 值类型在内存的一个地方存储它们自己和它们的内容。
- 引用类型存储指向内存中其他某个位置(称为堆)的引用，而在另一个位置存储内容。

实际上，在使用 C#时，不必过多地考虑这个问题。到目前为止，所使用的 string 变量(这是引用类型)与使用其他简单变量(大多数是值类型，例如 int)的方式完全相同。

值类型和引用类型的一个主要区别是：值类型总是包含一个值，而引用类型可以是 null，表示它们不包含值。但是，可以使用可空类型(这是泛型的一种形式)创建一个值类型，使值类型在这个方面的行为方式类似于引用类型(即可以为 null)。这是一个高级论题，详见第 12 章。

只有 string 和 object 简单类型是引用类型，但数组也是隐式的引用类型。我们创建的每个类都是引用类型，这就是在这里说明这一点的原因。

结构类型和类的重要区别是，结构类型是值类型。您可能认为结构类型和类非常类似，特别是第 6 章介绍了如何在结构类型上使用函数。第 9 章还将进一步探讨这个问题。

8.3　Windows 应用程序中的 OOP

第 2 章介绍了如何在 C#中创建简单的 Windows 应用程序。Windows 应用程序非常依赖 OOP 技术，本节将论述 OOP 技术，说明本章的一些论点。下面通过一个简单示例加以说明。

试一试：使用对象

(1) 在 C:\BegVCSharp\Chapter08 目录中创建一个新的 Windows 应用程序 Ch08Ex0l。

(2) 使用 Toolbox 添加一个新的按钮控件，使之位于 Form1 的中央，如图 8-12 所示。

(3) 双击按钮，为鼠标单击事件添加代码，修改代码，如下所示：


```
private void button1_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!";
   Button newButton = new Button();
   newButton.Text = "New Button!";
   newButton.Click += new EventHandler(newButton_Click);
   Controls.Add(newButton);
}
private void newButton_Click(object sender, System.EventArgs e)
{
   ((Button)sender).Text = "Clicked!!";
}
```

代码段 Ch08Ex01\Form1.cs

(4) 运行应用程序，窗体如图 8-13 所示。

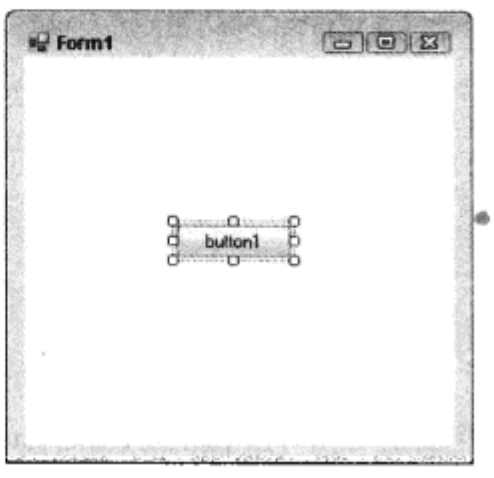


图　8-12

图　8-13

(5) 单击标记为 button1 的按钮，显示内容将随之变化，如图 8-14 所示。

(6) 单击标记为 New Button!的按钮，显示内容将随之变化，如图 8-15 所示。

图　8-14

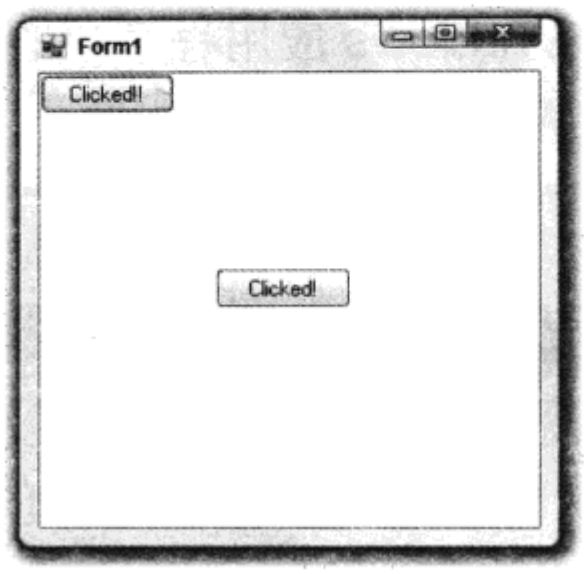


图　8-15

示例的说明

添加几行代码，就创建了一个可以完成某项任务的 Windows 应用程序。下面说明 C#中的一些 OOP 技术。即使在谈到 Windows 应用程序时，“一切皆对象”这句话也是正确的。从运行的窗体，到窗体上的控件，都需要使用OOP 技术。在这个示例中，重点说明本章前面介绍的一些概念，解释如何把它们组合在一起。

在应用程序中，首先是在 Form1 窗体上添加一个新按钮，这个按钮是一个对象，它是 Button 类的一个实例；窗体是 Form1 类的实例，该类从 Form 类派生而来。接着双击按钮，添加一个事件处理程序，监听 Button 类提供的Click 事件。这个事件处理程序添加到封装应用程序的 Form 对象代码中，是一个私有方法：

```
private void button1_Click(object sender, System.EventArgs e)
{
}
```

这段代码使用 C#关键字 private 作为修饰符。现在不要考虑这个关键字，第 9 章将详细解释本章提及的 OOP 技术。

我们添加的第一行代码改变了按钮上的文本。它利用了本章前面讨论的多态性。表示按钮的 Button 对象作为一个 object 参数发送给事件处理程序，该事件处理程序把参数强制转换为 Button 类型(这是可能的，因为 Button 对象继承于 System.Object，System.Object 是一个.NET 类，object 是其别名)。然后修改对象的 Text 属性，改变显示的文本：

```
((Button)sender).Text = "Clicked!";
```

接着用 new 关键字创建一个新 Button 对象(注意在这个项目中设置了名称空间，因此可以使用这个简单的语法，否则，就需要使用这个对象的完整限定名 System.Windows.Forms.Button)：

```
Button newButton = new Button();
newButton.Text = "New Button!";
```

在代码的其他地方添加一个新的事件处理程序，以响应新按钮生成的 Click 事件：

```
private void newButton_Click(object sender, System.EventArgs e)
{
```

```
        ((Button)sender).Text = "Clicked!!";
    }
```

接着使用一些重载运算符语法，把这个事件处理程序注册为 Click 事件的监听程序。同时使用非默认的构造函数创建一个新的 EventHandler 对象，其名称是新事件处理函数的名称：

```
newButton.Click += new EventHandler(newButton_Click);
```

最后，利用窗体的 Controls 属性，这个属性是一个对象，是窗体上所有控件的集合，通过它的 Add()方法把新按钮添加到窗体上：

```
Controls.Add(newButton);
```

Controls 属性说明，属性不一定是字符串或整型等简单类型，可以是任何类型的对象。这个简短示例几乎使用了本章介绍的所有技术。可以看出，OOP 编程并不复杂——只需要从另一个角度来看待它即可。

8.4　小结

本章完整地描述了面向对象技术。我们在 C#编程环境中进行论述，但主要是用示例来说明。本章介绍的 OOP 大都适用于任何语言。

首先介绍基础知识，例如术语“对象”的含义，对象如何成为类的实例。接着讨论对象有各种成员，例如字段、属性和方法。这些成员的可访问性都有一定的限制，然后解释了公共和私有成员。之后，说明成员也可以是受保护的，还可以是虚拟和抽象的(抽象方法只能存在于抽象类中)，另外还解释了静态(共享)和实例成员的区别，说明使用静态类的原因。

接下来简要介绍了对象的生命周期，包括如何使用构造函数创建对象，如何使用析构函数删除对象。在说明了接口如何组合对象后，介绍了更高级的对象删除方式：支持 IDisposable 接口的可删除对象。

本章的其他部分重点介绍了OOP 的特性，其中有许多特性将在随后的章节中详细讨论。我们论述了继承(类可以继承基类)，两个版本的多态性(即基类和共享接口)，对象如何用于包含一个或多个其他对象(通过包含和集合关系)。最后介绍运算符重载如何用于简化使用对象的语法，对象如何引发事件。

本章的最后一部分用一个 Windows 应用程序示例演示了许多理论。第 9 章将介绍如何使用 C#定义类。

8.5　练习

(1) 下述哪些项在 OOP 中有真实级别的可访问性？

- 友元
- 公共
- 安全

- 私有
- 受保护的
- 松散的
- 通配符

(2) “必须手动调用对象的析构函数，否则就会浪费资源”的说法正确吗?

(3) 只有创建一个对象，才能调用其类的静态方法吗?

(4) 为下述类和接口绘制一个类似于本章介绍的图形的 UML 图:

- 抽象类 HotDrink，它有方法 Drink()、AddMilk()和 AddSugar()，以及属性 Milk 和 Sugar。
- 接口 ICup，它有方法 Refill()和 Wash()，以及属性 Color 和 Volume。
- 派生于 HotDrink 的类 CupOfCoffee 支持 ICup 接口，还有一个属性 BeanType。
- 派生于 HotDrink 的类 CupOfTea 支持 ICup 接口，还有一个属性 LeafType。

(5) 为一个函数编写一些代码，接受上述示例的两个杯子对象中的任意一个，作为一个参数。该函数应可以为它传送的任何杯子对象调用 AddMilk()、Drink()和 Wash()方法。

附录 A 给出了练习答案。

8.6 本章要点

主　题	重 要 概 念
对象和类	对象是 OOP 应用程序的组成部件。类是用于实例化对象的类型定义。对象可以包含数据，提供其他代码可以使用的操作。数据可以通过属性供外部代码使用，操作可以通过方法供外部代码使用。属性和方法都称为类成员。属性可以进行读取访问、写入访问或读写访问。类成员可以是公共的(可用于所有的代码)或私有的(只有类定义中的代码可以使用)。在.NET 中，所有的东西都是对象
对象的生存周期	对象通过调用它的一个构造函数来实例化。不再需要对象时，就执行其析构函数，以删除它。要清理对象，常常需要手工删除它
静态和实例成员	实例成员只能在类的对象实例上使用，静态成员只能直接通过类定义使用，它不与实例关联
接口	接口是可以在类上实现的公共属性和方法的集合。可以给实例类型的变量赋予其类定义实现了该接口的任意对象的值。之后通过该变量，可以使用该接口定义的成员
继承	继承是一个类定义派生于另一个类定义的机制。类从其父类中继承成员，每个类都只能有一个父类。子类不能访问父类的私有成员，但可以定义受保护的成员，受保护的成员只能在该类和派生于该类的子类中使用。子类可以重写父类中定义为虚拟的成员。所有的类都有一个以 System.Object 结尾的继承链，在 C#中，System.Object 有一个别名 Object
多态性	从一个派生类中实例化的所有对象都可以看作是其父类的实例
对象关系和特性	对象可以包含其他对象，也可以表示其他对象的集合。要在表达式中处理对象，常常需要通过运算符重载，定义运算符如何处理对象。对象可以提供事件，事件因某种内部处理而被触发，客户代码可以提供事件处理程序，来响应事件

第 9 章

定 义 类

本章内容：

- 如何在 C#中定义类和接口
- 如何使用控制可访问性和继承的关键字
- System.Object 类及其在类定义中的作用
- 如何使用 VS 和 VCE 提供的一些帮助工具
- 如何定义类库
- 接口和抽象类的异同
- 结构类型的更多内容
- 复制对象的一些重要信息

第 8 章介绍了面向对象编程(OOP)的特性，本章则要将理论付诸实践，看看如何在 C#中定义类。本章并不讨论如何定义类的成员，而重点讨论如何定义类本身。这听起来有一定的限制，但不必担心，本章有足够丰富的内容供读者学习。

首先看看基本的类定义语法、用于确定类可访问性的关键字以及指定继承的方式。我们还将介绍接口的定义，因为它们在许多方面都类似于类的定义。

本章的其他部分介绍在 C#中定义类时涉及到的其他主题。

9.1 C#中的类定义

C#使用 class 关键字来定义类：

```
class MyClass
{
   // Class members.
}
```

这段代码定义了一个类 MyClass。定义了一个类后，就可以在项目中能访问该定义的其他位置

对该类进行实例化。默认情况下，类声明为内部的，即只有当前项目中的代码才能访问它。可以使用 internal 访问修饰符关键字显式指定，如下所示(但这是不必要的)：

```
internal class MyClass
{
   // Class members.
}
```

另外，还可以指定类是公共的，应该可以由其他项目中的代码来访问。为此，要使用关键字 public。

```
public class MyClass
{
   // Class members.
}
```

以这种方式声明的类不能是私有或受保护的。可以把这些声明类的修饰符用于声明类成员，详见第 10 章。

除了这两个访问修饰符关键字外，还可以指定类是抽象的(不能实例化，只能继承，可以有抽象成员)或密封的(sealed，不能继承)。为此，可以使用两个互斥的关键字 abstract 或 sealed。所以，抽象类必须用下述方式声明：

```
public abstract class MyClass
{
   // Class members, may be abstract.
}
```

其中 MyClass 是一个公共抽象类，也可以是内部抽象类。

密封类的声明如下所示：

```
public sealed class MyClass
{
   // Class members.
}
```

与抽象类一样，密封类也可以是公共或内部的。

还可以在类定义中指定继承。为此，要在类名的后面加上一个冒号，其后是基类名，例如：

```
public class MyClass : MyBase
{
   // Class members.
}
```

注意，在 C#的类定义中，只能有一个基类，如果继承了一个抽象类，就必须实现所继承的所有抽象成员(除非派生类也是抽象的)。

编译器不允许派生类的可访问性高于基类。也就是说，内部类可以继承于一个公共基类，但公共类不能继承于一个内部类。因此，下述代码是合法的：

```
public class MyBase
```

```
{
   // Class members.
}

internal class MyClass : MyBase
{
   // Class members.
}
```

但下述代码不能编译：

```
internal class MyBase
{
   // Class members.
}
public class MyClass : MyBase
{
   // Class members.
}
```

如果没有使用基类，则被定义的类就只继承于基类 System.Object(它在C#中的别名是 object)。毕竟，在继承层次结构中，所有类的根都是 System.Object，稍后将详细介绍这个基类。

除了以这种方式指定基类外，还可以在冒号之后指定支持的接口。如果指定了基类，它必须紧跟在冒号的后面，之后才是指定的接口。如果没有指定基类，则接口就跟在冒号的后面。必须使用逗号分隔基类名(如果有基类)和接口名。

例如，给 MyClass 添加一个接口，如下所示：

```
public class MyClass : IMyInterface
{
   // Class members.
}
```

所有接口成员都必须在支持该接口的类中实现，但如果不想使用给定的接口成员，就可以提供一个“空”的实现方式(没有函数代码)。还可以把接口成员实现为抽象类中的抽象成员。

下面的声明是无效的，因为基类 MyBase 不是继承列表中的第一项：

```
public class MyClass : IMyInterface, MyBase
{
   // Class members.
}
```

指定基类和接口的正确方式如下：

```
public class MyClass : MyBase, IMyInterface
{
   // Class members.
}
```

可以指定多个接口，所以下列代码是有效的：

```
public class MyClass : MyBase, IMyInterface, IMySecondInterface
{
```

```
   // Class members.
}
```

表 9-1 是类定义中可以使用的访问修饰符的组合。

表 9-1

修 饰 符	含 义
无或 internal	只能在当前项目中访问类
public	可以在任何地方访问类
abstract 或 internal abstract	类只能在当前项目中访问，不能实例化，只能供继承之用
public abstract	类可以在任何地方访问，不能实例化，只能供继承之用
sealed 或 internal sealed	类只能在当前项目中访问，不能供派生之用，只能实例化
public sealed	类可以在任何地方访问，不能供派生之用，只能实例化

接口的定义

声明接口的方式与声明类的方式相似，但使用的关键字是 interface，而不是 class，例如：

```
interface IMyInterface
{
   // Interface members.
}
```

访问修饰符关键字 public 和 internal 的使用方式是相同的，与类一样，接口也默认定义为内部接口。所以要使接口可以公开访问，必须使用 public 关键字：

```
public interface IMyInterface
{
   // Interface members.
}
```

不能在接口中使用关键字 abstract 和 sealed，因为这两个修饰符在接口定义中是没有意义的(它们不包含实现代码，所以不能直接实例化，且必须是可以继承的)。

接口的继承也可以用与类继承类似的方式来指定。主要的区别是可以使用多个基接口，例如：

```
public interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
{
   // Interface members.
}
```

接口不是类，所以没有继承 System.Object。但是为了方便起见，System.Object 的成员可以通过接口类型的变量来访问。如上所述，不能用实例化类的方式来实例化接口。下面的示例提供了一些类定义的代码和使用它们的代码。

试一试：定义类

(1) 在 C:\BegVCSharp\Chapter09 目录中创建一个新的控制台应用程序 Ch09Ex01。

(2) 修改 Program.cs 中的代码，如下所示：

```
namespace Ch09Ex01
{
   public abstract class MyBase
   {
   }

   internal class MyClass : MyBase
   {
   }

   public interface IMyBaseInterface
   {
   }

   internal interface IMyBaseInterface2
   {
   }

internal interface IMyInterface : IMyBaseInterface, IMyBaseInterface2
   {
   }

   internal sealed class MyComplexClass : MyClass, IMyInterface
   {
   }
   class Program
    {
      static void Main(string[] args)
      {
         MyComplexClass myObj = new MyComplexClass();
         Console.WriteLine(myObj.ToString ());
         Console.ReadKey();
      }
   }
}
```

代码段Ch09Ex01\Program.cs

(3) 执行项目，结果如图 9-1 所示。

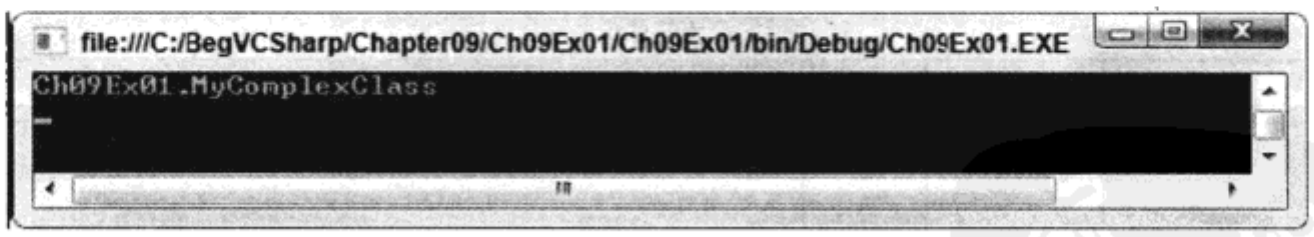


图 9-1

示例的说明

这个项目在下面的继承层次结构中定义了类和接口，如图 9-2 所示。

这里包含Program，是因为这个类的定义方式与其他类的定义方式相同，而它不是主要类层次结构中的一部分。这个类处理的 Main()方法是应用程序的入口点。

MyBase 和 IMyBaseInterface 被定义为公共的，所以它们可以在其他项目中使用。其他类和接口都是内部的，只能在本项目中使用。

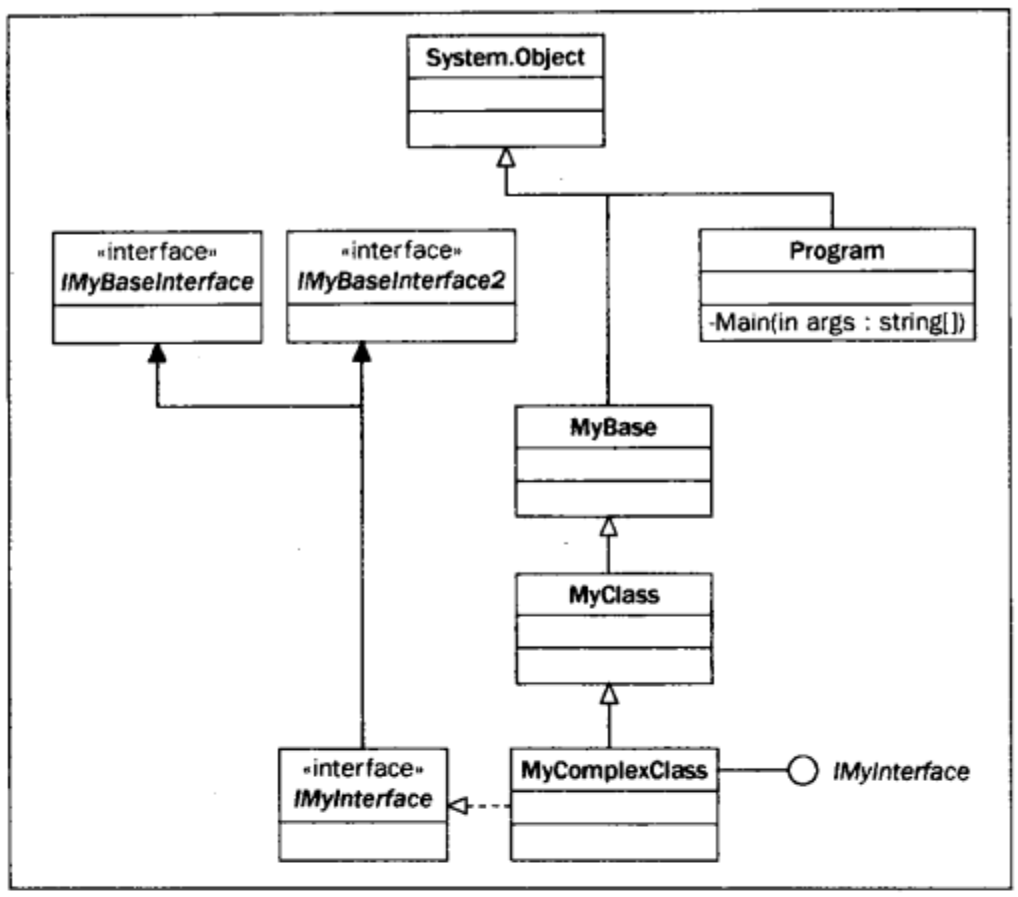


图　9-2

Main()中的代码调用 MyComplexClass 的一个实例 myObj 的 ToString()方法：

```
MyComplexClass myObj = new MyComplexClass();
Console.WriteLine(myObj.ToString());
```

这是继承自System.Object 的一个方法(图中没有显示，该图省略了这个类的成员，使图变得更清晰)，并把对象的类名作为一个字符串返回，该类名用任意相关的命名空间来限定。

这个示例没有完成什么具体的工作，但本章后面还要利用这个示例演示几个重要概念和技术。

9.2　System.Object

因为所有的类都继承于 System.Object，所以这些类都可以访问该类中受保护的成员和公共的成员。下面看看可供使用的成员有哪些。System.Object 包含的方法如表 9-2 所示。

表　9-2

方　法	返回类型	虚拟	静态	说　明
Object()	N/A	无	无	System.Object 类型的构造函数，由派生类型的构造函数自动调用
~Object()(也 称 为 Finalize()，参见下一节)	N/A	无	无	System.Object 类型的析构函数，由派生类型的析构函数自动调用，不能手动调用
Equals(object)	bool	有	无	把调用该方法的对象与另一个对象相比，如果它们相等，就返回 true。默认的实现代码会查看对象的参数是否引用了同一个对象(因为对象是引用类型)。如果想以不同的方式来比较对象，则可以重写该方法，例如，比较两个对象的状态

(续表)

方 法	返回类型	虚拟	静态	说 明
Equals (object, object)	bool	无	有	这个方法比较传送给它的两个对象，看看它们是否相等。检查时使用了 Equals(object) 方法。注意，如果两个对象都是空引用，这个方法就返回 true
ReferenceEquals (object,object)	bool	无	有	这个方法比较传送给它的两个对象，看看它们是否是同一个实例的引用
ToString()	String	有	无	返回一个对应于对象实例的字符串。默认情况下，这是一个类类型的限定名称，但可以重写它，给类型提供合适的实现方式
MemberwiseClone()	object	无	无	通过创建一个新对象实例并复制成员，以复制该对象。成员拷贝不会得到这些成员的新实例。新对象的任何引用类型成员都将引用与源类相同的对象，这个方法是受保护的，所以只能在类或派生的类中使用
GetType()	System. Type	无	无	以 System.Type 对象的形式返回对象的类型
GetHashCode()	int	有	无	用作对象的散列函数，这是一个必选函数，返回一个以压缩形式标识的对象状态的值

这些方法是.NET Framework 中对象类型必须支持的基本方法，但我们可能从不使用其中某些类型(或者只在特殊情况下使用，如 GetHashCode())。

在利用多态性时，GetType()是一个有用的方法，允许根据对象的类型来执行不同的操作，而不是像通常那样，对所有的对象都执行相同的操作。例如，如果函数接受一个 object 类型的参数(表示可以给该函数传送任何信息)，就可以在遇到某些对象时执行额外的任务。联合使用 GetType()和 typeof(这是一个 C#运算符，可以把类名转换为 System.Type 对象)，就可以进行比较，如下所示：

```
if (myObj.GetType() == typeof(MyComplexClass))
{
   // myObj is an instance of the class MyComplexClass.
}
```

返回的 System.Type 对象可以做更多的工作，这里不讨论它们。重写 ToString()方法也是非常有效的，特别是在对象的内容中可以用一个人们能理解的字符串表示时，就更是如此。后面的章节将反复讨论这些 System.Object 方法，现在就讨论到这里，后面在需要时再详细讨论。

9.3 构造函数和析构函数

在 C#中定义类时，常常不需要定义相关的构造函数和析构函数，因为在建立代码时，如果没有提供它们，编译器会自动添加它们。但是，如果需要，可以提供自己的构造函数和析构函数，以便

初始化对象和清理对象。

使用下述语法可以把一个简单的构造函数添加到类中：

```
class MyClass
{
   public MyClass()
   {
      // Constructor code.
   }
}
```

这个构造函数与包含它的类同名，且没有参数(使之成为类的默认构造函数)，这是一个公共函数，所以类的对象可以使用这个构造函数进行实例化(详见第 8 章)。

也可以使用私有的默认构造函数，即不能用这个构造函数来创建这个类的对象实例(它是不可创建的，详见第 8 章)：

```
class MyClass
{
   private MyClass()
   {
      // Constructor code.
   }
}
```

最后，也可以用相同的方式给类添加非默认的构造函数，其方法是提供参数，例如：

```
class MyClass
{
   public MyClass()
   {
      // Default constructor code.
   }

   public MyClass(int myint)
   {
      // Nondefault constructor code (uses myInt).
   }
}
```

可提供的构造函数的数量不受限制(当然不能耗尽内存，也不能有相同的参数集，所以“几乎无限制”更合适)。

使用略微不同的语法来声明析构函数。在.NET 中使用的析构函数(由 System.Object 类提供)叫作 Finalize()，但这不是我们用于声明析构函数的名称。使用下面的代码，而不是重写 Finalize()：

```
class MyClass
{
   ~MyClass()
   {
      // Destructor body.
   }
}
```

类的析构函数由带有～前缀的类名(与构造函数的相同)来声明。当进行垃圾回收时，就执行析构函数中的代码，释放资源。在调用这个析构函数后，还将隐式地调用基类的析构函数，包括System.Object 根类中的Finalize()调用。这个技术可以让.NET Framework 确保调用 Finalize()，因为重写 Finalize()是指基类调用需要显式地执行，这是具有潜在危险的(第 10 章将详细讨论如何调用基类的方法)。

构造函数的执行序列

如果在类的构造函数中执行多个任务，把这些代码放在一个地方是非常方便的，这与第 6 章论述的把代码放在函数中有相同的优势。使用一个方法就可以把代码放在一个地方(详见第 10 章)，而C#提供了一个更好的替代方式。任何构造函数都可以配置，在执行自己的代码前调用其他构造函数。

在讨论构造函数前，先看看在默认情况下，创建类的实例时会发生什么情况。除了前面说过的便于把初始化代码集中起来之外，还要了解这些代码。在开发过程中，对象常常并没有按照预期的那样执行，而是在调用构造函数时出现错误。这常常是因为类继承结构中的某个基类没有正确实例化，或者没有正确地给基类构造函数提供信息。如果理解在对象生命周期的这个阶段发生的事情，将更利于解决这类问题。

为了实例化派生的类，必须实例化它的基类。而要实例化这个基类，又必须实例化这个基类的基类，这样一直到实例化 System.Object(所有类的根)为止。结果是无论使用什么构造函数实例化一个类，总是要先调用 System.Object.Object。

无论在派生类上使用什么构造函数(默认的构造函数或非默认的构造函数)，除非明确指定，否则就使用基类的默认构造函数(稍后将介绍如何改变这个操作)。下面介绍一个简短示例，说明执行的顺序。考虑下面的对象层次结构：

```
public class MyBaseClass
{
   public MyBaseClass()
   {
   }

   public MyBaseClass(int i)
   {
   }
}

public class MyDerivedClass : MyBaseClass
{
   public MyDerivedClass()
   {
   }

   public MyDerivedClass(int i)
   {
   }

   public MyDerivedClass(int i, int j)
   {
```

```
    }
}
```

如果以下面的方式实例化 MyDerivedClass：

```
MyDerivedClass myObj = new MyDerivedClass();
```

则执行的顺序如下：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass()构造函数。
- 执行 MyDerivedClass.MyDerivedClass()构造函数。

另外，如果使用下面的语句：

```
MyDerivedClass myObj = new MyDerivedClass(4);
```

则执行的顺序如下：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass() 构造函数。
- 执行 MyDerivedClass.MyDerivedClass(int i) 构造函数。

最后，如果使用下面的语句：

```
MyDerivedClass myObj = new MyDerivedClass(4, 8);
```

则执行顺序如下：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass()构造函数。
- 执行 MyDerivedClass.MyDerivedClass(int i, int j)构造函数。

大多数情况下，这个系统会正常工作。但是，有时需要对发生的事件进行更多的控制。例如，在上面的实例化示例中，执行的顺序如下：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass(int i)构造函数。
- 执行 MyDerivedClass.MyDerivedClass(int i, int j)构造函数。

使用这个顺序可以把使用 int i 参数的代码放在 MyBaseClass(int i)中，即 MyDerivedClass(int i, int j)构造函数要做的工作比较少，只需要处理 int j 参数(假定 int i 参数在两种情况下含义相同，虽然事情并非总是如此，但实际上我们常常做这样的安排)。只要愿意，C#就可以指定这种操作。

为此，只需使用构造函数初始化器，它把代码放在方法定义的冒号后面。例如，可以在派生类的构造函数定义中指定所使用的基类构造函数，如下所示：

```
public class MyDerivedClass : MyBaseClass
{
   ...

   public MyDerivedClass(int i, int j) : base(i)
   {
   }
}
```

其中，base 关键字指定.NET 实例化过程使用基类中有指定参数的构造函数。这里使用了一个 int 参数(其值通过参数 i 传送给 MyDerivedClass 构造函数)，所以应使用 MyBaseClass(int i)。这么做将不调用 MyBaseClass()，而是执行本例前面列出的事件序列—— 也就是我们希望执行的事件序列。

也可以使用这个关键字指定基类构造函数的字面值，例如，使用 MyDerivedClass 的默认构造函数调用 MyBaseClass 非默认的构造函数：

```
public class MyDerivedClass : MyBaseClass
{
   public MyDerivedClass() : base(5)
   {
   }
   ...
}
```

这段代码将执行下述序列：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass(int i)构造函数。
- 执行 MyDerivedClass.MyDerivedClass()构造函数。

除了 base 关键字外，这里还可以将另一个关键字 this 用作构造函数初始化器。这个关键字指定在调用指定的构造函数前，.NET 实例化过程对当前类使用非默认的构造函数。例如：

```
public class MyDerivedClass : MyBaseClass
{
   public MyDerivedClass() : this(5, 6)
   {
   }
   ...

   public MyDerivedClass(int i, int j) : base(i)
   {
   }
}
```

这段代码将执行下述序列：

- 执行 System.Object.Object()构造函数。
- 执行 MyBaseClass.MyBaseClass(int i)构造函数。
- 执行 MyDerivedClass.MyDerivedClass(int i, int j)构造函数。
- 执行 MyDerivedClass.MyDerivedClass()构造函数。

唯一的限制是使用构造函数初始化器只能指定一个构造函数。但是，如上一个示例所示，这并不是一个很严格的限制，因为我们仍可以构造相当复杂的执行序列。

如果没有给构造函数指定构造函数初始化器，编译器就会自动添加 base()。这会执行本节前面介绍的默认序列。

注意在定义构造函数时，不要创建无限循环。例如：

```
public class MyBaseClass
{
  public MyBaseClass() : this(5)
  {
  }
  public MyBaseClass(int i) : this()
  {
  }
}
```

使用上述任何一个构造函数，都需要先执行另一个构造函数，而另一个构造函数需要先执行原告的构造函数，因此这段代码可以编译，但如果尝试实例化 MyBaseClass，就会得到一个 SystemOverflowException 异常。

9.4　VS 和 VCE 中的 OOP 工具

OOP 在.NET Framework 中是一个非常基础的主题，所以 VS 和 VCE 提供了几个工具来帮助开发 OOP 应用程序。本节就介绍其中的一些工具。

9.4.1　Class View 窗口

第 2 章介绍了 Solution Explorer 窗口与 Class View 窗口共用相同的空间。这个窗口显示了应用程序中的类层次结构，可供查看我们使用的类的特性。对于上一节的示例项目，其视图如图 9-3 所示。

这个窗口分为两半，底下的一半显示了类型的成员。为了使用 Class View 窗口查看这个示例项目的成员和其他内容，需要显示当前隐藏的一些项。为此，应在 Class View 窗口中勾选 Class View Grouping 下拉列表中的项，如图 9-4 所示。

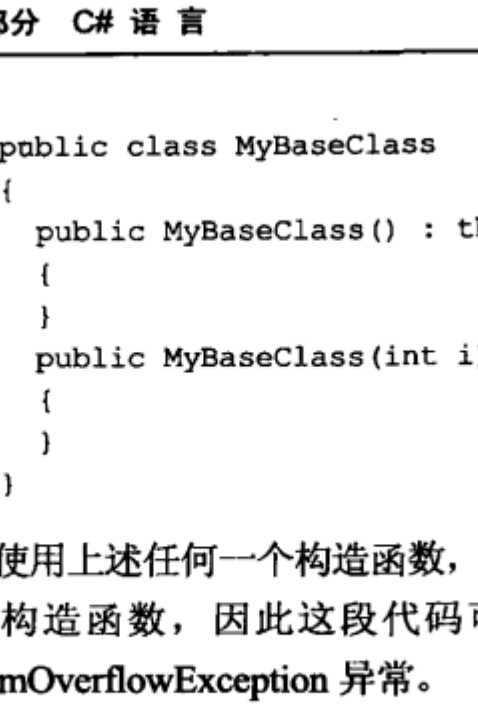


图　9-3

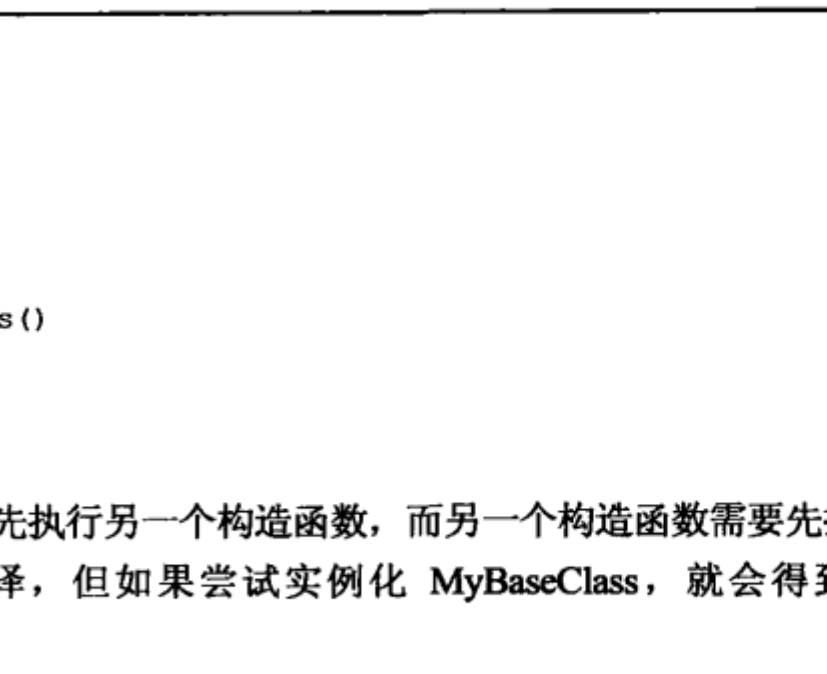


图　9-4

现在就可以看到成员和其他信息，如图 9-5 所示。

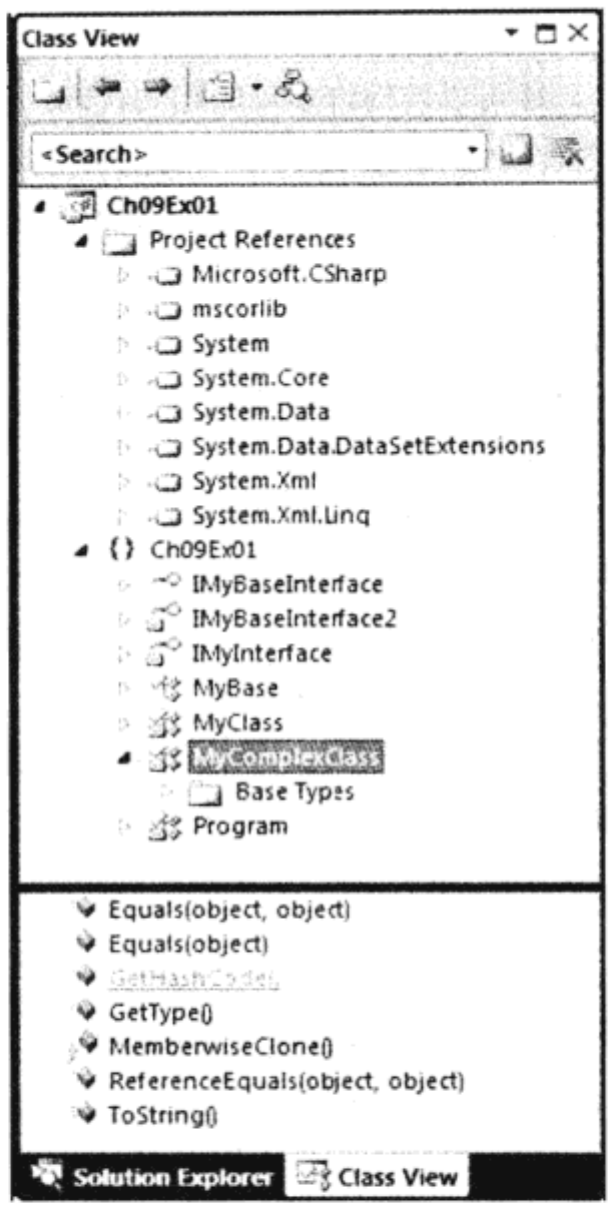


图 9-5

这里使用了许多符号，如表 9-3 中所示。

表 9-3

图 标	含 义	图 标	含 义	图 标	含 义
	项目		属性		事件
{}	名称空间		字段		委托
	类		结构		程序集
	接口		枚举		
	方法		枚举项		

注意，其中一些图标用于类型定义，而不是类定义，例如，枚举和结构类型。

还可以在一些项的下面放置其他符号，表示它们的访问级别(公共项没有这样的符号)，表 9-4 中列出了这些符号。

表 9-4

图 标	含 义	图 标	含 义	图 标	含 义
	私有		受保护的		内部

没有符号用于表示抽象、密封和虚拟项。

在这里除了可以查看信息外，还可以访问许多项的相关代码。双击某个项，或者右击该项，并选择 Go To Definition，就可以查看项目中用于定义该项的代码(假定代码是可以查看的)。如果无法查看代码，例如不能访问基类型 System.Object 中的代码，就应选择 Browse Definition，打开 Object Browser 视图(详见下一节)。

图 9-5 显示的另一项是 Project References，它可以供查看项目引用了哪些程序集，本例的项目包含 mscorlib 和 System 中的核心.NET 类型、system.Data 中的数据访问类型和 System.Xml 中的 XML 操纵类型。这里的引用也是可以扩展的，显示这些程序集中包含的名称空间和类型。

Class View 还可以查找代码中的类型和成员。其方法是，右击一项，选择 Find All References，就会在 Find Symbol Results 窗口中打开搜索结果列表，该窗口位于屏幕底部，是 Error List 显示区域的一个选项卡。还可以使用 Class View 给项重命名：在重命名时，可以重命名代码中出现的项的引用。也就是说，类名中不能有拼写错误，因为我们可以随时修改它们。

另外，VS2010 引入了浏览代码的一种新方式，称为调用层次结构，通过 View Call Hierarchy 右击菜单项就可以在 Class View 窗口中访问 Call Hierarchy 窗口。这个功能非常适于查看类成员如何彼此交互，参见下一章。

9.4.2　对象浏览器

对象浏览器(Object Browser)是 Class View 窗口的扩展版本，可以查看项目中能使用的其他类，甚至可以查看外部的类。可以自动(如上一节的情况)或手动(通过 View | Object Browser)进入这个窗口。这个视图显示在主窗口中，可以用与 Class View 窗口相同的方式浏览该视图。

这个窗口显示了与 Class View 窗口相同的信息，还显示了.NET 类型的其他信息。选中某项，还可以在第三个窗口中获得该项的信息，如图 9-6 所示。

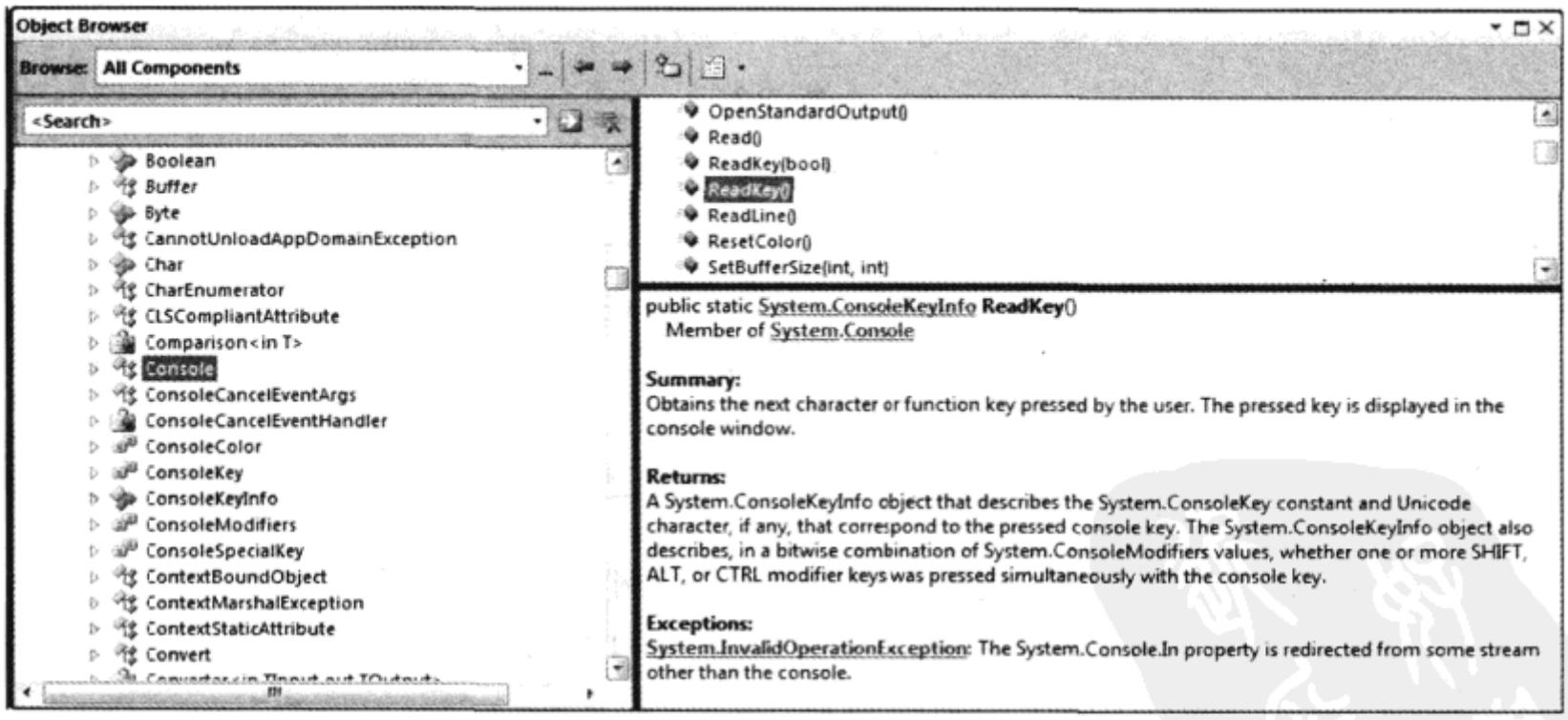


图　9-6

在图 9-6 中，选中了 Console 类的 ReadKey()方法(Console 在 mscorlib 程序集的 System 名称空间中)。右下角的信息窗口显示了方法签名、该方法所属的类和方法函数的小结。在研究.NET 类型时，或者了解某个类的用途时，这些信息非常有用。

另外，还可以在自己创建的类型中使用这个信息窗口。对 Ch09Ex01 中的代码进行如下修改：

可从 wrox.com 下载源代码

```
/// <summary>
/// This class contains my program!
/// </summary>
class Program
{
   static void Main(string[] args)
   {
      MyComplexClass myObj = new MyComplexClass();
      Console.WriteLine(myObj.ToString());
      Console.ReadKey();
   }
}
```

代码段 Ch09Ex01\Program.cs

然后返回到对象浏览器，就会看到这些变化反映在信息窗口中。这是 XML 文档说明的一个示例，本书不讨论 XML 文档说明，但读者有闲暇时间时，应学习这个主题。

如果手工修改上面的代码，只要键入 3 个斜杠///，IDE 就会添加输入的其他内容。它会自动分析应用于 XML 文档说明的代码，建立基本的 XML 文档说明。显然，如果需要 XML 文档说明，VS 和 VCE 就是一个很强大的工具。

9.4.3 添加类

VS 和 VCE 包含可以加速执行某些常见任务的工具，其中一些可以应用于 OOP。有一个 Add New Item Wizard 工具可以给项目快速添加新类，且需要键入的代码数量最少。

该工具的访问方式是单击 Project | Add New Item 菜单项，或在 Solution Explorer 窗口中右击项目，选择相应的项。采用这两种方式，都会打开一个对话框，在该对话框中，可以选择要添加的项。这个窗口的默认显示在 VS 和 VCE 中是不同的，但功能相同。在这两个 IDE 中，要添加一个类，可以在 Templates 窗口中选择 Class 项，如图 9-7 所示，为包含类的文件提供一个文件名，再单击 Add 按钮。所创建的类就以所提供的文件名命名。

在本章前面的示例中，我们在 Program.cs 文件中手动添加类定义。把类放在独立的文件中，常常可以更轻松地跟踪类。打开 Ch09Ex01 项目后，在 Add New Item 对话框中输入信息，就会在 MyNewClass.cs 中生成下列代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch09Ex01
{
```

```
    class MyNewClass
    {
    }
}
```

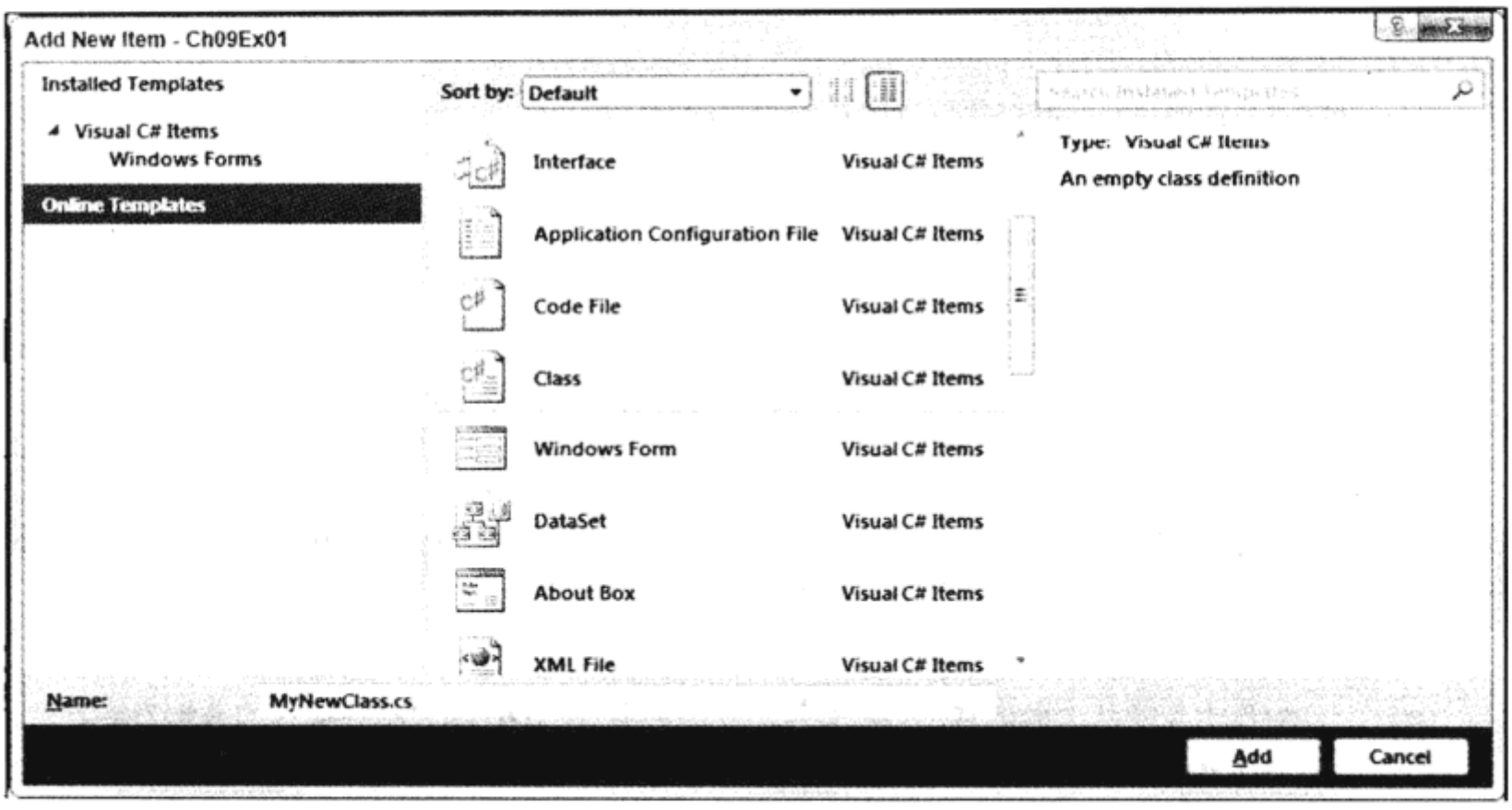


图　9-7

这个类 MyNewClass 定义在入口点类 Program 所在的名称空间中，所以可以在代码中使用它，就像它们是在相同的文件中定义一样。从代码中可以看出，生成的类不包含构造函数。如果类定义没有包含构造函数，编译器就会在编译代码时自动添加一个默认的构造函数。

9.4.4　类图

还没有介绍的 VS 的一个强大功能是从代码中生成类图，并使用类图修改项目。VS 中的类图编辑器可以很方便地为代码生成类似于 UML 的图。为了描述这个功能，下面的示例将为前面创建的 Ch09Ex01 项目生成类图。

VCE 没有类图功能，所以只能在 VS 中建立这个示例。

试一试：生成类图

(1) 打开本章前面创建的 Ch09Ex01 项目。

(2) 在 Solution Explorer 窗口中，选择 Program.cs，单击工具栏中的 View Class Diagram 按钮，如图 9-8 所示。

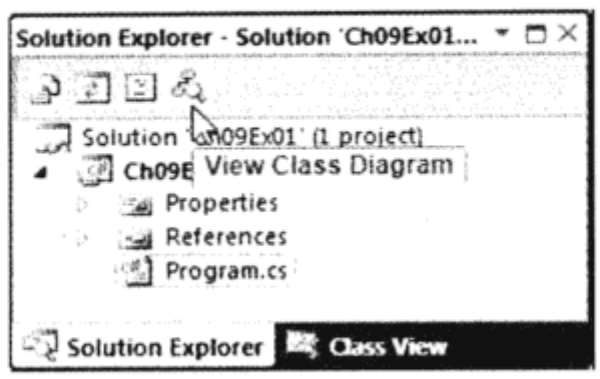


图 9-8

(3) 显示一个类图 ClassDiagram1.cd。

(4) 单击 IMyInterface“棒棒糖”，在 Properties 窗口中，把它的 Position 属性改为 Right。

(5) 右击 MyBase，从上下文菜单中选择 Show Base Type 选项。

(6) 拖动图中的对象，生成较好的布局。完成这些步骤后，类图将如图 9-9 所示。

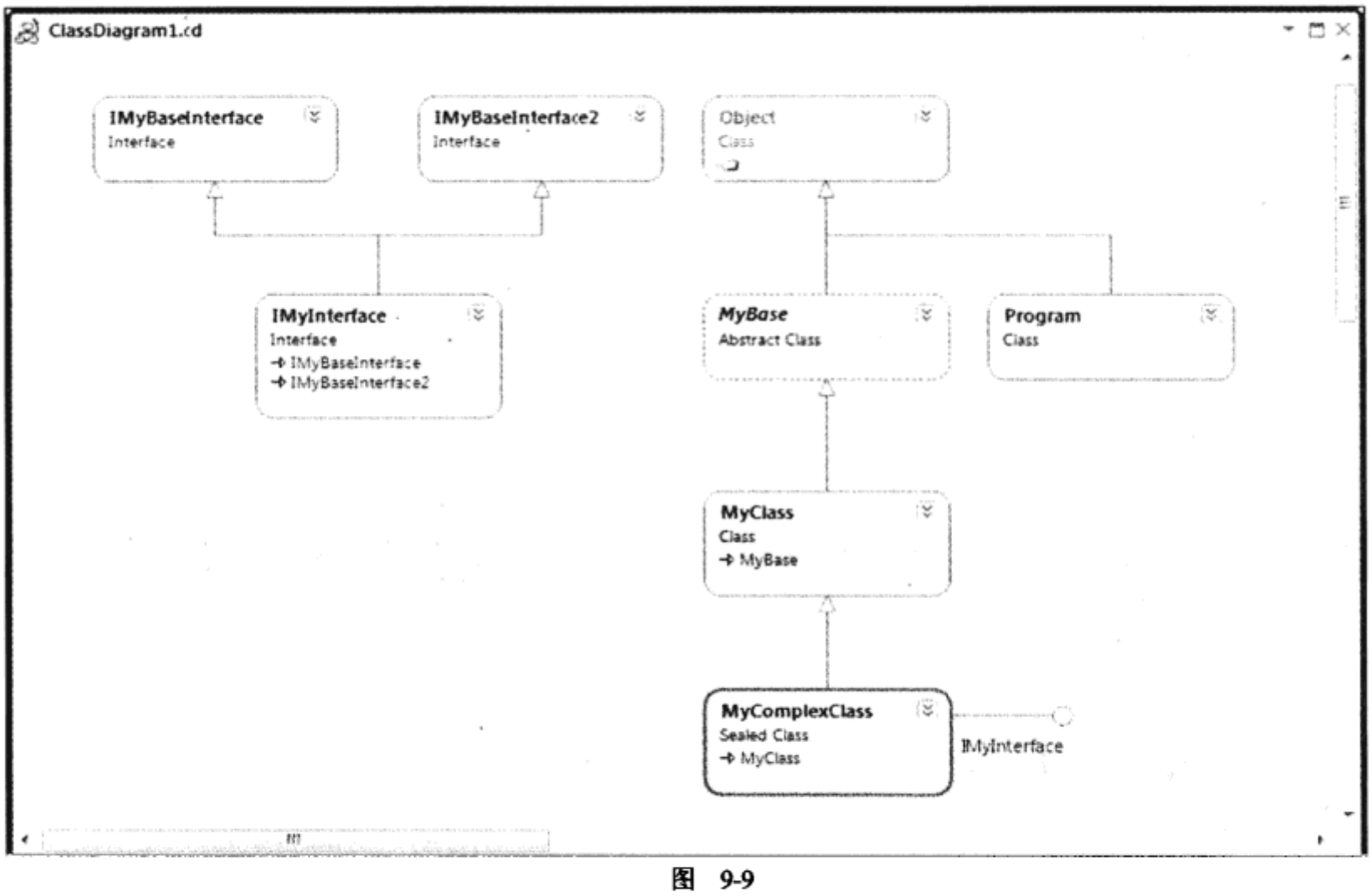


图 9-9

示例的说明

本例毫不费力地创建了一个与 UML 图(见图 9-2)非常类似的类图，下面的特性得到了证明：

- 类显示为蓝色框，其中包含类的名称和类型。
- 接口显示为绿色框，其中包含接口的名称和类型。
- 继承用白色箭头表示，在某些情况下，类框中包含文本。
- 实现接口的类有“棒棒糖”图标。
- 抽象类显示为虚点外框，名称显示为斜体。
- 密封类显示为粗黑外框。

单击一个对象会在屏幕底部的 Class Details 窗口中显示其他信息(如果 Class Details 窗口没有显

示出来，可以右击一个对象，选择 Class Details)。可以在此查看(和修改)类成员，还可以在 Properties 窗口中修改类的信息。

第 10 章将深入讨论如何使用类图给类添加成员。

在 Toolbox 中，可以给图添加新项，例如，类、接口和枚举等，定义图中对象之间的关系。此时，新项的代码会自动生成。

使用这个编辑器可以图形化地设计整个类型系列，而无需使用代码编辑器。显然，在实际添加功能时，必须手工完成一些工作，但这个类图编辑器是开始工作的一种绝佳方式。后面的章节还将介绍这个视图，了解它的其他用途。现在读者可以自己学习其功能。

9.5　类库项目

除了在项目中把类放在不同的文件中之外，还可以把它们放在完全不同的项目中。如果一个项目什么都不包含，只包含类(以及其他相关的类型定义，但没有入口点)，该项目就称为类库。

类库项目编译为.dll 程序集，在其他项目中添加对类库项目的引用，就可以访问它的内容(这可以是同一个解决方案的一部分，但这不是必须的)。这将扩展对象提供的封装性，因为类库可以进行修改和更新，而不会影响使用它们的其他项目。这意味着，您可以方便地升级类提供的服务(这会影响多个用户应用程序)。

下面看一个类库项目的示例和一个利用该项目包含的类的独立项目。

试一试：使用类库

(1) 在 C:\BegVCSharp\Chapter09 目录中创建一个 Class Library 类型的新项目 Ch09ClassLib，如图 9-10 所示。

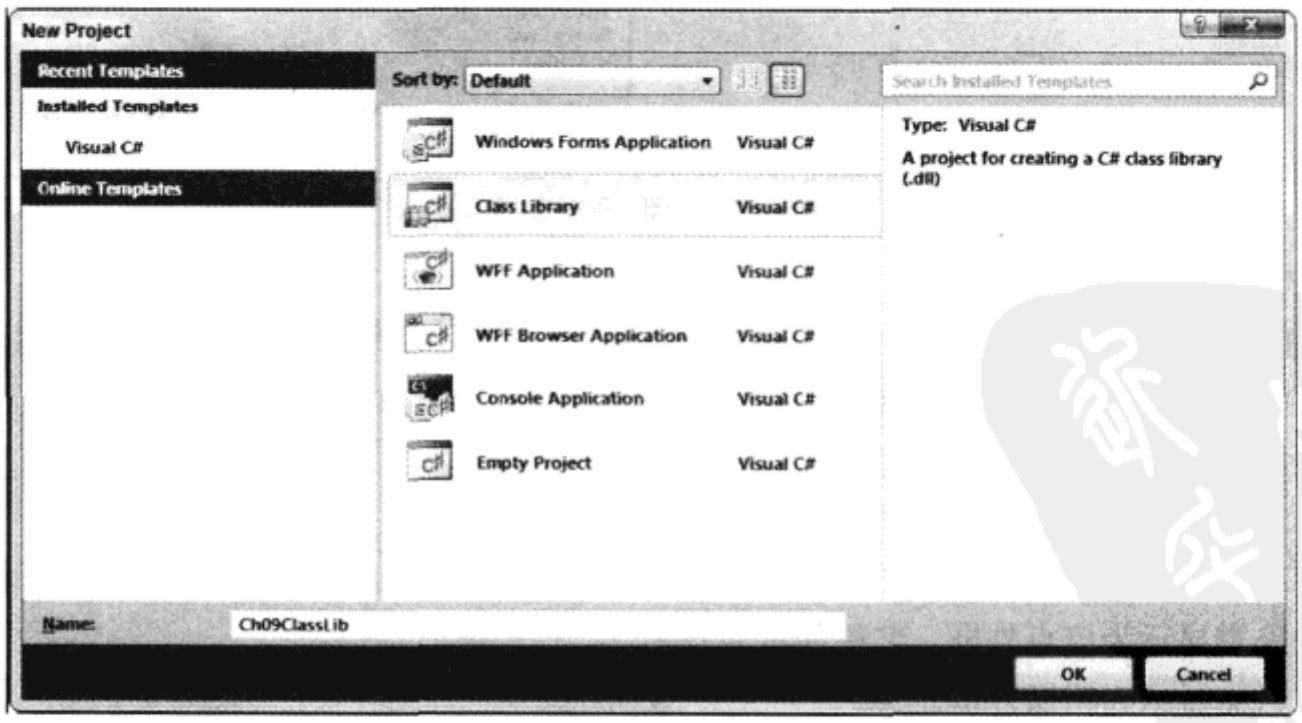


图　9-10

(2) 把文件 Class1.cs 重命名为 MyExternalClass.cs(在 Solution Explorer 窗口中右击该文件，然后选择 Rename 来重命名该文件名)。在弹出的对话框中单击 Yes。

(3) MyExternalClass.cs 中的代码已随之自动改变，以反映类名的改变：

可从 wrox.com 下载源代码

```
public class MyExternalClass
{
}
```

代码段 Ch09ClassLib\MyExternalClass.cs

(4) 使用文件名 MyInternalClass.cs 给项目添加一个新类。

(5) 修改代码，使类 MyInternalClass 成为内部类：

可从 wrox.com 下载源代码

```
internal class MyInternalClass
{
}
```

代码段 Ch09ClassLib\MyInternalClass.cs

(6) 编译项目(注意这个项目没有入口点，所以不能像通常那样运行它——可以选择 Build | Build Solution 菜单项来生成它)。

(7) 在 C:\BegVCSharp\Chapter09 目录中创建一个新的控制台应用程序项目 Ch09Ex02。

(8) 选择 Project | Add Reference 菜单项，或者在 Solution Explorer 窗口中右击 References，选择相同的选项。

(9) 单击 Browse 选项卡，导航到 C:\BegVCSharp\Chapter09\Chapter09\Ch09ClassLib\bin\Debug\，双击 Ch09ClassLib.dll。

(10) 完成了上述操作后，检查引用是否已添加到 Solution Explorer 窗口中，如图 9-11 所示。

图 9-11

(11) 打开 Object Browser 窗口，检查新引用，看看其中包含的对象，其结果如图 9-12 所示。

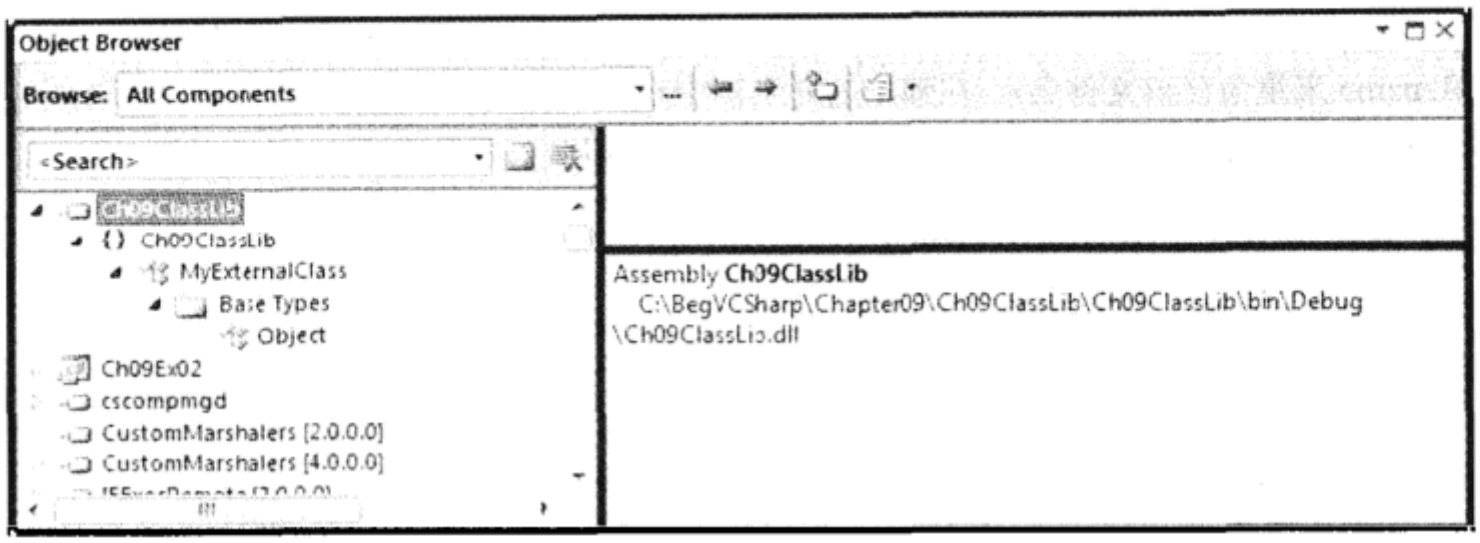


图　9-12

(12) 修改 Program.cs 中的代码，如下所示：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch09ClassLib;

namespace Ch09Ex02
{
   class Program
   {
      static void Main(string[] args)
      {
         MyExternalClass myObj = new MyExternalClass();
         Console.WriteLine(myObj.ToString());
         Console.ReadKey();
      }
   }
}
```

代码段 Ch09Ex02\Program.cs

(13) 运行应用程序，其结果如图 9-13 所示。

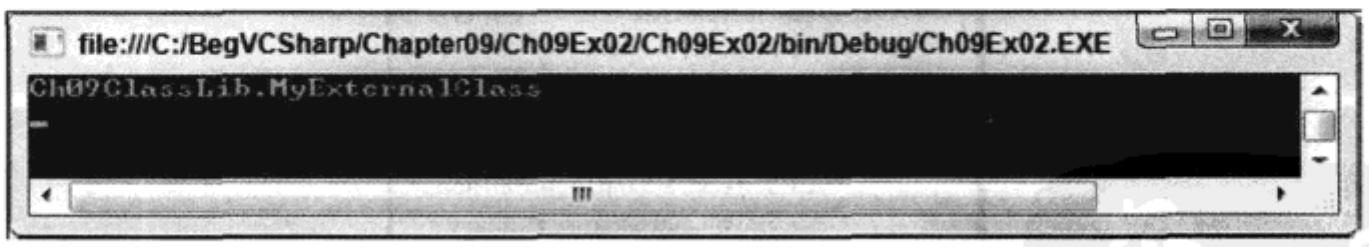


图　9-13

示例的说明

这个示例创建了两个项目，一个是类库项目，另一个是控制台应用程序项目。类库项目 Ch09ClassLib 包含两个类 MyExternalClass(可公开访问)和 MyInternalClass(只能在内部访问)。注意，默认情况下，会隐式将类确定为供内部访问，因为它没有访问修饰符。最好明确指定可访问性，因为这会使代码更容易理解，所以指令增加了 internal 关键字。控制台应用程序项目 Ch09Ex02 包

含利用类库项目的简单代码。

应用程序使用外部库中定义的类时，可以把该应用程序称为库的客户应用程序。使用所定义的类的代码一般简称为客户代码。

为了使用 Ch09ClassLib 中的类，在控制台应用程序中添加了对 Ch09ClassLib.dll 的引用。对于这个示例，该引用是指向类库的输出文件，也可以把这个文件复制到 Ch09Ex02 的本地位置上，以便继续开发类库，而不影响控制台应用程序。为了用新类库项目替换旧版本的程序集，只需用新生成的 DLL 文件覆盖旧文件即可。

在添加了引用后，就可以使用对象浏览器查看可用的类。因为类 MyInternalClass 是内部的，所以在对象浏览器窗口中看不到这个类——它不能由外部的项目访问。但是，MyExternalClass 是可供访问的，这是我们在控制台应用程序中使用的类。

可以把控制台应用程序中的代码替换为使用内部类的代码，如下所示：

```
static void Main(string[] args)
{
   MyInternalClass myObj = new MyInternalClass();
   Console.WriteLine(myObj.ToString());
   Console.ReadKey();
}
```

如果试图编译这段代码，就会产生如下编译错误：

```
'Ch09ClassLib.MyInternalClass' is inaccessible due to its protection level
```

利用外部程序集中的类的技术是使用 C#和.NET Framework 编程的关键。实际上，使用.NET Framework 中的任何类，也就是在利用外部程序集中的类，因为它们的处理方式是相同的。

9.6 接口和抽象类

本章介绍了如何创建接口和抽象类(现在不考虑其成员，第 10 章会讲述类的成员)。这两种类型在许多方面都很类似，所以应看看它们的相似和不同之处，看看哪些情况应使用什么技术。

首先讨论它们的类似之处。抽象类和接口都包含可以由派生类继承的成员。接口和抽象类都不能直接实例化，但可以声明这些类型的变量。如果这样做，就可以使用多态性把继承这两种类型的对象指定给它们的变量。接着通过这些变量来使用这些类型的成员，但不能直接访问派生对象的其他成员。

下面看看它们的区别。派生类只能继承一个基类，即只能直接继承一个抽象类(但可以用一个继承链包含多个抽象类)。相反，类可以使用任意多个接口。但这不会产生太大的区别——这两种情况取得的效果是类似的。只是采用接口的方式略有不同。

抽象类可以拥有抽象成员(没有代码体，且必须在派生类中实现，否则派生类本身必须也是抽象的)和非抽象成员(它们拥有代码体，也可以是虚拟的，这样就可以在派生类中重写)。另一方面，接口成员必须都在使用接口的类上实现——它们没有代码体。另外，按照定义，接口成员是公共的(因

为它们倾向于在外部使用)，但抽象类的成员可以是私有的(只要它们不是抽象的)、受保护的、内部的或受保护的内部成员(其中受保护的内部成员只能在应用程序的代码或派生类中访问)。此外，接口不能包含字段、构造函数、析构函数、静态成员或常量。

抽象类主要用作对象系列的基类，共享某些主要特性，例如，共同的目的和结构。接口则主要用于类，这些类在基础水平上有所不同，但仍可以完成某些相同的任务。

例如，假定有一个对象系列表示火车，基类 Train 包含火车的核心定义，例如车轮的规格和引擎的类型(可以是蒸汽发动机、柴油发动机等)。但这个类是抽象的，因为并没有“一般的”火车。为了创建一辆实际的火车，需要给该火车添加特性。为此，派生一些类，例如：PassengerTrain、FreightTrain 和 424DoubleBogey 等，如图 9-14 所示。

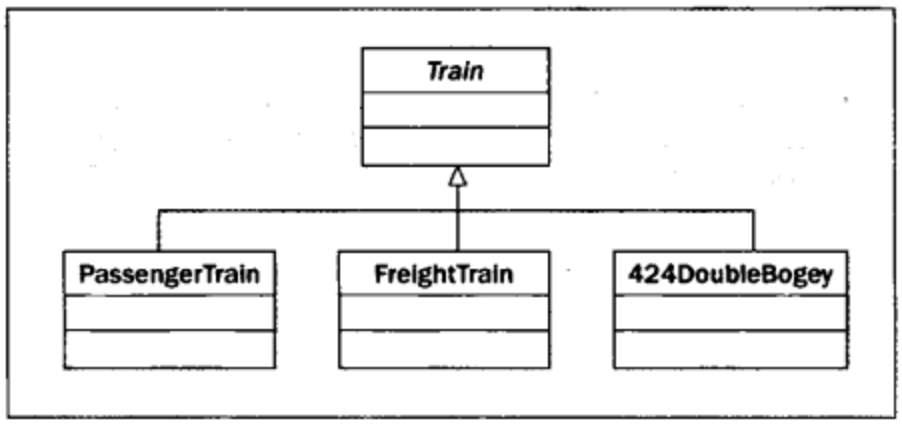


图　9-14

也可以用相同的方式来定义汽车对象系列，使用 Car 抽象基类，其派生类有 Compact、SUV 和 PickUp.Car 和Train 甚至可以派生于一个相同的基类 Vehicle，如图 9-15 所示。

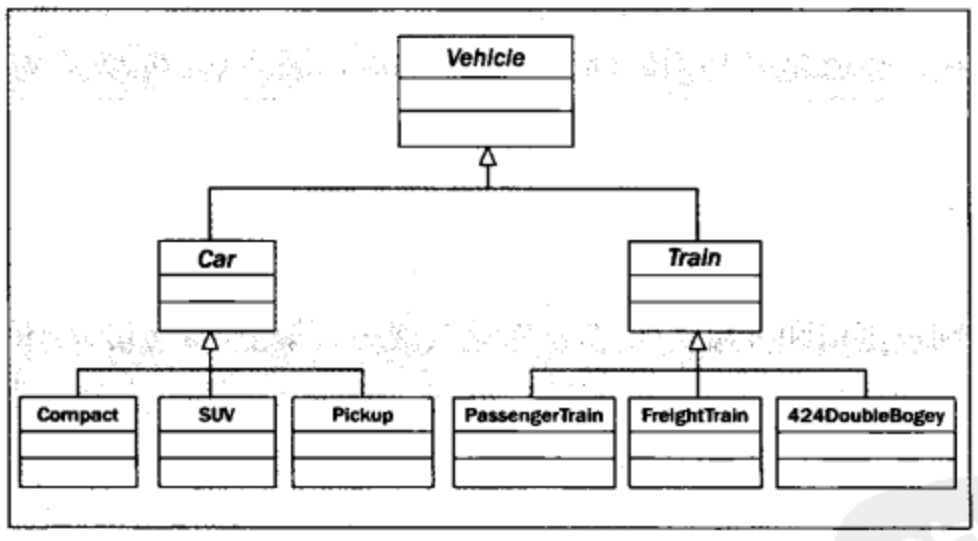


图　9-15

现在，层次结构中的一些类共享相同的特性，这是因为它们的目的是相同的，而不是因为它们派生于相同的基类。例如，PassengerTrain、Compact、SUV 和 Pickup 都可以运送乘客，所以它们都拥有 IPassengerCarrier 接口，FreightTrain 和 Pickup 可以运送货物，所以它们都拥有 IHeavyLoadCarrier 接口，如图 9-16 所示。

在进行更详细的细分前，把对象系统以这种方式进行分解，可以清晰地看到哪种情形适合使用抽象类，哪种情形适合使用接口。只使用接口或只使用抽象继承，就得不到这个示例的结果。

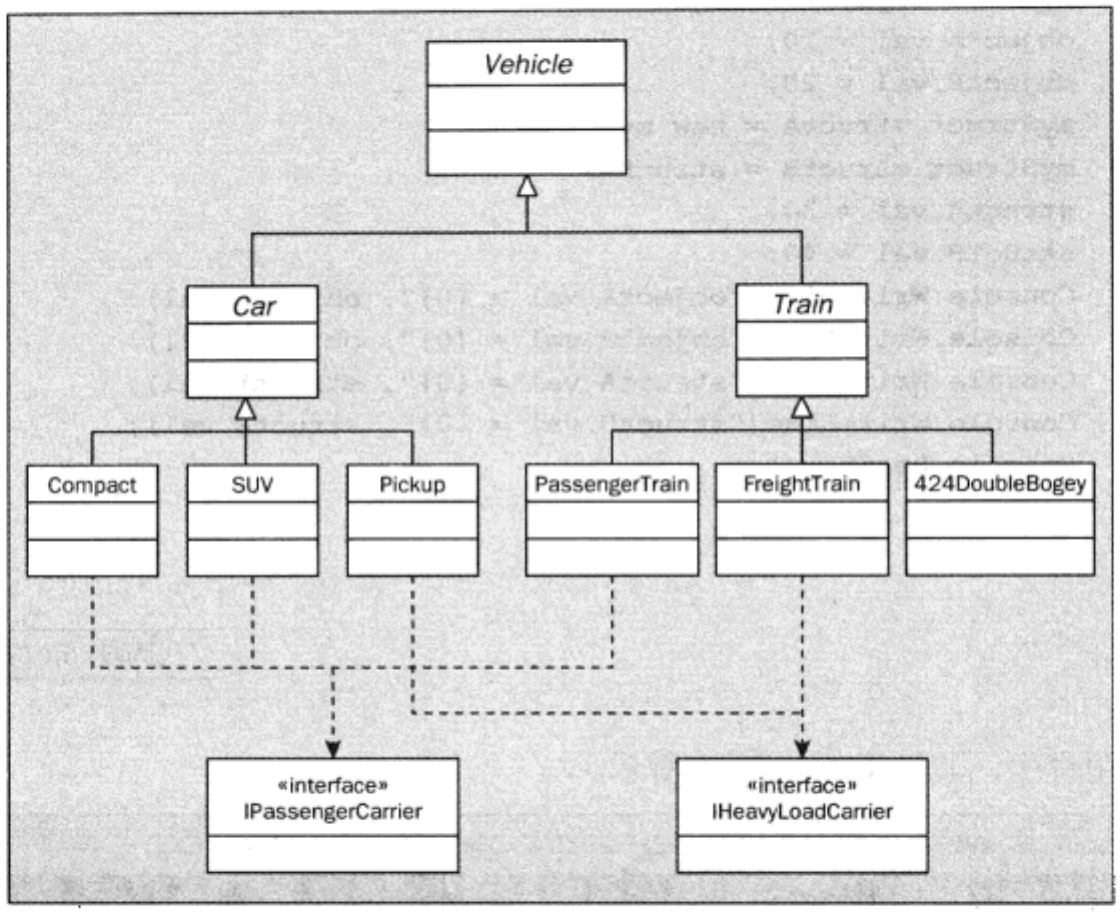


图 9-16

9.7 结构类型

第 8 章提到过结构和类非常相似，但结构是值类型，而类是引用类型。这意味着什么？最简明的方式是用一个示例来说明。

试一试：类和结构

(1) 在 C:\BegVCSharp\Chapter09 目录中创建一个新控制台应用程序项目 Ch09Ex03。

(2) 修改代码，如下所示：

可从 wrox.com 下载源代码

```
namespace Ch09Ex03
{
   class MyClass
   {
      public int val;
   }

   struct myStruct
   {
      public int val;
   }

   class Program
   {
      static void Main(string[] args)
      {
         MyClass objectA = new MyClass();
```

```
            MyClass objectB = objectA;
            objectA.val = 10;
            objectB.val = 20;
            myStruct structA = new myStruct();
            myStruct structB = structA;
            structA.val = 30;
            structB.val = 40;
            Console.WriteLine("objectA.val = {0}", objectA.val);
            Console.WriteLine("objectB.val = {0}", objectB.val);
            Console.WriteLine("structA.val = {0}", structA.val);
            Console.WriteLine("structB.val = {0}", structB.val);
            Console.ReadKey();
        }
    }
}
```

代码段 Ch09Ex03\Program.cs

(3) 运行应用程序，其结果如图 9-17 所示。

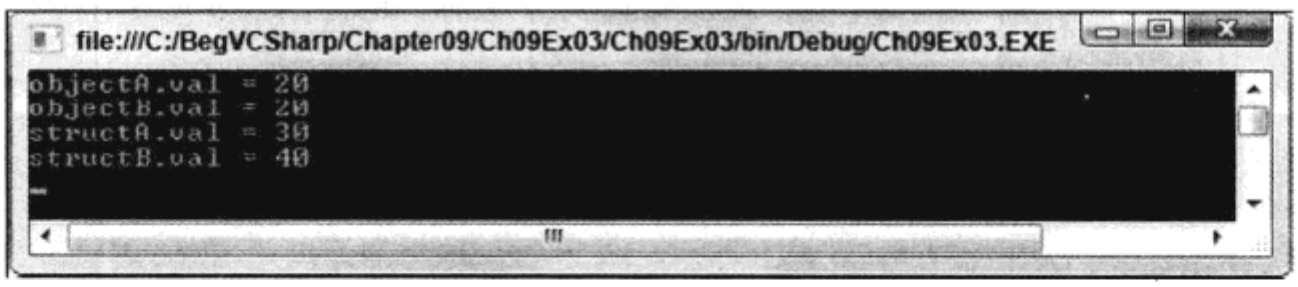


图　9-17

示例的说明

这个应用程序包含两个类型定义。一个是结构 myStruct 的定义，它有一个公共 int 字段 val，另一个是类 MyClass 的定义，它包含一个相同的字段(第 10 章介绍类的成员，如字段，现在只要知道它们的语法是相同的即可)。接着对这两种类型的实例执行相同的操作：

- 声明类型的变量。
- 在这个变量中创建该类型的新实例。
- 声明类型的第二个变量。
- 把第一个变量赋给第二个变量。
- 在第一个变量的实例中，给 val 字段赋一个值。
- 在第二个变量的实例中，给 val 字段赋一个值。
- 显示两个变量的 val 字段值。

尽管对两种类型的变量执行了相同的操作，但结果是不同的。在显示 val 字段的值时，两个 object 类型有相同的值，而结构类型有不同的值。为什么会这样？

对象是引用类型。在把对象赋给变量时，实际上是把带有一个指针的变量赋给了该指针所指向的对象。在实际代码中，指针是内存中的一个地址。在这种情况下，地址是内存中该对象所在的位置。在用下面的代码行把第一个对象引用赋给类型为 MyClass 的第二个变量时，实际上是复制了这个地址。

```
MyClass objectB = objectA;
```

这样两个变量就包含同一个对象的指针。

结构是值类型。其变量并不是包含结构的指针，而是包含结构本身。在用下面的代码把第一个结构赋给类型为 myStruct 的第二个变量时，实际上是把第一个结构的所有信息复制到另一个结构中。

```
myStruct structB = structA;
```

这个过程与本书前面介绍的简单变量类型如 int 是一样的。最终的结果是两个结构类型变量包含不同的结构。使用指针的全部技术隐藏在托管 C#代码中，它使得代码更简单。使用 C#中的不安全代码可以进行低级操作，如指针操作，但这是一个比较高级的论题，这里不予以讨论。

9.8 浅度和深度复制

从一个变量到另一个变量按值复制对象，而不是按引用复制对象(即以与结构相同的方式复制)可能非常复杂。因为一个对象可能包含许多其他对象的引用，例如，字段成员等，这将涉及许多烦琐的操作。把每个成员从一个对象复制到另一个对象中可能不会成功，因为其中一些成员可能是引用类型。

.NET Framework 考虑了这个问题。简单地按照成员复制对象可以通过派生于 System.Object 的 MemberwiseClone()方法来完成，这是一个受保护的方法，但很容易在对象上定义一个调用该方法的公共方法。这个方法提供的复制功能称为浅度复制(shallow copy)，因为它没有考虑引用类型成员。因此，新对象中的引用成员就会指向与源对象中相同成员的对象，在许多情况下这并不理想。如果要创建成员的新实例(复制值，而不复制引用)，此时需要使用深度复制(deep copy)。

可以实现一个 ICloneable 接口，以标准的方式来进行。如果使用这个接口，就必须实现它包含的Clone()方法。这个方法返回一个类型为 System.Object 的值。我们可以采用各种处理方式，执行所选的任何一个方法体得到这个对象。如果愿意，就可以进行深度复制(但执行过程不是必选的，所以可以按照需要执行浅度复制)。详见第 11 章。

9.9 小结

本章讨论了如何在 C#中定义类和接口，把第 8 章的理论以更具体的方式表达出来。我们论述了基本声明所需要的 C#语法和可以使用的可访问关键字，继承接口和其他类的方式，如何定义抽象和密封类以控制这种继承，以及如何定义构造函数和析构函数。

本章介绍了 System.Object，它是我们所定义的所有类的基类。这个类提供了几个方法，其中一些是虚拟的，所以可以重写它们的实现代码。这个类还可以把任何对象实例当作这个类的实例，对任意对象应用多态性。

我们还研究了VS和VCE 为 OOP 开发提供的一些工具，包括 Class View 窗口、Object Browser 窗口，以及给项目添加新类的快速方法。在扩展“多文件”这个概念时，我们还介绍了如何创建程序集，程序集虽然不能运行，但它包含可以在其他项目中使用的类定义。

接着深入探讨了抽象类和接口，理解它们的共同和不同之处，以及使用它们的场合。

最后，讨论了引用类型和值类型，较详细地介绍了结构(对象的值类型)。这引出了浅度复制和深度复制对象的讨论，该主题将在本书的后面再次讨论。

第 10 章将介绍如何定义类成员，如属性和方法，以便在 C#中利用 OOP 创建真正的应用程序。

9.10　练习

(1) 下面的代码存在什么错误？

```
public sealed class MyClass
{
   // Class members.
}

public class myDerivedClass : MyClass
{
   // Class members.
}
```

(2) 如何定义不能创建的类？

(3) 为什么不能创建的类(noncreatable class)仍旧有用？如何利用它们的功能？

(4) 在类库项目 Vehicles 中编写代码，实现本章前面讨论的对象系列 Vehicle，其中有 9 个对象和 2 个接口需要实现。

(5) 创建一个控制台应用程序项目 Traffic，它引用 Vehicles.dll(在第(4)题中创建)，其中包括函数 AddPassenger()，它接受任何带有 IPassengerCarrier 接口的对象。要证明代码可以运行，使用支持这个接口的每个对象实例调用该函数，在每个对象上调用派生于System.Object的ToString()方法，并把结果输出到屏幕上。

附录 A 给出了练习答案。

9.11　本章要点

主　题	重 要 概 念
类和接口定义	类用 class 关键字定义，接口用 interface 关键字定义。可以使用 public 和 internal 关键字定义类和接口的可访问性，类可以定义为 abstract 或 sealed，以控制继承性。父类和父接口在一个用逗号分隔的列表中指定，放在类或接口名和一个冒号的后面。在类定义中，只能指定一个父类，且必须是列表中的第一项
构造函数和析构函数	类自动带有默认的构造函数和析构函数的实现代码，我们很少需要提供自己的析构函数。可以使用可访问性、类名和可能需要的任何参数来定义构造函数。基类的构造函数在派生类的构造函数之前执行，使用 this 和 base 构造函数初始化器关键字，可以控制类中这些构造函数的执行顺序
类库	可以创建只包含类定义的类库项目。这些项目不能直接执行，而必须通过客户代码在可执行程序中访问。VS 和 VCE 为创建、修改和测试类提供了各种工具
类系列	类可以组合为系列，以提供公共的操作或共享公共的特性。为此，可以从共享的基类(可以是抽象的)中继承，或者实现接口
结构定义	结构的定义方式与类非常类似，但结构是值类型，而类是引用类型
复制对象	复制对象时，必须注意应复制该对象包含的其他对象，而不是仅复制这些对象的引用。复制引用称为浅度复制，而完全复制称为深度复制。可以使用 ICloneable 接口作为一个框架，来提供类定义中的深度复制功能

第10章

定义类成员

本章内容：

- 如何定义类成员
- 如何使用类图添加成员
- 如何控制类成员的继承
- 如何定义嵌套的类
- 如何实现接口
- 如何使用部分类定义
- 如何使用 Call Hierarchy 窗口

本章继续讨论在 C#中如何定义类，主要介绍的是如何定义字段、属性和方法等类成员。首先介绍每种类型需要的代码，以及如何使用向导生成相应代码的结构。我们还将论述如何通过编辑成员的属性，来快速修改这些成员。

在介绍完成员定义的基础知识后，将讨论一些比较高级的成员技术：隐藏基类成员、调用重写的基类成员、嵌套的类型定义和部分类定义。

最后将理论付诸实践，创建一个类库，以便在后面的章节中使用它。

10.1 成员定义

在类定义中，也提供了该类中所有成员的定义，包括字段、方法和属性。所有成员都有自己的访问级别，用下面的关键字之一来定义：

- public——成员可以由任何代码访问。
- private——成员只能由类中的代码访问(如果没有使用任何关键字，就默认使用这个关键字)。
- internal——成员只能由定义它的程序集(项目)内部的代码访问。
- protected——成员只能由类或派生类中的代码访问。

后两个关键字可以合并使用，所以也有 protected internal 成员。它们只能由项目(更确切地讲，

是程序集)中派生类的代码来访问。

也可以使用关键字static 来声明字段、方法和属性，这表示它们是类的静态成员，而不是对象实例的成员，详见第 8 章。

10.1.1　定义字段

字段用标准的变量声明格式和前面介绍的修饰符来定义(可以进行初始化)，例如：

```
class MyClass
{
   public int MyInt;
}
```

.NET Framework 中的公共字段以 PascalCasing 形式来命名，而不是 camelCasing。这里使用的就是这种命名方法。这就是上面的字段叫作 MyInt 而不是 myInt 的原因。这仅是推荐使用的命名模式之一，但它的意义非常重大。私有字段没有推荐的命名模式，它们通常使用 camelCasing 来命名。

字段也可以使用关键字readonly，表示这个字段只能在执行构造函数的过程中赋值，或由初始化赋值语句赋值。例如：

```
class MyClass
{
   public readonly int MyInt = 17;
}
```

如本章的导言所述，字段可以使用 static 关键字声明为静态，例如：

```
class MyClass
{
   public static int MyInt;
}
```

静态字段必须通过定义它们的类来访问(在上面的示例中，是 MyClass.MyInt)，而不是通过这个类的对象实例来访问。另外，可以使用关键字 const 来创建一个常量。按照定义，const 成员也是静态的，所以不需要用 static 修饰符(实际上，用 static 修饰符会产生一个错误)。

10.1.2　定义方法

方法使用标准函数格式、可访问性和可选的 static 修饰符来声明。例如：

```
class MyClass
{
   public string GetString()
   {
      return "Here is a string.";
   }
}
```

与公共字段一样，.NET Framework 中的公共方法也采用 PascalCasing 形式来命名。

注意，如果使用了static 关键字，这个方法就只能通过类来访问，不能通过对象实例来访问。也可以在方法定义中使用下述关键字：

- virtual——方法可以重写。
- abstract——方法必须在非抽象的派生类中重写(只用于抽象类中)。
- override——方法重写了一个基类方法(如果方法被重写，就必须使用该关键字)。
- extern——方法定义放在其他地方。

下面的代码是方法重写的一个示例：

```
public class MyBaseClass
{
   public virtual void DoSomething()
   {
      // Base implementation.
   }
}

public class MyDerivedClass : MyBaseClass
{
   public override void DoSomething()
   {
      // Derived class implementation, overrides base implementation.
   }
}
```

如果使用了override，也可以使用 sealed 指定在派生类中不能对这个方法作进一步的修改，即这个方法不能由派生类重写。例如：

```
public class MyDerivedClass : MyBaseClass
{
   public override sealed void DoSomething()
   {
      // Derived class implementation, overrides base implementation.
   }
}
```

使用 extern 可以在项目外部提供方法的实现代码。这是一个高级论题，这里不做详细讨论。

10.1.3　定义属性

属性定义方式与字段类似，但包含的内容比较多。如前所述，属性涉及的内容比字段多，是因为它们在修改状态前还可以执行一些额外的操作，实际上，它们可能并不修改状态。属性拥有两个类似于函数的块，一个块用于获取属性的值，另一个块用于设置属性的值。

这两个块也称为访问器，分别用 get 和 set 关键字来定义，可以用于控制对属性的访问级别。可以忽略其中的一个块来创建只读或只写属性(忽略 get 块创建只写属性，忽略 set 块创建只读属性)。

当然，这仅适用于外部代码，因为类中的其他代码可以访问这些代码块能访问的数据。还可以在访问器上包含可访问修饰符，例如使 get 块变成公共的，把 set 块变成受保护的。只有包含其中一个一个块，才能获得有效属性(既不能读取也不能修改的属性没有任何用处)。

属性的基本结构包括标准的可访问修饰符(public、private 等)，后跟类名、属性名和 get 块(或 set 块，或者 get 块和 set 块，其中包含属性处理代码)，例如：

```
public int MyIntProp
{
   get
   {
      // Property get code.
   }
   set
   {
      // Property set code.
   }
}
```

.NET 中的公共属性也以 PascalCasing 方式来命名，而不是 camelCasing 方式命名，与字段和方法一样，这里使用 PascalCasing 方式。

定义代码中的第一行非常类似于定义字段的代码。区别是行末没有分号，而是一个包含嵌套 get 和 set 块的代码块。

get块必须有一个属性类型的返回值，简单的属性一般与私有字段相关联，以控制对这个字段的访问，此时 get 块可以直接返回该字段的值，例如：

```
// Field used by property.
private int myInt;

// Property.
public int MyIntProp
{
   get
   {
      return myInt;
   }
   set
   {
      // Property set code.
   }
}
```

类外部的代码不能直接访问这个 myInt 字段，因为其访问级别是私有的。外部的代码必须使用属性来访问该字段。set 函数以类似的方式把一个值赋给字段。这里可以使用关键字 value 表示用户提供的属性值：

```
// Field used by property.
private int myInt;
```

```
// Property.
public int MyIntProp
{
   get
   {
      return myInt;
   }
   set
   {
      myInt = value;
   }
}
```

value 等于类型与属性相同的一个值，所以如果属性和字段使用相同的类型，就不必担心数据类型转换了。

这个简单的属性只能直接访问 myInt 字段。在对操作进行更多的控制时，属性的真正作用才能发挥出来。例如，使用下面的代码实现 set 块：

```
set
{
   if (value >= 0 && value <= 10)
      myInt = value;
}
```

只有赋给属性的值在 0~10 之间，才会改 myInt。此时，要做一个重要的设计选择：如果使用了无效值，该怎么办？有 4 种选择：

- 什么也不做(如上述代码所示)。
- 给字段赋默认值。
- 继续执行，就好像没有发生错误一样，但记录下该事件，以备将来分析。
- 抛出异常。

一般情况下，后两个选择效果较好，选择哪个选项取决于如何使用类，以及给类的用户授予多少控制权。抛出异常给用户提供的控制权相当大，可以让他们知道发生了什么情况，并作出适当的响应。为此可以使用 System 名称空间中的标准异常，例如：

```
set
{
   if (value >= 0 && value <= 10)
      myInt = value;
   else
      throw (new ArgumentOutOfRangeException("MyIntProp", value,
                "MyIntProp must be assigned a value between 0 and 10."));
}
```

这可以在使用属性的代码中通过 try…catch…finally 逻辑来处理，详见第 7 章。

记录数据，例如，记录到文本文件中，对产品代码会比较有效，因为产品代码不应发生错误。它们允许开发人员检查性能，如有必要，还可以调试现有的代码。

属性可以使用 virtual、override 和abstract 关键字，就像方法一样，但这几个关键字不能用于字段。最后，如上所述，访问器可以有自己的可访问性，例如：

```
// Field used by property.
private int myInt;

// Property.
public int MyIntProp
{
   get
   {
      return myInt;
   }
   protected set
   {
      myInt = value;
   }
}
```

只有类或派生类中的代码才能使用 set 访问器。

访问器可以使用的访问修饰符取决于属性的可访问性，访问器的可访问性不能高于它所属的属性，也就是说，私有属性对它的访问器不能包含任何可访问修饰符，而公共属性可以对其访问器使用所有的可访问修饰符。下面的示例中将定义和使用字段、方法和属性。

试一试：使用字段、方法和属性

(1) 在 C:\BegVCSharp\Chapter10 目录中创建一个新控制台应用程序项目 Ch10Ex01。

(2) 使用 Add Class 快捷方式添加一个新类 MyClass，这将在新文件 MyClass.cs 中定义这个新类。

(3) 修改 MyClass.cs 中的代码，如下所示：

可从 wrox.com 下载源代码

```
public class MyClass
{
   public readonly string Name;
   private int intVal;

   public int Val
   {
      get
      {
         return intVal;
      }
      set
      {
         if (value >= 0 && value <= 10)
            intVal = value;
         else
            throw (new ArgumentOutOfRangeException("Val", value,
               "Val must be assigned a value between 0 and 10."));
      }
   }
   public override string ToString()
   {
      return "Name: " + Name + "\nVal: " + Val;
   }
   private MyClass() : this("Default Name")
   {
```

```
    }
    public MyClass(string newName)
    {
       Name = newName;
       intVal = 0;
    }
}
```

代码段 Ch10Ex01\MyClass.cs

(4) 修改 Program.cs 中的代码，如下所示：


可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   Console.WriteLine("Creating object myObj...");
   MyClass myObj = new MyClass("My Object");
   Console.WriteLine("myObj created.");
   for (int i = -1; i <= 0; i++)
   {
      try
      {
         Console.WriteLine("\nAttempting to assign {0} to myObj.Val...",
                           i);
         myObj.Val = i;
         Console.WriteLine("Value {0} assigned to myObj.Val.", myObj.Val);
      }
      catch (Exception e)
      {
         Console.WriteLine("Exception {0} thrown.", e.GetType().FullName);
         Console.WriteLine("Message:\n\"{0}\"", e.Message);
      }
   }
   Console.WriteLine("\nOutputting myObj.ToString()...");
   Console.WriteLine(myObj.ToString());
   Console.WriteLine("myObj.ToString() Output.");
   Console.ReadKey();
}
```

代码段 Ch10Ex01\Program.cs

(5) 运行应用程序，其结果如图 10-1 所示。

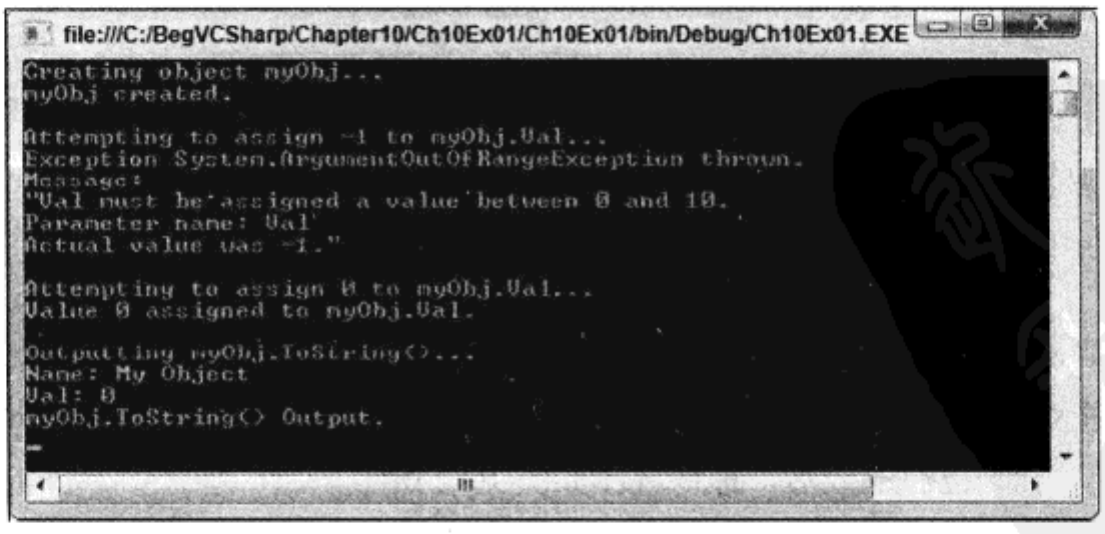


图　10-1

示例的说明

Main()中的代码创建并使用在 MyClass.cs 中定义的 MyClass 类的实例。实例化这个类必须使用非默认的构造函数来进行，因为 MyClass 类的默认构造函数是私有的：

```
private MyClass() : this("Default Name")
{
}
```

注意，这里用 this("Default Name")来保证，如果调用了该构造函数，Name 就获取一个值。如果这个类用于派生一个新类，这就是可能的。这是必须的，因为不给 Name 字段赋值，就会在后面产生错误。

所使用的非默认构造函数把值赋给只读字段 name(只能在字段声明或在构造函数中给它赋值)和私有字段 intVal。

接着，Main()试着给 myObj(MyClass 的实例)的 Val 属性赋值。for 循环在两次循环中赋值–1 和 0，try...catch 结构用于检查抛出的异常。把–1 赋给属性时，会抛出 System.ArgumentOutOf RangeException 类型的异常，catch 块中的代码会把该异常的信息输出到控制台窗口中。在下一个循环中，值 0 成功地赋给了Val 属性，通过这个属性再把值赋给私有字段 intVal。

最后，使用重写的 ToString()方法输出一个格式化的字符串，来表示对象的内容：

```
public override string ToString()
{
   return "Name: " + Name + "\nVal: " + Val;
}
```

必须使用 override 关键字来声明这个方法，因为它重写了基类 System.Object 的虚拟方法 ToString()。此处的代码直接使用属性 Val，而不是私有字段 intVal，没有理由不以这种方式使用类中的属性，但这可能会对性能产生比较轻微的影响(对性能的影响非常小，我们不可能察觉到)。当然，使用属性也可以在属性中进行固有的有效性验证，这对类中的代码也是有好处的。

10.1.4　在类图中添加成员

第 9 章介绍了如何使用类图研究项目中的类，还提到类图可以用于添加成员，本节就介绍这些内容。

类图功能只能在 VS 中使用，不能在 VCE 中使用。

添加和编辑成员的所有工具都显示在 Class Diagram 视图的 Class Details 窗口中。要查看这个窗口，可以为 Ch10Ex01 中的 MyClass 类创建一个类图。在类设计器中扩展类的视图(单击两个向下箭头的图标)，就可以看到已有的成员，最终视图如图 10-2 所示。

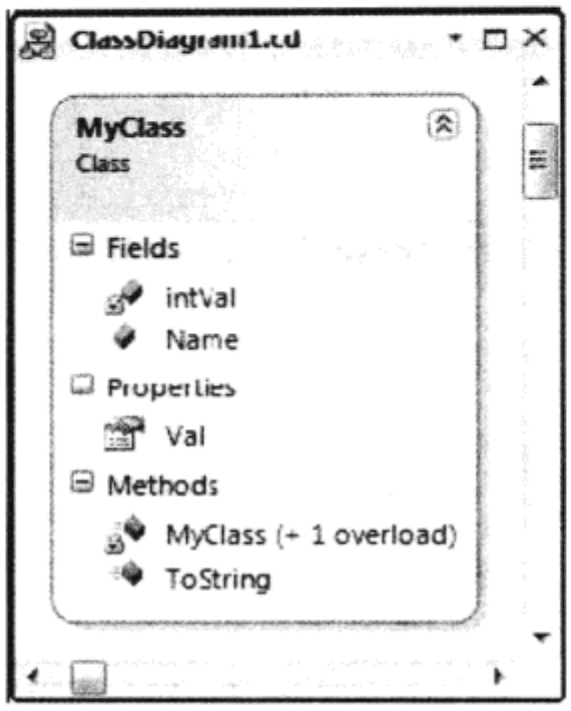


图　10-2

在 Class Details 窗口中选中类，就可以看到如图 10-3 所示的信息。

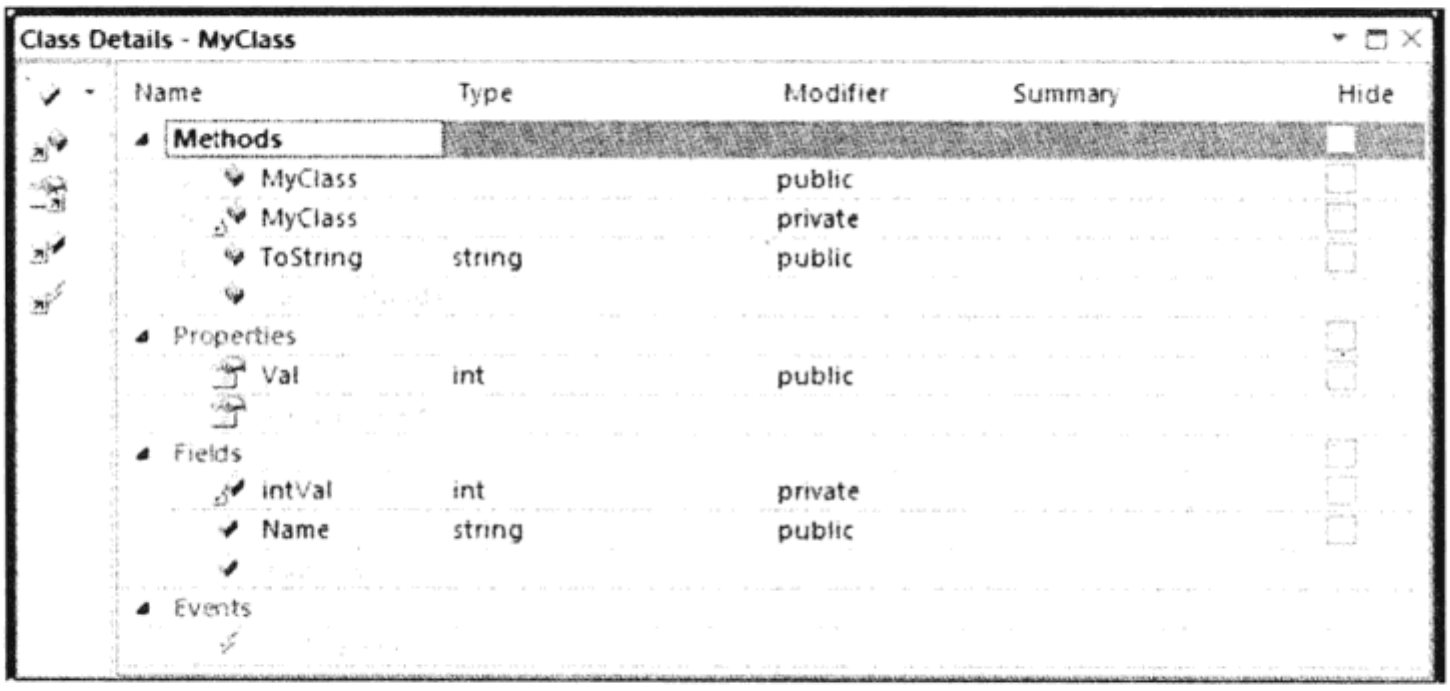


图　10-3

其中显示了当前为类定义的所有成员，并允许在相关的空间键入信息，添加新成员。

1. 添加方法

在<add method>框中键入一个方法，就可以把这个方法添加到类中。给方法命名后，就可以使用 Tab 键导航到后续的设置，从方法的返回类型开始，然后是方法的可访问性、汇总信息(它们会转换为 XML 文档说明)、是否在类图中隐藏该方法等设置。

添加好方法后，就可以按相同的方式扩展各项，添加参数。对于参数，也可以使用修饰符 out、ref 和 params。新方法的一个示例如图 10-4 所示。

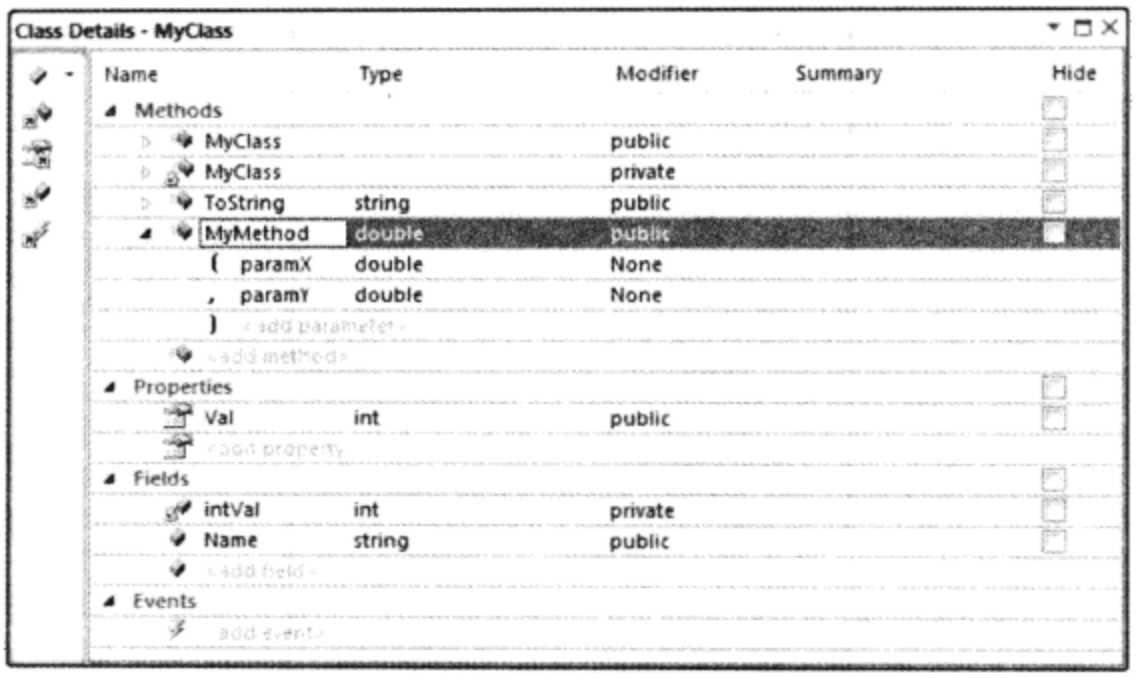


图　10-4

这个新方法在类中添加了如下代码：

```
public double MyMethod(double paramX, double ParamY)
{
   Throw new System.NotImplementedException();
}
```

方法的其他配置可以在 Properties 窗口中完成，如图 10-5 所示。

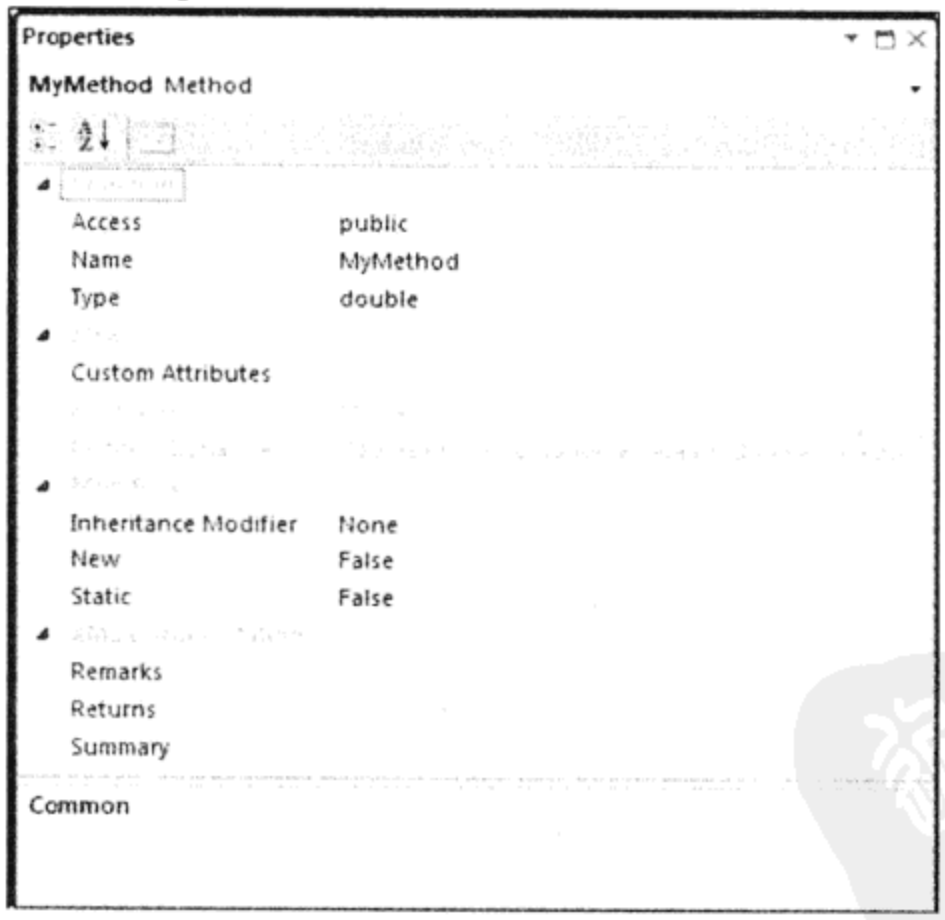


图　10-5

在这个窗口中可以把方法设置为静态的。显然，这种技术不能提供方法的实现代码，但提供了基本结构，肯定可以减少键入错误！

2. 添加属性

可以采用相同的方式添加属性。图 10-6 显示了使用 Class Details 窗口添加的新属性。

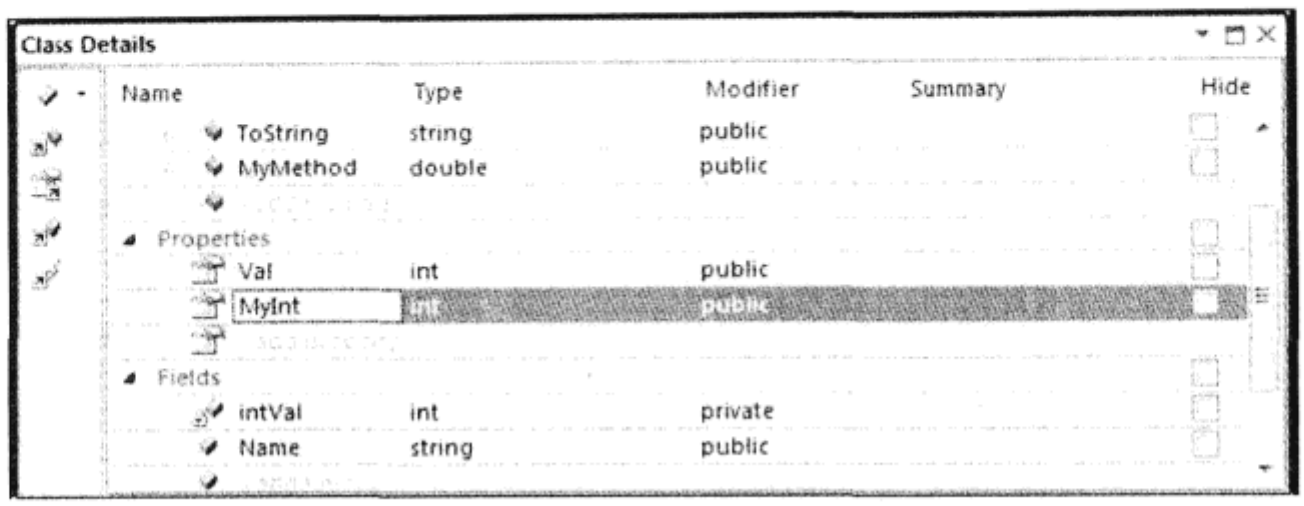


图　10-6

这会添加如下属性：

```
public int myInt
{
   get
   {
      throw new System.NotImplementedException();
   }
   set {}
}
```

注意，该窗口没有提供完整的实现代码，您需要自己去完成，包括为简单的属性匹配一个带字段的属性，删除访问器(把属性设置为只读或只写的)，或者给访问器应用可访问修饰符。该窗口提供了基本结构。

3. 添加字段

添加字段是很简单的。只需键入字段的名称，选择类型和访问修饰符即可。

10.1.5　重构成员

在添加属性时有一项很方便的技术，可以从字段中生成属性，下面是一个重构(refactoring)的示例，“重构”表示使用工具修改代码，而不是手工修改。为此，只需右击类图中的某个成员，或者在代码视图中右击某个成员即可。

VCE 包含有限的重构功能，但不包含这里介绍的字段封装功能。在这方面，VS 提供的选项要远远多于VCE。

例如，如果 MyClass 类包含如下字段：

```
public string myString;
```

右击该字段，选择 Refactor | Encapsulate Field，就会打开如图 10-7 所示的对话框。

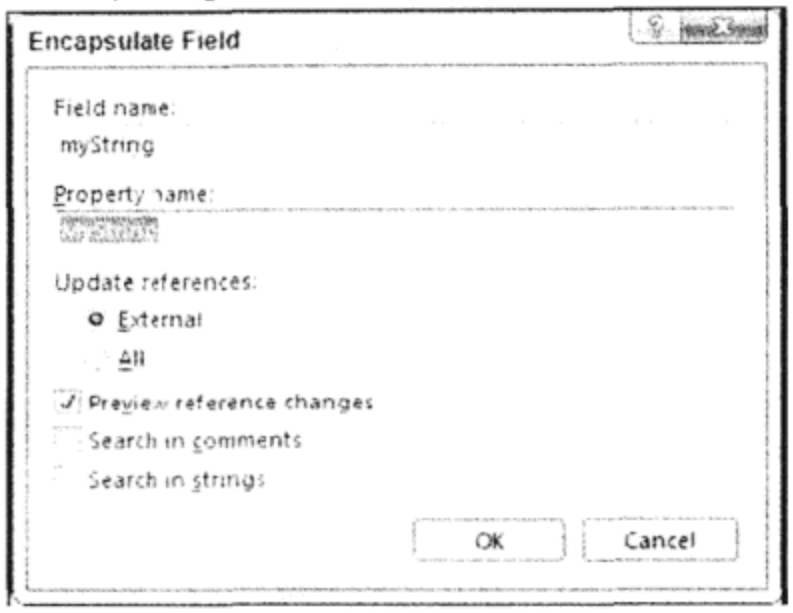


图　10-7

接受默认选项，就会修改 MyClass 的代码，如下所示：

```
Private string myString;
public string MyString;
{
   get
   {
       Return myString;
   }
   set
   {
      myString = value;
   }
}
```

myString 字段的可访问性变成了 private，同时创建了一个公共属性 MyString，它自动链接到 myString 上。这会减少单纯为字段创建属性的时间。

10.1.6　自动属性

属性是访问对象状态的首选方式，因为它们禁止外部代码实现对象内部的数据存储机制。属性还对内部数据的访问方式施加了更多的控制，本章代码在多处体现了这一点。但是，一般以非常标准的方式定义属性，即通过一个公共属性来直接访问一个私有成员。其代码非常类似于上一节的代码，这是 VS 重构工具自动生成的。

重构功能肯定加快了键入速度，C#还为此提供了另一种方式：自动属性。利用自动属性，可以用简化的语法声明属性，C#编译器会自动添加未键入的内容。具体而言，编译器会声明一个用于存储属性的私有字段，并在属性的 get 和 set 块中使用该字段，我们无需考虑细节。

使用下面的代码结构就可以定义一个自动属性：

```
public int MyIntProp
{
    get;
    set;
}
```

甚至可以在一行代码上定义自动属性，以便节省空间，而不会过度地降低属性的可读性：

```
public int MyIntProp { get; set;}
```

我们按照通常的方式定义属性的可访问性、类型和名称，但没有给 get 和 set 块提供实现代码。这些块的实现代码(和底层的字段)都由编译器提供。

使用自动属性时，只能通过属性访问数据，不能通过底层的私有字段来访问，因为我们不知道底层私有字段的名称(该名称是在编译期间定义的)。但这并不是一个真正意义上的限制，因为可以直接使用属性名。自动属性的唯一限制是它们必须包含 get 和 set 存取器，无法使用这种方式定义只读或只写属性。

10.2　类成员的其他议题

下面该讨论一些比较高级的成员议题了。本节主要研究：

- 隐藏基类方法
- 调用重写或隐藏的基类方法
- 嵌套的类型定义

10.2.1　隐藏基类方法

当从基类继承一个(非抽象的)成员时，也就继承了其实现代码。如果继承的成员是虚拟的，就可以用override关键字重写这段实现代码。无论继承的成员是否为虚拟，都可以隐藏这些实现代码。这是很有用的，例如，当继承的公共成员不像预期的那样工作时，就可以隐藏它。

使用下面的代码就可以隐藏：

```
public class MyBaseClass
{
   public void DoSomething()
   {
      // Base implementation.
   }
}

public class MyDerivedClass : MyBaseClass
{
   public void DoSomething()
   {
      // Derived class implementation, hides base implementation.
   }
}
```

尽管这段代码正常运行，但它会产生一个警告，说明隐藏了一个基类成员。如果是无意间隐藏了一个需要使用的成员，此时就可以改正错误。如果确实要隐藏该成员，就可以使用new 关键字显式地表明意图：

```
public class MyDerivedClass : MyBaseClass
{
```

```
    new public void DoSomething()
    {
        // Derived class implementation, hides base implementation.
    }
}
```

其工作方式是完全相同的，但不会显示警告。此时应注意隐藏基类成员和重写它们的区别。考虑下面的代码：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething()
    {
        Console.WriteLine("Derived imp");
    }
}
```

其中重写方法将替换基类中的实现代码，这样，下面的代码就将使用新版本，即使这是通过基类类型进行的，情况也是这样(使用多态性)：

```
MyDerivedClass myObj = new MyDerivedClass();
MyBaseClass myBaseObj;
myBaseObj = myObj;
myBaseObj.DoSomething();
```

结果如下：

```
Derived imp
```

另外，还可以使用下面的代码隐藏基类方法：

```
public class MyBaseClass
{
    public virtual void DoSomething()
    {
        Console.WriteLine("Base imp");
    }
}

public class MyDerivedClass : MyBaseClass
{
    new public void DoSomething()
    {
        Console.WriteLine("Derived imp");
    }
}
```

基类方法不必是虚拟的，但结果是一样的，只需修改上面代码中的一行即可。对于基类的虚拟方法和非虚拟方法来说，其结果如下：

```
Base imp
```

尽管隐藏了基类的实现代码，但仍可以通过基类访问它。

10.2.2　调用重写或隐藏的基类方法

无论是重写成员还是隐藏成员，都可以在派生类的内部访问基类成员。这在许多情况下都是很有用的，例如：

- 要对派生类的用户隐藏继承的公共成员，但仍能在类中访问其功能。
- 要给继承的虚拟成员添加实现代码，而不是简单地用重写的新执行代码替换它。

为此，可以使用base 关键字，它表示包含在派生类中的基类的实现代码(在控制构造函数时，其用法是类似的，如第 9 章所述)，例如：

```
public class MyBaseClass
{
   public virtual void DoSomething()
   {
      // Base implementation.
   }
}

public class MyDerivedClass : MyBaseClass
{
   public override void DoSomething()
   {
      // Derived class implementation, extends base class implementation.
      base.DoSomething();
      // More derived class implementation.
   }
}
```

这段代码执行包含在MyBaseClass 中的 DoSomething()版本，MyBaseClass 是 MyDerivedClass 的基类，而 DoSomething()版本包含在 MyDerivedClass 中。因为 base 使用的是对象实例，所以在静态成员中使用它会产生错误。

this 关键字

除了使用第 9 章的 base 关键字外，还可以使用 this 关键字。与 base 一样，this 也可以用在类成员的内部，且该关键字也引用对象实例。只是 this 引用的是当前的对象实例(即不能在静态成员中使用 this 关键字，因为静态成员不是对象实例的一部分)。

this 关键字最常用的功能是把当前对象实例的引用传递给一个方法，如下例所示：

```
public void doSomething()
{
   MyTargetClass myObj = new MyTargetClass();
   myObj.DoSomethingWith(this);
}
```

其中，被实例化的 MyTargetClass 实例有一个 DoSomethingWith()方法，该方法带一个参数，其

类型与包含上述方法的类兼容。这个参数类型可以是类的类型、由这个类继承的类类型，或者由这个类或 System.Object 实现的一个接口。

this 关键字的另一个常见用法是限定本地类型的成员，例如：

```
public class MyClass
{
   private int someData;

   public int SomeData
   {
      get
      {
         return this.someData;
      }
   }
}
```

许多开发人员都喜欢这个语法，它可以用于任意成员类型，因为可以一眼看出引用的是成员，而不是局部变量。

10.2.3 嵌套的类型定义

除了在名称空间中定义类型之外，还可以在其他类中定义这些类。如果这么做，就可以在定义中使用各种访问修饰符，而不仅仅是 public 和internal，也可以使用 new 关键字隐藏继承于基类的类型定义。例如，下面的代码定义了 MyClass，也定义了一个嵌套的类 myNestedClass：

```
public class MyClass
{
   public class myNestedClass
   {
      public int nestedClassField;
   }
}
```

如果要在 MyClass 的外部实例化 myNestedClass，就必须限定名称，例如：

```
MyClass.myNestedClass myObj = new MyClass.myNestedClass();
```

但是，如果嵌套的类声明为私有，或者声明为其他与执行该实例化的代码不兼容的访问级别，就不能这么做。这个功能主要用于定义对于其包含类来说是私有的类，这样，名称空间中的其他代码就不能访问它。

10.3 接口的实现

在继续前，先讨论一下如何定义和实现接口。第 9 章介绍了接口定义的方式与类相似，使用的代码如下：

```
interface IMyInterface
{
   // Interface members.
}
```

接口成员的定义与类成员的定义相似，但有几个重要的区别：

- 不允许使用访问修饰符(public、private、protected 或 internal)，所有的接口成员都是公共的。
- 接口成员不能包含代码体。
- 接口不能定义字段成员。
- 接口成员不能用关键字 static、virtual、abstract 或 sealed 来定义。
- 类型定义成员是禁止的。

但要隐藏继承了基接口的成员，可以用关键字 new 来定义它们，例如：

```
interface IMyBaseInterface
{
   void DoSomething();
}

interface IMyDerivedInterface : IMyBaseInterface
{
   new void DoSomething();
}
```

其执行方式与隐藏继承的类成员的方式一样。

在接口中定义的属性可以定义访问块 get 和 set 中的哪一个能用于该属性(或将它们同时用于该属性)，例如：

```
interface IMyInterface
{
   int MyInt { get; set; }
}
```

其中 int 属性 MyInt 有 get 和 set 存取器。对于访问级别有更严限制的属性来说，可以省略它们中的任一个。

这个语法类似于自动属性，但自动属性是为类(而不是接口)定义的，自动属性必须包含 get 和 set 存取器。

接口没有指定应如何存储属性数据。接口不能指定字段，例如用于存储属性数据的字段。最后，接口与类一样，可以定义为类的成员(但不能定义为其他接口的成员，因为接口不能包含类型定义)。

在类中实现接口

实现接口的类必须包含该接口所有成员的实现代码，且必须匹配指定的签名(包括匹配指定的 get 和set 块)，并且必须是公共的。例如：

```
public interface IMyInterface
{
   void DoSomething();
   void DoSomethingElse();
}
```

```
public class MyClass : IMyInterface
{
   public void DoSomething()
   {
   }

   public void DoSomethingElse()
   {
   }
}
```

可以使用关键字 virtual 或 abstract 来实现接口成员，但不能使用 static 或 const。还可以在基类上实现接口成员，例如：

```
public interface IMyInterface
{
   void DoSomething();
   void DoSomethingElse();
}

public class MyBaseClass
{
   public void DoSomething()
   {
   }
}

public class MyDerivedClass : MyBaseClass, IMyInterface
{
   public void DoSomethingElse()
   {
   }
}
```

继承一个实现给定接口的基类，就意味着派生类隐式地支持这个接口，例如：

```
public interface IMyInterface
{
   void DoSomething();
   void DoSomethingElse();
}

public class MyBaseClass : IMyInterface
{
   public virtual void DoSomething()
   {
   }

   public virtual void DoSomethingElse()
   {
   }
}

public class MyDerivedClass : MyBaseClass
```

```
{
   public override void DoSomething()
   {
   }
}
```

显然，在基类中把实现代码定义为虚拟，派生类就可以替换该实现代码，而不是隐藏它们。如果要使用 new 关键字隐藏一个基类成员，而不是重写它，则方法IMyInterface. DoSomething()就总是引用基类版本，即使通过这个接口来访问派生类，也是这样。

1. 显式实现接口成员

也可以由类显式地实现接口成员。如果这么做，该成员就只能通过接口来访问，不能通过类来访问。上一节的代码中使用的隐式成员可以通过类和接口来访问。

例如，如果类 MyClass 隐式地实现接口 IMyInterface 的方法 DoSomething()，如上所述，则下面的代码就是有效的：

```
MyClass myObj = new MyClass();
myObj.DoSomething();
```

下面的代码也是有效的：

```
MyClass myObj = new MyClass();
IMyInterface myInt = myObj;
myInt.DoSomething();
```

另外，如果 MyDerivedClass 显式实现 DoSomething()，就只能使用后一种技术。其代码如下：

```
public class MyClass : IMyInterface
{
   void IMyInterface.DoSomething()
   {
   }

   public void DoSomethingElse()
   {
   }
}
```

其中DoSomething()是显式实现的，而 DoSomethingElse()是隐式实现的。只有后者可以直接通过 MyClass 的对象实例来访问。

2. 用非公共的可访问性添加属性存取器

前面说过，如果实现带属性的接口，就必须实现匹配的 get/set 存取器。这并不是绝对正确的——如果在定义属性的接口中只包含 set 块，就可给类中的属性添加 get 块，反之亦然。但是，只有所添加的存取器的可访问修饰符比接口中定义的存取器的可访问修饰符更严格时，才能这么做。因为按照定义，接口定义的存取器是公共的，也就是说，只能添加非公共的存取器。例如：

```
public interface IMyInterface
{
```

```
    int MyIntProperty
    {
      get;
    }
}

public class MyBaseClass : IMyInterface
{

    public int MyIntProperty { get; protected set; }
}
```

10.4 部分类定义

如果所创建的类包含一种类型或其他类型的许多成员时，就很容易混淆，代码文件也比较长。这里可以采用前面章节介绍的一种方法，即给代码分组。在代码中定义区域，就可以折叠和展开各个代码区，使代码更便于阅读。例如，有一个类的定义如下：

```
public class MyClass
{
    #region Fields
    private int myInt;
    #endregion

    #region Constructor
    public MyClass()
    {
      myInt = 99;
    }
    #endregion

    #region Properties
    public int MyInt
    {
      get
      {
        return myInt;
      }

      set
      {
        myInt = value;
      }
    }
    #endregion

    #region Methods
    public void DoSomething()
    {
      // Do something...
    }
    #endregion
}
```

上述代码可以展开和折叠类的字段、属性、构造函数和方法，以便集中精力考虑自己感兴趣的内容。甚至可以按这种方式嵌套各个区域，这样一些区域就只能在包含它们的区域被展开后才能看到。

但是，即便使用这种技术，代码也可能难以理解。对此，一种方法是使用部分类定义(partial class definition)。简言之，就是使用部分类定义，把类的定义放在多个文件中。例如，可以把字段、属性和构造函数放在一个文件中，而把方法放在另一个文件中。为此，只需在每个包含部分类定义的文件中对类使用 partial 关键字即可，如下所示：

```
public partial class MyClass
{
   ...
}
```

如果使用部分类定义，partial 关键字就必须出现在包含定义部分的每个文件的与此相同的位置。

部分类对 Windows 应用程序隐藏与窗体布局相关的代码有很大的作用。第 2 章已经介绍了这些内容。在 Form1 类中，Windows 窗体的代码存储在 Form1.cs 和 Form1.Designer.cs 中，这样就可以主要考虑窗体的功能，无需担心代码会被自己不感兴趣的信息搅乱。

对于部分类，最后要注意的一点是：应用于部分类的接口也会应用于整个类，也就是说，下面的两个定义：

```
public partial class MyClass : IMyIntefacel
{
   ...
}
public partial class MyClass : IMyInteface2
{
   ...
}
```

和

```
public class MyClass : IMyIntefacel, IMyInteface2
{
   ...
}
```

是等价的。

部分类定义可以在一个部分类定义文件或者多个部分类定义文件中包含基类。但如果基类在多个定义文件中指定，它就必须是同一个基类，因为在 C#中，类只能继承一个基类。

10.5　部分方法定义

部分类也可以定义部分方法。部分方法在部分类中定义，但没有方法体，在另一个部分类中包含实现代码。在这两个部分类中，都要使用 partial 关键字。

```
public partial class MyClass
{
    partial void MyPartialMethod();
}

public partial class MyClass
{
    partial void MyPartialMethod()
    {
        // Method implementation
    }
}
```

部分方法也可以是静态的，但它们总是私有的，且不能有返回值。它们使用的任何参数都不能是out 参数，但可以是 ref 参数。部分方法也不能使用 virtual、abstract、override、new、sealed 和 extern 修饰符。

有了这些限制，就不太容易看出部分方法的作用了。实际上，部分方法在编译代码时非常重要，其用法倒并不重要。考虑下面的代码：

```
public partial class MyClass
{
    partial void DoSomethingElse();

    public void DoSomething()
    {
        Console.WriteLine("DoSomething() execution started.");
        DoSomethingElse();
        Console.WriteLine("DoSomething() execution finished.");
    }
}

public partial class MyClass
{
    partial void DoSomethingElse()
    {
        Console.WriteLine("DoSomethingElse() called.");
    }
}
```

在第一个部分类定义中定义和调用部分方法DoSomethingElse，在第二个部分类中实现它。在控制台应用程序中调用DoSomething 时，输出如下内容：

```
DoSomething() execution started.
DoSomethingElse() called.
DoSomething() execution finished.
```

如果删除第二个部分类定义，或者删除部分方法的全部执行代码(注释掉代码)，输出就如下所示：

```
DoSomething() execution started.
DoSomething() execution finished.
```

读者可能认为，调用 DoSomethingElse 时，运行库发现该方法没有实现代码，因此会继续执行下一行代码。但实际上，编译代码时，如果代码包含一个没有实现代码的部分方法，编译器会完全删除该方法，还会删除对该方法的所有调用。执行代码时，不会检查实现代码，因为没有检查方法的调用。这会略微提高性能。

与部分类一样，在定制自动生成的代码或设计器创建的代码时，部分方法是很有用的。设计器会声明部分方法，用户根据具体情形选择是否实现它。如果不实现它，就不会影响性能，因为该方法在编译过的代码中不存在。

现在考虑为什么部分方法不能有返回类型。如果可以回答这个问题，就可以确保完全理解了这个主题，我们将此留作练习。

10.6　示例应用程序

为了解释前面使用的一些技术，下面开发一个类模块，以便在后续章节中使用。这个类模块包含两个类：

- Card—— 表示一张标准的扑克牌，包含梅花、方块、红心和黑桃，其顺序是从 A 到 K。
- Deck—— 表示一副完整的 52 张扑克牌，在扑克牌中可以按照位置访问各张牌，并可以洗牌。

再开发一个简单的客户程序，确保程序正常工作，但在整个扑克牌应用程序中不使用扑克牌。

10.6.1　规划应用程序

这个应用程序的类库 Ch10CardLib 包含类。在开始编写代码前，应规划一下需要的结构和类的功能。

1. Card 类

Card 类基本上是两个只读字段 suit 和rank 的容器。把字段指定为只读的原因是“空白”的牌是没有意义的，牌在创建好后也不能修改。为此，要把默认的构造函数指定为私有，并提供另一个构造函数，从给定的 suit 和rank 中建立一副扑克牌。

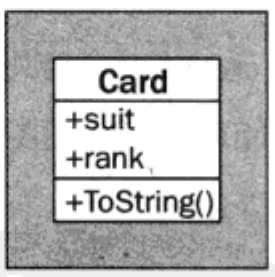


图　10-8

此外，Card 类要重写 System.Object 的 ToString()方法，这样才能获得人们可以理解的字符串，以表示扑克牌。为使编码简单一些，为两个字段 suit 和rank 提供枚举。

Card 类如图 10-8 所示。

2. Deck 类

Deck 类包含 52 个 Card 对象。我们为这些对象使用一个简单的数组类型。这个数组不能直接访问，因为对 Card 对象的访问要通过 GetCard()方法来实现，该方法返回指定索引的 Card 对象。这个类也有一个 Shuffle()方法，重新安排数组中的牌，所以它应如图 10-9 所示。

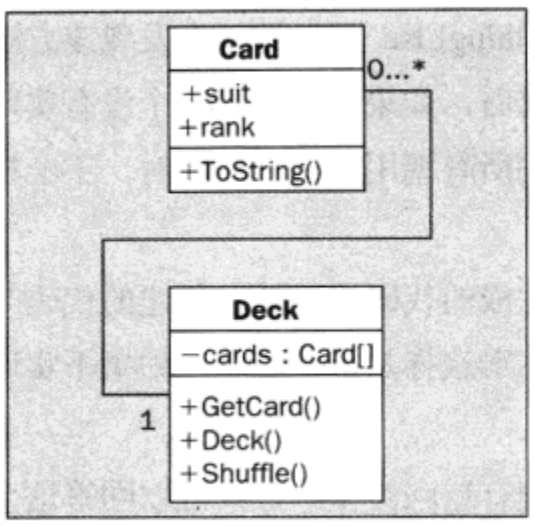


图　10-9

10.6.2　编写类库

对于本例，假定读者对 IDE 比较熟悉，所以不再使用标准的“试一试”方式，不再显式地列出步骤(这些步骤已经在前面多次用过)，重要的是详细讨论代码。不过，这里要包含一些指针以确保不出问题。

类和枚举都包含在一个类库项目 Ch10CardLib 中。这个项目将包含 4 个.cs 文件，Card.cs 包含 Card 类的定义，Deck.cs 包含 Deck 类的定义，Suit.cs 和 Rank.cs 文件包含枚举。

可以使用 VS 的类图工具把许多代码组合在一起。

如果使用的是 VCE，没有类图工具，不必担心。下面各节都包含了类图生成的代码，所以读者可以完成本例。这个项目的代码在不同的 IDE 中并没有区别。

首先需要完成如下操作：

(1) 在 C:\BegVCSharp\Chapter10 目录中创建一个新类库项目 Ch10CardLib。

(2) 从项目中删除 Class1.cs。

(3) 在 VS 中，使用 Solution Explorer 窗口中打开项目的类图(只有选择项目，而不是选择解决方案，才能显示出类图图标)。类图开始时应为空白，因为项目不包含类。

如果在这个类图中看到 Resources 和 Settings 类，可以右击它们，选择 Remove from Diagram 选项，隐藏它们。

1. 添加 Suit 和 Rank 枚举

把一个 Enum 从工具箱拖动到图中，再在显示的对话框中填充，就可以添加一个枚举。例如，对于 Suit 枚举，应在对话框中添加如图 10-10 所示的信息。

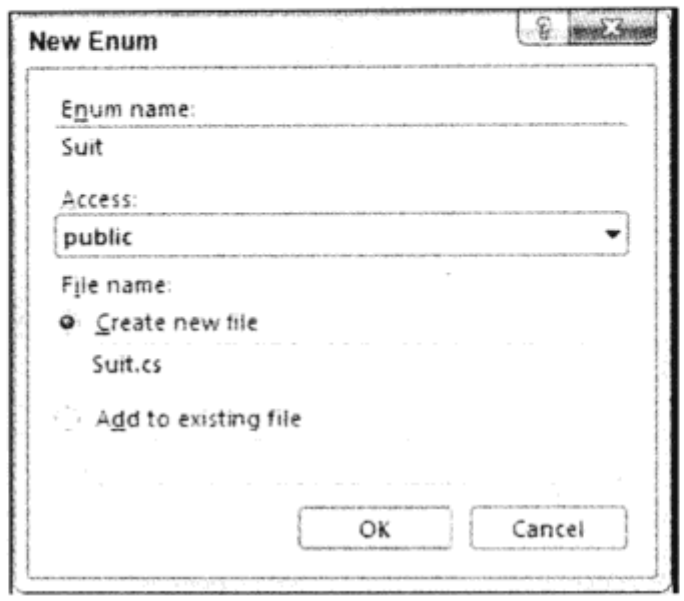


图　10-10

接着，使用 Class Details 窗口添加枚举的成员。需要添加的值如图 10-11 所示。

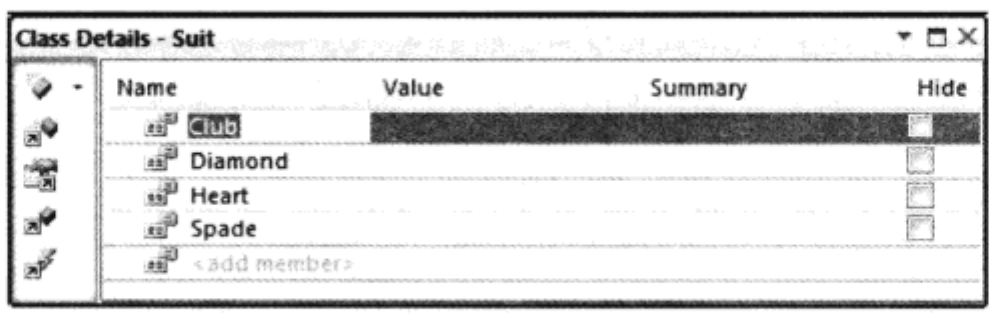


图　10-11

以相同的方式利用工具箱添加 Rank 枚举。需要的值如图 10-12 所示。

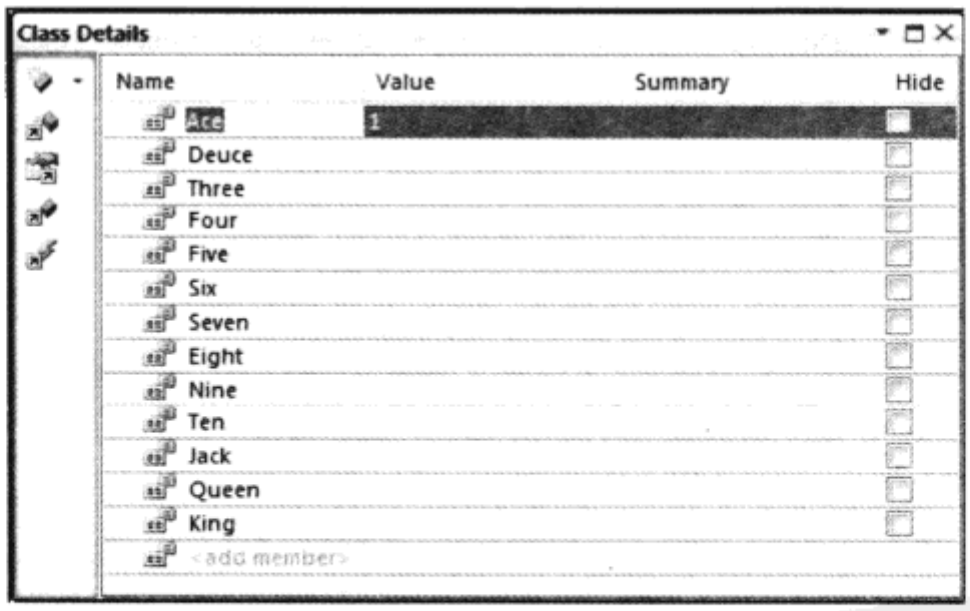


图　10-12

第一个成员 Ace 的输入值为 1，它会使枚举的底层存储匹配扑克牌的 Rank，这样 Six 就存储为 6。

完成上述操作后，类图就如图 10-13 所示。

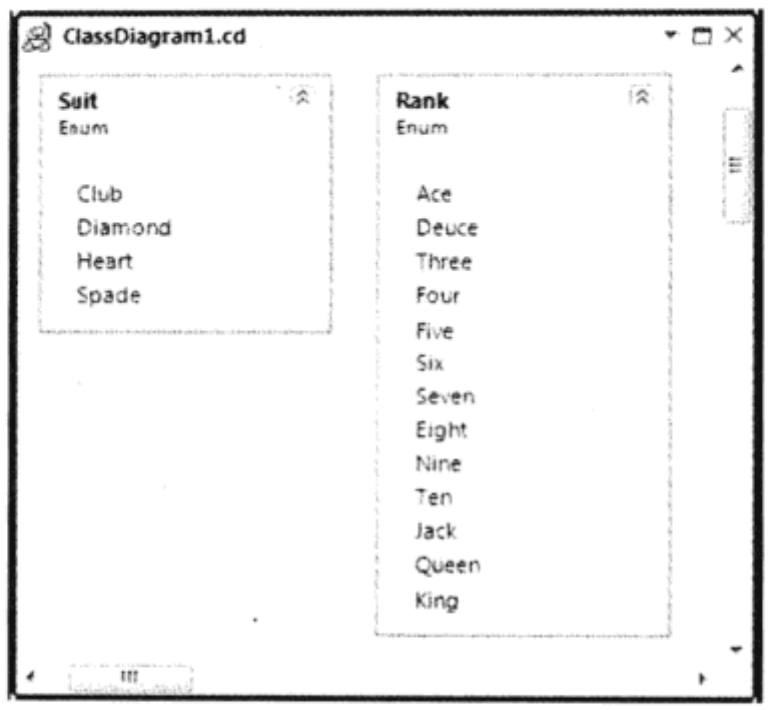


图 10-13

为这两个枚举生成的代码位于 Suit.cs 和 Rank.cs 文件中，如下所示：


```
using System;
using System.Collections.Generic;
using System.Ling;
using System.Text;

namespace Ch10CardLib
{
   public enum Suit
   {
      Club,
      Diamond,
      Heart,
      Spade,
   }
}
```

代码段 Ch10CardLib\Suit.cs


```
using System;
using System.Collections.Generic;
using System.Ling;
using System.Text;

namespace Ch10CardLib
{
   public enum Rank
   {
      Ace = 1,
      Deuce,
      Three,
      Four,
      Five,
      Six,
```

```
        Seven,
        Eight,
        Nine,
        Ten,
        Jack,
        Queen,
        King,
    }
}
```

代码段 Ch10CardLib\Rank.cs

如果使用的是 VCE，就可以添加 Suit.cs 和 Rank.cs 代码文件，再手工输入这些代码。注意，代码生成器在最后一个枚举成员后添加的逗号不会妨碍编译，不会创建一个额外的空成员，但它们可能会带来一些混乱。

2. 添加 Card 类

本节将结合使用类设计器和 VS 的代码编辑器添加 Card 类，也可以仅使用 VCE 中的代码编辑器。使用类设计器添加类与添加枚举十分类似，也是把相应的项从工具箱拖动到类图中。这里要把 Class 拖动到类图中，并把新类命名为 Card。

为了添加字段 rank 和 suit，可以使用 Class Details 窗口添加字段，再使用 Properties 窗口把字段的 Constant Kind 设置为 readonly。还需要添加两个构造函数，一个是默认构造函数(私有)，另一个构造函数带有两个参数：newSuit 和 newRank，其类型分别是 Suit 和 Rank(公共)。最后重写 ToString()，这需要在 Properties 窗口中修改 Inheritance Modifier，将它设置为 override。

图 10-14 显示了 Class Details 窗口和已输入所有信息的 Card 类。

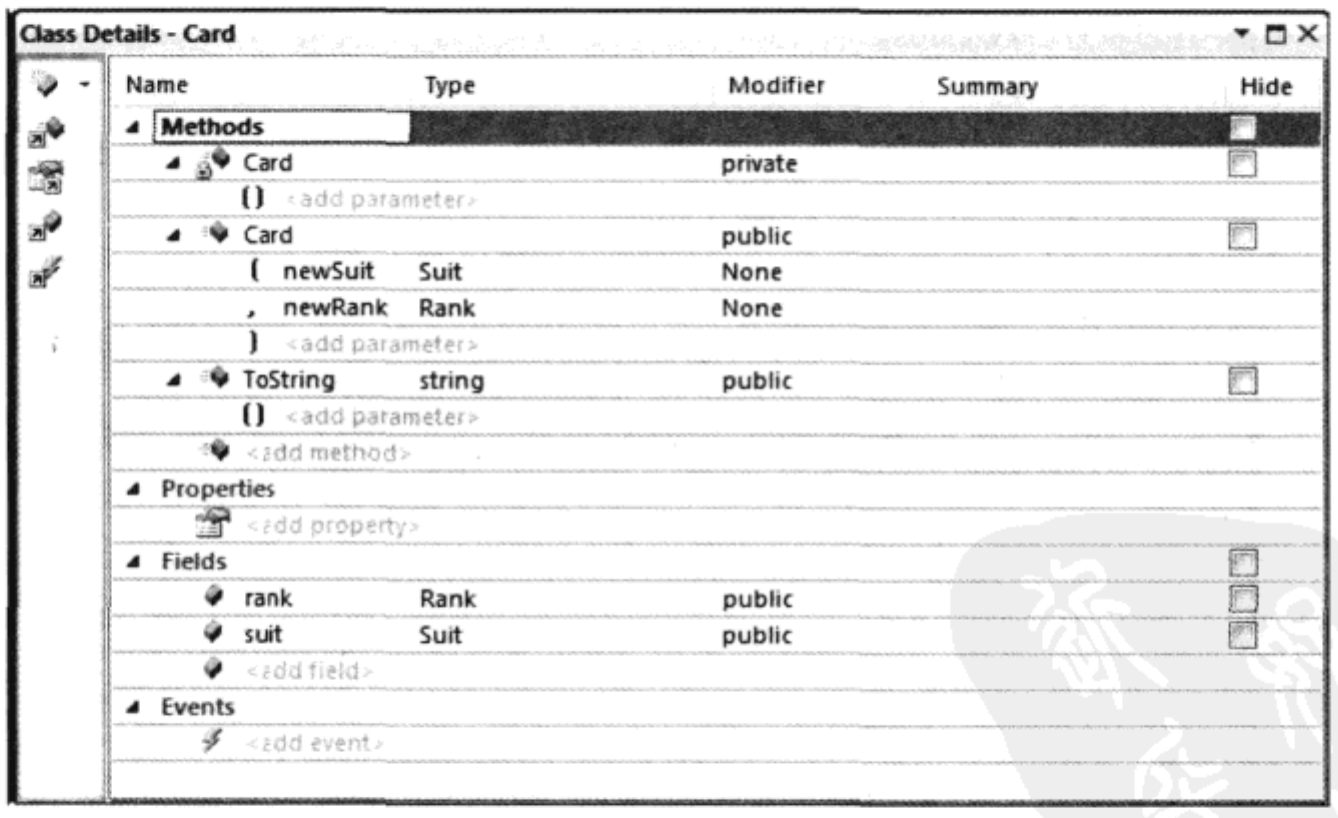


图　10-14

然后需要修改 Card.cs 中类的代码(如果使用的是 VCE，就应把这些代码添加到名称空间 Ch10CardLib 的新类 Card 中)，如下所示：

可从 wrox.com 下载源代码

```
public class Card
{
   public readonly Suit suit;
   public readonly Rank rank;

   public Card(Suit newSuit, Rank newRank)
   {
      suit = newSuit;
      rank = newRank;
   }

   private Card()
   {

   }

   public override string ToString()
   {
      return "The " + rank + " of " + suit + "s";
   }
}
```

代码段 Ch10CardLib\Card.cs

重写的 ToString()方法将已存储的枚举值的字符串表示写入到返回的字符串中，非默认的构造函数初始化 suit 和 rank 字段的值。

3. 添加 Deck 类

Deck 类需要使用类图定义的如下成员：

- Card[]类型的私有字段 cards。
- 公共的默认构造函数。
- 公共方法 GetCard()，它带有一个 int 参数 cardNum，并返回一个 Card 类型的对象。
- 公共方法 Shuffle()，它不带参数，返回 void。

添加了这些成员后，Deck 类的 Class Details 窗口就如图 10-15 所示。

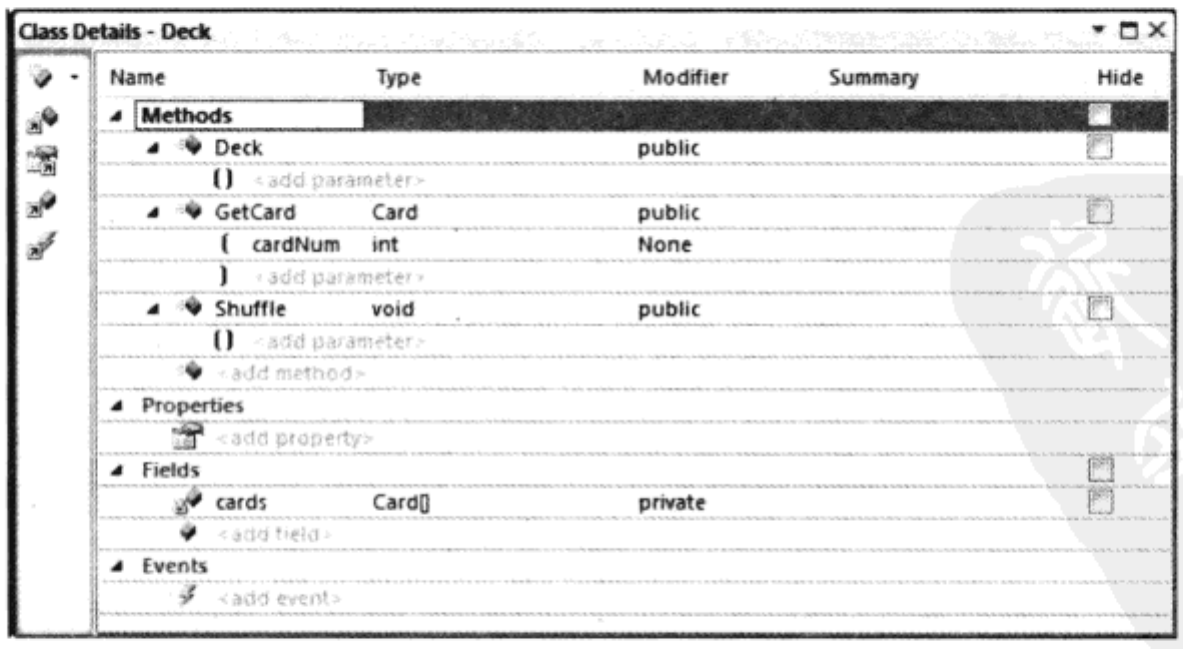


图 10-15

为了使类图更清晰，可以显示所添加的成员和类型之间的关系。在类图中依次右击下面的项，从菜单中选择 Show as Association 选项：

- Deck 中的 cards
- Card 中的 suit
- Card 中的 rank

完成后，类图如图 10-16 所示。

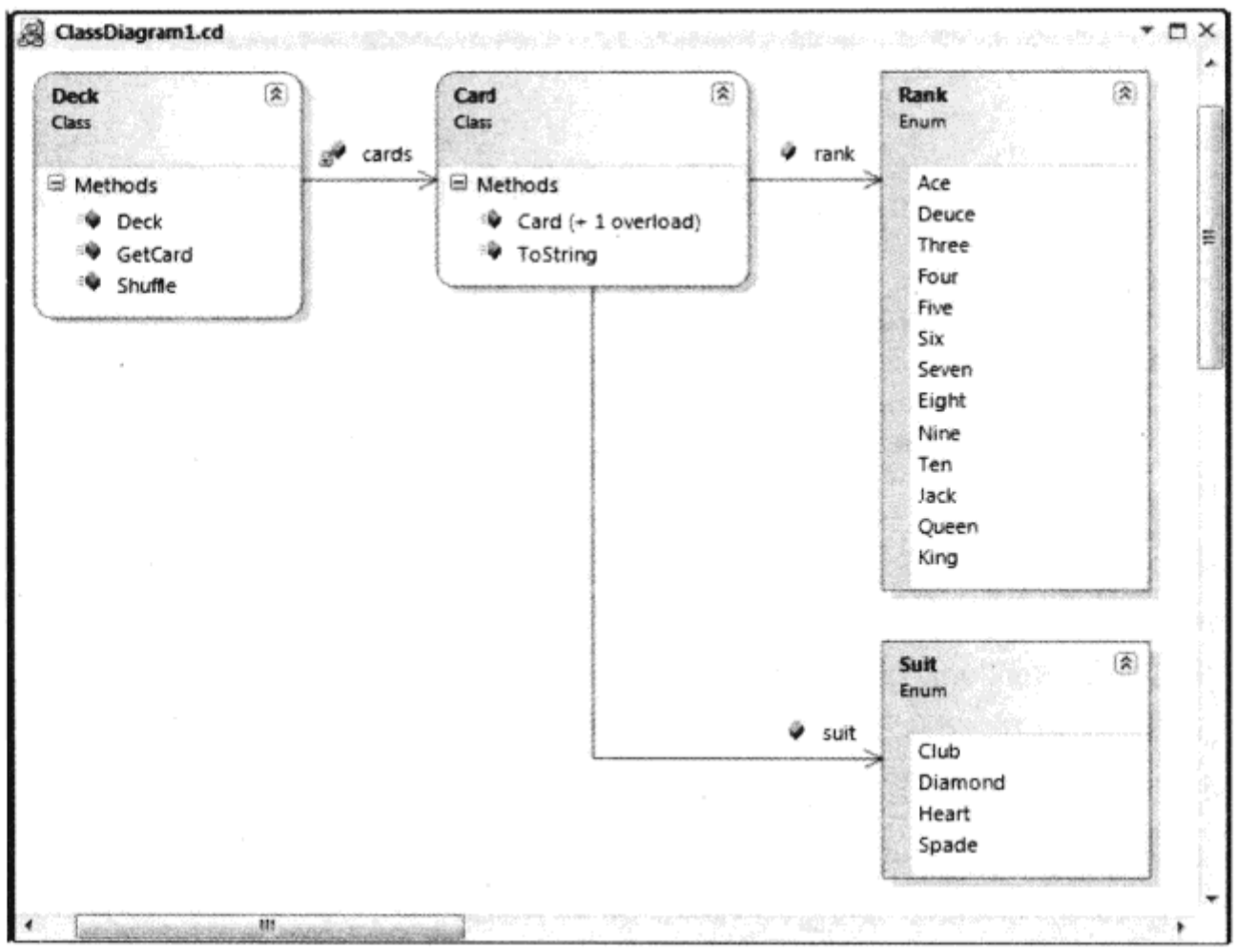


图　10-16

接着，修改 Deck.cs 中的代码(如果使用的是 VCE，就必须先使用下面的代码添加这个类)。首先实现构造函数，它在 cards 字段中创建 52 张牌，并给它们赋值。对两个枚举的所有组合进行迭代，每次迭代都创建一张牌。这将使 cards 最初包含一个有序的扑克牌列表：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Ling;
using System.Text;

namespace Ch10CardLib
{
   public Class Deck()
   {
      Private Card[] cards;
       public Deck()
      {
         cards = new Card[52];
         for (int suitVal = 0; suitVal < 4; suitVal++)
         {
```

```
            for (int rankVal = 1; rankVal < 14; rankVal++)
            {
               cards[suitVal * 13 + rankVal - 1] = new Card((Suit)suitVal,
                                                            (Rank)rankVal);
            }
         }
      }
```

代码段 Ch10CardLib\Deck.cs

然后实现 GetCard()方法，为指定的索引返回 Card 对象，或者以与前面相同的方式抛出一个异常：

```
public Card GetCard(int cardNum)
{
   if (cardNum >= 0 && cardNum <= 51)
      return cards[cardNum];
   else
      throw (new System.ArgumentOutOfRangeException("cardNum", cardNum,
               "Value must be between 0 and 51."));
}
```

最后实现 Shuffle()方法。这个方法创建一个临时的扑克牌数组，并把扑克牌从现有的 cards 数组随机复制到这个数组中。这个函数的主体是一个从 0~51 的循环，在每次循环时，都会使用.NET Framework 中 System.Random 类的实例生成一个 0~51 之间的随机数。进行了实例化后，这个类的对象使用方法 Next(X)生成一个介于 0～X 之间的随机数。有了一个随机数后，就可以使用它作为临时数组中 Card 对象的索引，以便复制 cards 数组中的扑克牌。

为了记录已赋值的扑克牌，我们还有一个 bool 变量的数组，在复制每张牌时，把该数组中的值指定为 true。在生成随机数时，检查这个数组，看看是否已经把一张牌复制到临时数组中由随机数指定的位置上了，如果已经复制好了，就将生成另一个随机数。

这不是完成该任务的最高效的方式，因为生成的许多随机数都可能找不到空位置以复制扑克牌。但是，它仍能完成任务，而且很简单，因为 C#代码的执行速度很快，我们几乎觉察不到延迟。代码如下：

```
public void Shuffle()
{
   Card[] newDeck = new Card[52];
   bool[] assigned = new bool[52];
   Random sourceGen = new Random();
   for (int i = 0; i < 52; i++)
   {
      int destCard = 0;
      bool foundCard = false;
      while (foundCard == false)
      {
         destCard = sourceGen.Next(52);
         if (assigned[destCard] == false)
            foundCard = true;
      }
      assigned[destCard] = true;
      newDeck[destCard] = cards[i];
```

```
            }
            newDeck.CopyTo(cards, 0);
        }
    }
}
```

这个方法的最后一行使用 System.Array 类的 CopyTo()方法(在创建数组时使用)，把 newDeck 中的每张扑克牌复制回 cards 中。也就是说，我们使用同一个 cards 对象中的同一组 Card 对象，而不是创建新的实例。如果改用 cards=newDeck，就会用另一个对象替代 cards 引用的对象实例。如果其他地方的代码仍保留对原 cards 实例的引用，就会出问题——不会洗牌。

至此，就完成了类库代码。

10.6.3　类库的客户应用程序

为了简单起见，可以在包含类库的解决方案中添加一个客户控制台应用程序。为此，只需在 Solution Explorer 窗口中右击解决方案，选择 Add | New Project，新项目命名为 Ch10CardClient。

为了在这个新的控制台应用程序项目中使用前面创建的类库，只需添加一个对类库项目 Ch10CardLib 的引用。为此，可以使用 Add Reference 对话框的 Projects 选项卡，如图 10-17 所示。

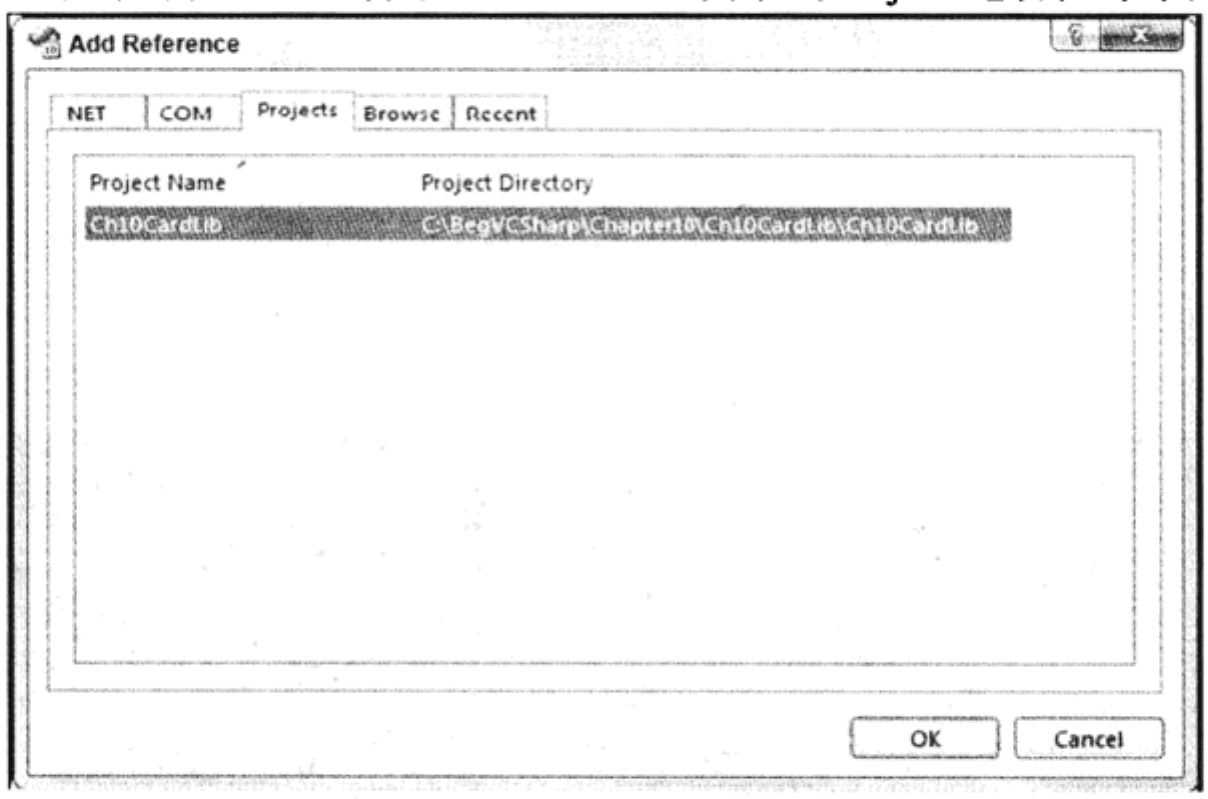


图　10-17

选择项目，单击 OK 按钮，就添加了引用。

因为这个新项目是第二个创建的，所以还需要指定该项目为解决方案的启动项目，即在单击 Run 后，将执行这个项目。为此，在 Solution Explorer 窗口中右击该项目名，选择 Set as StartUp Project 菜单项。

然后需要添加使用新类的代码，这些代码不需要做什么特别的任务，所以添加下面的代码就可以：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using Ch10CardLib;

namespace Ch10CardClient
{
   class Program
   {
      static void Main(string[] args)
      {
         Deck myDeck = new Deck();
         myDeck.Shuffle();
         for (int i = 0; i < 52; i++)
         {
            Card tempCard = myDeck.GetCard(i);
            Console.Write(tempCard.ToString());
            if (i != 51)
               Console.Write(", ");
            else
               Console.WriteLine();
         }
         Console.ReadKey();
      }
   }
}
```

代码段 Ch10CardClient\Program.cs

其结果如图 10-18 所示。

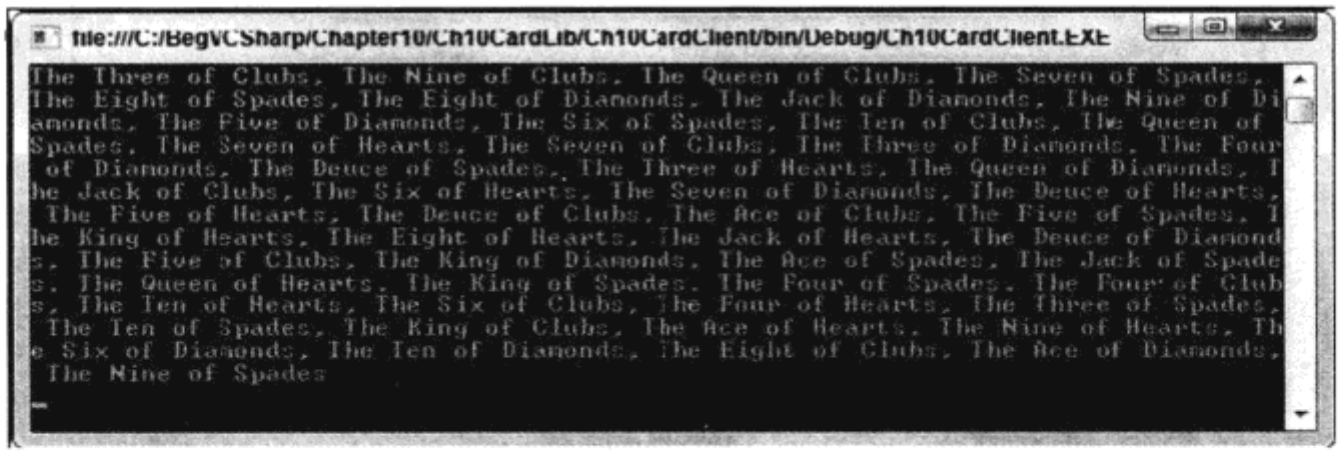


图 10-18

52 张扑克牌是随机放置的。后面的章节将继续开发和使用这个类库。

10.7 Call Hierarchy 窗口

现在分析 VS 2010 中的一项新功能：Call Hierarchy 窗口，它可以审查代码，确定方法在哪里调用，以及它们与其他方法的关系。说明这个功能的最好方式是举一个例子。

打开上一节的示例应用程序，再打开 Deck.cs 代码文件，找到 Shuffle()方法，右击它，选择 View Call Hierarchy 菜单项，将显示如图 10-19 所示的窗口(其中展开了一些区域)。

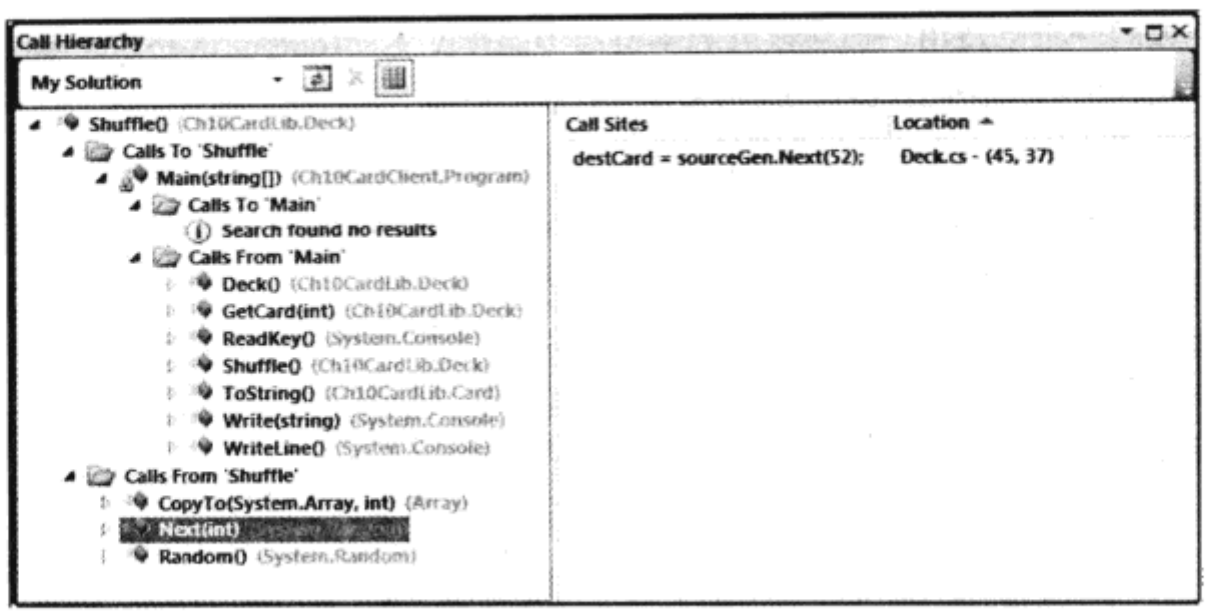


图　10-19

从 Shuffle()方法开始，可以在窗口的树形视图中找出调用该方法的所有代码，以及这个方法进行的所有调用。例如，在 Shuffle()中调用了图中突出显示的 Next(int)方法，所以它显示在 Calls From 'Shuffle'部分。单击一个调用时，会在右边看到进行这个调用的代码行及其位置。双击该位置，会立即跳到进行这个调用的代码行上。

还可以沿着层次结构向下研究其中的方法，在图 10-19 中就是 Main()方法，图中显示了从 Main()方法中调用的方法和调用 Main()的方法。

调试和重构代码时，这个窗口是非常有用的，因为它允许查看不同部分的代码是如何相关的。

10.8　小结

本章结束了定义基类的讨论。仍有许多内容没有包含进来，但前面涉及到的技术已经足够创建相当复杂的应用程序了。

本章介绍了如何定义字段、方法和属性，接着讨论了各种访问级别和修饰关键字。我们还介绍了把类组合在一起的快捷工具。

介绍过这些基本主题后，我们详细讨论了继承行为，主要内容是如何用 new 关键字隐藏不想要的继承成员，扩展基类成员，而不是替代它们的实现代码(使用 base 关键字)。我们还论述了嵌套的类定义。之后，详细研究了接口的定义和实现，包括显式和隐式实现的概念。学习了如何使用部分类和部分方法定义把定义放在多个代码文件中。

最后，我们开发和使用了一个表示扑克牌的简单类库，使用方便的类图工具使工作更便于完成。后面的章节还将进一步使用这个库。

第 11 章将介绍集合，这是类的一种类型，在开发过程中经常使用。

10.9　练习

(1) 编写代码，定义一个基类 MyClass，其中包含虚拟方法 GetString()。这个方法应返回存储在受保护字段 myString 中的字符串，该字段可以通过只写公共属性 ContainedString 来访问。

(2) 从类 MyClass 中派生一个类 MyDerivedClass。重写 GetString()方法，使用该方法的基类实现

代码从基类中返回一个字符串，但在返回的字符串中添加文本“(output from derived class)”。

(3) 部分方法定义必须使用 void 返回类型。说明其原因。

(4) 编写一个类 MyCopyableClass，该类可以使用方法 GetCopy()返回它本身的一个副本。这个方法应使用派生于 System.Object 的 MemberwiseClone()方法。给该类添加一个简单的属性，并且编写客户代码，客户代码使用该类检查任务是否成功执行。

(5) 为 Ch10CardLib 库编写一个控制台客户程序，从搅乱的 Deck 对象中一次取出 5 张牌。如果这 5 张牌都是相同的花色，客户程序就应在屏幕上显示这 5 张牌，以及文本“Flush!”，否则就显示 50 张牌以及文本“No flush”，并退出。

附录 A 给出了练习答案。

10.10　本章要点

主　题	重 要 概 念
成员定义	可以在类中定义字段、方法和属性成员。字段用可访问性、名称和类型定义，方法用可访问性、返回类型、名称和参数定义，属性用可访问性、名称、get 和/或 set 存取器定义。各个属性存取器可以有自己的可访问性，但它必须低于整个属性的可访问性
成员隐藏和重写	属性和方法可以在基类中定义为抽象或虚拟，以定义继承。派生类必须实现抽象的成员，使用 override 关键字可以重写虚拟的成员。派生类还可以用 new 关键字提供新的实现代码，用 sealed 关键字禁止进一步重写虚拟成员。基类的实现代码可以用 base 关键字调用
接口的实现	实现了接口的类必须实现该接口定义为公共的所有成员。可以隐式或显式实现接口，其中显式实现代码只能通过接口引用来使用
部分定义	使用 partial 关键字可以把类定义放在多个代码文件中。还可以使用 partial 关键字创建部分方法。部分方法有一些限制，包括没有返回值或 out 参数，如果没有提供实现代码，部分方法就不能编译

第11章

集合、比较和转换

本章内容：

- 如何定义和使用集合
- 可以使用的不同类型的集合
- 如何比较类型，如何使用 is 运算符
- 如何比较值，如何重载运算符
- 如何定义和使用转换
- 如何使用 as 运算符

前面讨论了 C#中所有的基本 OOP 技术，读者还应熟悉一些比较高级的技术。本章的主要内容如下：

- **集合**：可以使用集合来维护对象组。与前面章节使用的数组不同，集合可以包含更高级的功能，例如，控制对它们包含的对象的访问、搜索和排序等。本章将介绍如何使用和创建集合类，学习掌握它们的一些强大技术。
- **比较**：在处理对象时，常常要比较它们。这对于集合尤其重要，因为这是排序的实现方式。本章将介绍如何以各种方式比较对象，包括运算符重载，使用IComparable和IComparer 接口对集合排序。
- **转换**：在前面的章节中，介绍了如何把对象从一种类型转换为另一种类型。本章讨论如何定制类型转换，以满足自己的要求。

11.1 集合

第 5 章介绍了如何使用数组创建包含许多对象或值的变量类型。但数组有一定的限制。最大的限制是一旦创建好数组，它们的大小就是固定的，不能在现有数组的末尾添加新项，除非创建一个新的数组。这常常意味着用于处理数组的语法比较复杂。OOP 技术可以创建在内部执行大多数此类处理的类，因此简化了使用项列表或数组的代码。

C#中的数组实现为 System.Array 类的实例，它们只是集合类(Collection Classes)中的一种类型。集合类一般用于处理对象列表，其功能比简单数组要多，功能大多是通过实现 System.Collections 名称空间中的接口而获得的，因此集合的语法已经标准化了。这个名称空间还包含其他一些有趣的东西，例如，以与 System.Array 不同的方式实现这些接口的类。

集合的功能(包括基本功能，例如，用[index]语法访问集合中的项)可以通过接口来实现，该接口不仅没有限制我们使用基本集合类，例如System.Array，相反，我们还可以创建自己的定制集合类。这些集合可以专用于要枚举的对象(即要从中建立集合的对象)。这么做的一个优点是定制的集合类可以是强类型化的。也就是说，从集合中提取项时，不需要把它们转换为正确的类型。另一个优点是提供专用的方法，例如，可以提供获得项子集的快捷方法，在扑克牌示例中，可以添加一个方法，来获得特定花色中的所有 Card 项。

System.Collections 名称空间中的几个接口提供了基本的集合功能：

- IEnumerable 可以迭代集合中的项。
- ICollection(继承于 IEnumerable)可以获取集合中项的个数，并能把项复制到一个简单的数组类型中。
- IList (继承于 IEnumerable 和 ICollection)提供了集合的项列表，允许访问这些项，并提供其他一些与项列表相关的基本功能。
- IDictionary(继承于 IEnumerable 和 ICollection)类似于 IList，但提供了可通过键值(而不是索引)访问的项列表。

System.Array 类实现了IList、ICollection 和 IEnumerable，但不支持 IList 的一些更高级的功能，它表示大小固定的项列表。

11.1.1　使用集合

Systems.Collections 名称空间中的类 System.Collections.ArrayList 也实现了 IList、ICollection 和 IEnumerable 接口，但实现方式比 System.Array 更复杂。数组的大小是固定的(不能增加或删除元素)，而这个类可以用于表示大小可变的项列表。为了更准确地理解这个高级集合的功能，下面介绍一个使用这个类和一个简单数组的示例。

试一试：数组和高级集合

(1) 在 C:\BegVCSharp\Chapter11 目录中创建一个新的控制台应用程序 Ch11Ex01。

(2) 在 Solution Explorer 窗口中右击项目，选择 Add | Class 选项，给项目添加 3 个新类：Animal、Cow 和 Chicken。

(3) 修改 Animal.cs 中的代码，如下所示：


可从 wrox.com 下载源代码

```
namespace Ch11Ex01
{
   public abstract class Animal
   {
      protected string name;

      public string Name
      {
         get
```

```
        {
          return name;
        }
        set
        {
          name = value;
        }
      }

      public Animal()
      {
        name = "The animal with no name";
      }

      public Animal(string newName)
      {
        name = newName;
      }

      public void Feed()
      {
        Console.WriteLine("{0} has been fed.", name);
      }
    }
}
```

代码段 Ch11Ex01\Animal.cs

(4) 修改 Cow.cs 中的代码，如下所示：


```
namespace Ch11Ex01
{
  public class Cow : Animal
  {
    public void Milk()
    {
      Console.WriteLine("{0} has been milked.", name);
    }

    public Cow(string newName) : base(newName)
    {
    }
  }
}
```

代码段 Ch11Ex01\Cow.cs

(5) 修改 Chicken.cs 中的代码，如下所示：


```
namespace Ch11Ex01
{
  public class Chicken : Animal
  {
    public void LayEgg()
    {
```

```
            Console.WriteLine("{0} has laid an egg.", name);
        }

        public Chicken(string newName) : base(newName)
        {
        }
    }
}
```

代码段 Ch11Ex01\Chicken.cs

(6) 修改 Program.cs 中的代码，如下所示：


```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex01
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("Create an Array type collection of Animal " +
                           "objects and use it:");

         Animal[] animalArray = new Animal[2];
   Cow myCow1 = new Cow("Deirdre");
animalArray[0] = myCow1;
animalArray[1] = new Chicken("Ken");

foreach (Animal myAnimal in animalArray)
{
   Console.WriteLine("New {0} object added to Array collection, " +
                     "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}

Console.WriteLine("Array collection contains {0} objects.",
                  animalArray.Length);
animalArray[0].Feed();
((Chicken)animalArray[1]).LayEgg();
Console.WriteLine();

Console.WriteLine("Create an ArrayList type collection of Animal " +
                  "objects and use it:");
ArrayList animalArrayList = new ArrayList();
Cow myCow2 = new Cow("Hayley");
animalArrayList.Add(myCow2);
animalArrayList.Add(new Chicken("Roy"));

foreach (Animal myAnimal in animalArrayList)
{
   Console.WriteLine("New {0} object added to ArrayList collection," +
```

```
                                " Name = {1}", myAnimal.ToString(), myAnimal.Name);
            }
            Console.WriteLine("ArrayList collection contains {0} objects.",
                animalArrayList.Count);
            ((Animal)animalArrayList[0]).Feed();
            ((Chicken)animalArrayList[1]).LayEgg();
            Console.WriteLine();

            Console.WriteLine("Additional manipulation of ArrayList:");
            animalArrayList.RemoveAt(0);
            ((Animal)animalArrayList[0]).Feed();
            animalArrayList.AddRange(animalArray);
            ((Chicken)animalArrayList[2]).LayEgg();
            Console.WriteLine("The animal called {0} is at index {1}.",
                              myCow1.Name, animalArrayList.IndexOf(myCow1));
            myCow1.Name = "Janice";
            Console.WriteLine("The animal is now called {0}.",
                              ((Animal)animalArrayList[1]).Name);
            Console.ReadKey();
        }
    }
}
```

代码段 Ch11Ex01\Program.cs

(7) 运行该应用程序，其结果如图 11-1 所示。

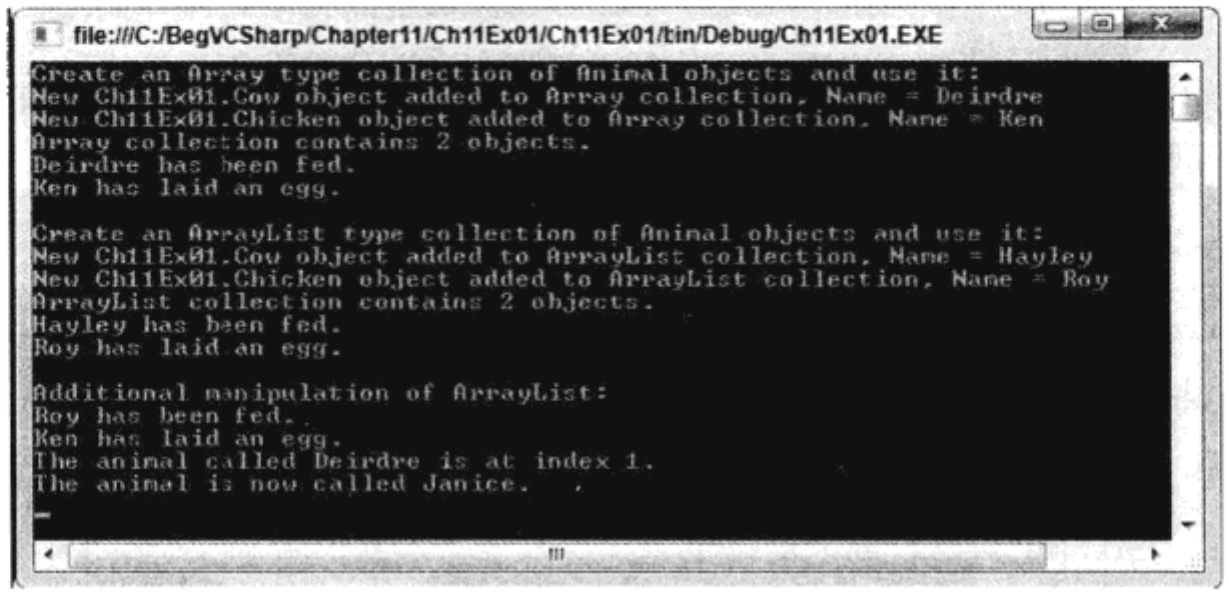


图　11-1

示例的说明

这个示例创建了两个对象集合，第一个集合使用 System.Array 类(这是一个简单的数组)，第二个集合使用 System.Collections.ArrayList 类。这两个集合都是 Animal 对象，在 Animal.cs 中定义。Animal 类是抽象类，所以不能进行实例化。但通过多态性(详见第 8 章)，可以使集合中的项成为派生于 Animal 类的 Cow 和 Chicken 类实例。

这些数组在 Class1.cs 的 Main()方法中创建好后，就可以显示其特性和功能。有几个处理操作可以应用到 Array 和 ArrayList 集合上，但它们的语法略有区别。也有一些操作只能使用更高级的 ArrayList 类型。

下面首先通过比较这两种集合类型的代码和结果，讨论一下类似的操作。首先是集合的创建。

对于简单的数组来说，只有用固定的大小来初始化数组，才能使用它。下面使用第 5 章介绍的标准语法创建数组 animalArray：

```
Animal[] animalArray = new Animal[2];
```

而 ArrayList 集合不需要初始化其大小，所以可以使用以下代码创建列表 animalArrayList：

```
ArrayList animalArrayList = new ArrayList();
```

这个类还有另外两个构造函数。第一个构造函数把现有的集合作为一个参数，把现有集合的内容复制到新实例中；而另一个构造函数通过一个参数设置集合的容量(capacity)。这个容量用一个 int 值指定，设置集合中可以包含的初始项数。但这并不是真实的容量，因为如果集合中的项数超过了这个值，容量就会自动增加一倍。

因为数组是引用类型(例如，Animal 和Animal 派生的对象)，所以用一个长度初始化数组并没有初始化它所包含的项。要使用一个指定的项，该项还需要初始化，即需要给这个项赋予初始化了的对象：

```
Cow myCow1 = new Cow("Deirdre");
animalArray[0] = myCow1;
animalArray[1] = new Chicken("Ken");
```

这段代码以两种方式完成该初始化任务：用现有的 Cow 对象来赋值，或者通过创建一个新的 Chicken 对象来赋值。主要的区别是前者引用了数组中的对象——我们在代码的后面就使用了这种方式。

对于ArrayList 集合，它没有现成的项，也没有 null 引用的项。这样就不能以相同的方式给索引赋予新实例。我们使用 ArrayList 对象的 Add()方法添加新项：

```
Cow myCow2 = new Cow("Hayley");
animalArrayList.Add(myCow2);
animalArrayList.Add(new Chicken("Roy"));
```

除了语法稍有不同外，还可以采用相同的方式把新对象或现有的对象添加到集合中。以这种方式添加完项后，就可以使用与数组相同的语法来改写它们，例如：

```
animalArrayList[0] = new Cow("Alma");
```

但不能在这个示例中这么做。

第 5 章介绍了如何使用 foreach 结构迭代一个数组。这是可以的，因为 System.Array 类实现了 IEnumerable 接口，这个接口的唯一方法 GetEnumerator()可以迭代集合中的各项。后面将更加深入地讨论这一点。在代码中，我们写出了数组中每个 Animal 对象的信息：

```
foreach (Animal myAnimal in animalArray)
{
   Console.WriteLine("New {0} object added to Array collection, " +
                     "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

这里使用的 ArrayList 对象也支持 IEnumerable 接口，并可以与 foreach 一起使用，此时语法是相同的：

```
foreach (Animal myAnimal in animalArrayList)
{
   Console.WriteLine("New {0} object added to ArrayList collection, " +
                     "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

接着，使用数组的 Length 属性，在屏幕上输出数组中元素的个数：

```
Console.WriteLine("Array collection contains {0} objects.",
                  animalArray.Length);
```

也可以使用ArrayList 集合得到相同的结果，但要使用 Count 属性，该属性是 ICollection 接口的一部分：

```
Console.WriteLine("ArrayList collection contains {0} objects.",
                  animalArrayList.Count);
```

集合——无论是简单的数组，还是比较复杂的集合——都用得不多，除非它们可以访问属于它们的项。简单数组是强类型化的，可以直接访问它们所包含的项类型。所以您可以直接调用项的方法：

```
animalArray[0].Feed();
```

数组的类型是抽象类型 Animal，因此不能直接调用由派生类提供的方法，而必须使用数据类型转换：

```
((Chicken)animalArray[1]).LayEgg();
```

ArrayList 集合是 System.Object 对象的集合(通过多态性赋给 Animal 对象)，所以必须对所有的项进行数据类型转换：

```
((Animal)animalArrayList[0]).Feed();
((Chicken)animalArrayList[1]).LayEgg();
```

代码的剩余部分利用的一些 ArrayList 集合功能超出了 Array 集合的功能范围。首先，可以使用 Remove()和 RemoveAt()方法删除项，这两个方法是在 ArrayList 类中实现的 IList 接口的一部分。它们分别根据项的引用或索引从数组中删除项。本例使用后一个方法删除列表中的第一项，即 Name 属性为 Hayley 的 Cow 对象：

```
animalArrayList.RemoveAt(0);
```

另外，还可以使用：

```
animalArrayList.Remove(myCow2);
```

因为这个对象已经有一个本地引用了，所以可以通过 Add()添加对数组的一个现有引用，而不是创建一个新对象。无论采用哪种方式，集合中唯一剩下的项是Chicken 对象，可以通过以下方式访问它：

```
((Animal)animalArrayList[0]).Feed();
```

对 ArrayList 对象中的项进行修改，使数组中剩下 N 个项，其实现方式与保留从 0~N-1 的索引

相同。例如，删除索引为 0 的项，会使其他项在数组中移动一个位置，所以应使用索引 0(而非 1)来访问 Chicken 对象。不再有索引为 1 的项了(因为集合中最初只有两个项)，所以如果试图执行下面的代码，就会抛出异常：

```
((Animal)animalArrayList[1]).Feed();
```

ArrayList 集合可以用AddRange()方法一次添加好几个项。这个方法接受带有 ICollection 接口的任何对象，包括前面的代码所创建的 animalArray 数组：

```
animalArrayList.AddRange(animalArray);
```

为了确定这是否有效，可以试着访问集合中的第三项，它将是 animalArray 中的第二项：

```
((Chicken)animalArrayList[2]).LayEgg();
```

AddRange()方法不是 ArrayList 提供的任何接口的一部分。这个方法专用于 ArrayList 类，论证了可以在集合类中执行定制操作，而不仅仅是前面介绍的接口要求的操作。这个类还提供了其他有趣的方法，如InsertRange()，它可以把数组对象插入到列表中的任何位置，还有用于排序和重新排序数组的方法。

最后，再回头来看看对同一个对象进行多个引用。使用 IList 接口中的 IndexOf()方法可以看出，myCow1(最初添加到 animalArray 中的一个对象)现在是 animalArrayList 集合的一部分，它的索引如下：

```
Console.WriteLine("The animal called {0} is at index {1}.",
                  myCow1.Name, animalArrayList.IndexOf(myCow1));
```

例如，接下来的两行代码通过对象引用重新命名了对象，并通过集合引用显示了新名称：

```
myCow1.Name = "Janice";
Console.WriteLine("The animal is now called {0}.",
                  ((Animal)animalArrayList[1]).Name);
```

11.1.2　定义集合

前面介绍了使用高级集合类能完成什么任务，下面讨论如何创建自己的、强类型化的集合。一种方式是手动执行需要的方法，但这比较费时间，在某些情况下也非常复杂。我们还可以从一个类中派生自己的集合，例如System.Collections.CollectionBase 类，这个抽象类提供了集合类的许多实现方式。这是推荐使用的方式。

CollectionBase 类有接口 IEnumerable、ICollection 和 IList，但只提供了一些需要的实现代码，特别是 IList 的 Clear()和 RemoveAt()方法，以及 ICollection 的 Count 属性。如果要使用提供的功能，就需要自己执行其他代码。

为便于完成任务，CollectionBase 提供了两个受保护的属性，它们可以访问存储的对象本身。我们可以使用 List 和 InnerList，List 可以通过 IList 接口访问项，InnerList 则是用于存储项的 ArrayList 对象。

例如，存储 Animal 对象的集合类可以定义如下(稍后介绍一个比较完整的实现代码)：

```
public class Animals : CollectionBase
{
```

```
    public void Add(Animal newAnimal)
    {
      List.Add(newAnimal);
    }

    public void Remove(Animal oldAnimal)
    {
      List.Remove(oldAnimal);
    }

    public Animals()
    {
    }
}
```

其中，Add()和 Remove()方法实现为强类型化的方法，使用 IList 接口中用于访问项的标准 Add()方法。该方法现在只用于处理 Animal 类或派生于 Animal 的类，而前面介绍的 ArrayList 实现代码可处理任何对象。

CollectionBase 类可以对派生的集合使用 foreach 语法。例如，可以使用下面的代码：

```
Console.WriteLine("Using custom collection class Animals:");
Animals animalCollection = new Animals();
animalCollection.Add(new Cow("Sarah"));
foreach (Animal myAnimal in animalCollection)
{
  Console.WriteLine("New {0} object added to custom collection, " +
                    "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

但不能使用下面的代码：

```
animalCollection[0].Feed();
```

要以这种方式通过索引来访问项，就需要使用索引符。

11.1.3　索引符

索引符(indexer)是一种特殊类型的属性，可以把它添加到一个类中，以提供类似于数组的访问。实际上，可以通过索引符提供更复杂的访问，因为我们可以用方括号语法定义和使用复杂的参数类型。它最常见的一个用法是对项执行简单的数字索引。

在 Animal 对象的 Animals 集合中添加一个索引符，如下所示：

```
public class Animals : CollectionBase
{
  ...

  public Animal this[int animalIndex]
  {
    get
    {
      return (Animal)List[animalIndex];
    }
    set
```

```
        {
            List[animalIndex] = value;
        }
    }
}
```

this 关键字与方括号中的参数一起使用，但这看起来类似于其他属性。这个语法是合理的，因为在访问索引符时，将使用对象名，后跟放在方括号中的索引参数(例如 MyAnimals[0])。

这段代码对 List 属性使用一个索引符(即在 IList 接口上，可以访问 CollectionBase 中的 ArrayList，ArrayList 存储了项)：

```
return (Animal)List[animalIndex];
```

这里需要进行显式数据类型转换，因为 IList.List 属性返回一个 System.Object 对象。注意，我们为这个索引符定义了一个类型。使用该索引符访问某项时，就可以得到这个类型。这种强类型化功能意味着，可以编写下述代码：

```
animalCollection[0].Feed();
```

而不是：

```
((Animal)animalCollection[0]).Feed();
```

这是强类型化的定制集合的另一个方便特性。下面扩展上一个示例，实践一下该特性。

试一试：实现 Animals 集合

(1) 在 C:\BegVCSharp\Chapter11 目录中创建一个新控制台应用程序 Ch11Ex02。

(2) 在 Solution Explorer 窗口中右击项目名，选择 Add | Existing Item 选项。

(3) 从 C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01 目录中选择 Animal.cs、Cow.cs 和 Chicken.cs 文件，单击 Add 按钮。

(4) 修改这 3 个文件中的名称空间声明，如下所示：

可从 wrox.com 下载源代码

```
namespace Ch11Ex02
```

代码段 Ch11Ex02\Animal.cs、Ch11Ex02\Cow.cs 和 Ch11Ex02\Chicken.cs

(5) 添加一个新类 Animals。

(6) 修改 Animals.cs 中的代码，如下所示：

可从 wrox.com 下载源代码

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex02
{
    public class Animals : CollectionBase
    {
        public void Add(Animal newAnimal)
```

```
        {
            List.Add(newAnimal);
        }

        public void Remove(Animal newAnimal)
        {
            List.Remove(newAnimal);
        }

        public Animals()
        {
        }

        public Animal this[int animalIndex]
        {
            get
            {
                return (Animal)List[animalIndex];
            }
            set
            {
                List[animalIndex] = value;
            }
        }
    }
}
```

代码段 Ch11Ex02\Animals.cs

(7) 修改 Program.cs，如下所示：


```
static void Main(string[] args)
{
    Animals animalCollection = new Animals();
    animalCollection.Add(new Cow("Jack"));
    animalCollection.Add(new Chicken("Vera"));
    foreach (Animal myAnimal in animalCollection)
    {
        myAnimal.Feed();
    }
    Console.ReadKey();
}
```

代码段 Ch11Ex02\Program.cs

(8) 执行应用程序，其结果如图 11-2 所示。

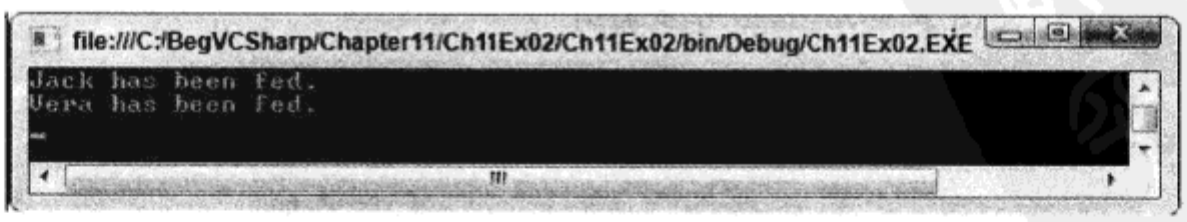


图　11-2

示例的说明

这个示例使用上一节详细介绍的代码，实现类 Animals 中强类型化的 Animal 对象集合。Main() 中的代码仅实例化了一个 Animals 对象 animalCollection，添加了两个项(它们分别是 Cow 和 Chicken 的实例)，再使用 foreach 循环调用这两个对象继承于基类 Animal 的 Feed()方法。

11.1.4　给 CardLib 添加 Cards 集合

第 10 章创建了一个类库项目 Ch10CardLib，它包含一个表示扑克牌的 Card 类和一个表示一幅扑克牌的 Deck 类，这个 Deck 类是 Card 类的集合，且实现为一个简单的数组。

本章给这个库添加一个新类，并把类库重命名为 Ch11CardLib。这个新类 Cards 是 Card 对象的一个定制集合，并拥有本章前面介绍的各种功能。在 C:\BegVCSharp\Chapter11 目录中创建一个新的类库Ch11CardLib，再从 Project | Add Existing Item 中选择 C:\BegVCSharp\Chapter10\Ch10CardLib\Ch10CardLib 目录中的 Card.cs、Deck.cs、Suit.cs 和 Rank.cs 文件，把它们添加到项目中。与第10章介绍的这个项目的上一个版本相同，这里也不使用标准的“试一试”格式介绍这些变化。读者可以在本章的下载代码中打开这个项目的版本，直接查看代码。

在把源文件从Ch10CardLib复制到Ch11CardLib 中时，必须修改名称空间声明，以引用 Ch11CardLib。对用于测试的 Ch10CardLib 控制台应用程序也要进行这个修改。

本章下载代码中的一个项目包含了对 Ch11CardClient 进行的各种扩展。其代码放在各个区段中，如果读者要实践一下，可以取消这些区段的注释。

如果要自己创建这个项目，就应添加一个新类 Cards，修改 Cards.cs 中的代码，如下所示：


```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11CardLib
{
   public class Cards : CollectionBase
   {
      public void Add(Card newCard)
      {
         List.Add(newCard);
      }

      public void Remove(Card oldCard)
      {
         List.Remove(oldCard);
      }

      public Cards()
      {
      }
```

```
public Card this[int cardIndex]
{
   get
   {
      return (Card)List[cardIndex];
   }
   set
   {
      List[cardIndex] = value;
   }
}

/// <summary>
/// Utility method for copying card instance into another Cards
/// instance - used in Deck.Shuffle(). This implementation assume that
/// source and target collections are the same size.
/// </summary>
public void CopyTo(Cards targetCards)
{
    for (int index = 0; index < this.Count; index++)
    {
        targetCards[index] = this[index];
    }
}

/// <summary>
/// Check to see if the Cards collection contains a particular card.
/// This calls the Contains method of the ArrayList for the collection,
/// which we access through the InnerList property.
/// </summary>
public bool Contains(Card card)
{
    return InnerList.Contains(card);
}
   }
}
```

代码段Ch11CardLib\Cards.cs

然后，需要修改 Deck.cs，以利用这个新集合，而不是数组：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11CardLib
{
   public class Deck
   {
      private Cards cards = new Cards();

      public Deck()
      {
          // line of code removed here.
```

```
        for (int suitVal = 0; suitVal < 4; suitVal++)
        {
           for (int rankVal = 1; rankVal < 14; rankVal++)
           {
              cards.Add(new Card((Suit)suitVal, (Rank)rankVal));
           }
        }
     }

     public Card GetCard(int cardNum)
     {
        if (cardNum >= 0 && cardNum <= 51)
           return cards[cardNum];
        else
           throw (new System.ArgumentOutOfRangeException("cardNum", cardNum,
                 "Value must be between 0 and 51."));
     }

     public void Shuffle()
     {
        Cards newDeck = new Cards();
        bool[] assigned = new bool[52];
        Random sourceGen = new Random();
        for (int i = 0; i < 52; i++)
        {
           int sourceCard = 0;
           bool foundCard = false;
           while (foundCard == false)
           {
              sourceCard = sourceGen.Next(52);
              if (assigned[sourceCard] == false)
                 foundCard = true;
           }
           assigned[sourceCard] = true;
           newDeck.Add(cards[sourceCard]);
        }
        newDeck.CopyTo(cards);
     }
  }
}
```

代码段 Ch11CardLib\Deck.cs

在此不需要进行很多修改。其中的大多数修改都涉及到改变洗牌逻辑，才能把 Cards 中随机的一张牌添加到新 Cards 集合 newDeck 的开头，而不是把 cards 集合中顺序位置的一张牌添加 newDeck 集合的随机位置上。

Ch10CardLib 解决方案的客户控制台应用程序 Ch10CardClient 可以使用这个新库得到与以前相同的结果，因为 Deck 的方法签名没有改变。这个类库的客户程序现在可以使用 Cards 集合类，而不是依赖 Card 对象数组，例如，在扑克牌游戏应用程序中定义一手牌。

11.1.5 关键字值集合和 IDictionary

除了 IList 接口外，集合还可以实现类似的 IDictionary 接口，允许项通过关键字值(如字符串名)

进行索引，而不是通过一个索引。这也可以使用索引符来完成，但这次的索引符参数是与存储的项相关联的关键字，而不是 int 索引，这样集合就更便于用户使用了。

与索引的集合一样，可以使用一个基类简化 IDictionary 接口的实现，这个基类就是 DictionaryBase，它也实现 IEnumerable 和 ICollection，提供了对任何集合都相同的基本集合处理功能。

DictionaryBase 与 CollectionBase 一样，实现通过其支持的接口获得的一些成员(但不是全部成员)。DictionaryBase 也实现 Clear 和 Count 成员，但不实现 RemoveAt()。这是因为 RemoveAt()是 IList 接口中的一个方法，不是 IDictionary 接口中的一个方法。但是，IDictionary 有一个 Remove()方法，这是一个应在基于 DictionaryBase 的定制集合类上实现的方法。

下面的代码是 Animals 类的另一个版本，这次该类派生于 DictionaryBase。下面代码包括 Add()、Remove()和一个通过关键字访问的索引符的实现代码：

```
public class Animals : DictionaryBase
{
   public void Add(string newID, Animal newAnimal)
   {
      Dictionary.Add(newID, newAnimal);
   }

   public void Remove(string animalID)
   {
      Dictionary.Remove(animalID);
   }

   public Animals()
   {
   }

   public Animal this[string animalID]
   {
      get
      {
         return (Animal)Dictionary[animalID];
      }
      set
      {
         Dictionary[animalID] = value;
      }
   }
}
```

这些成员的区别如下：

- **Add()**——带有两个参数：一个关键字和一个值，存储在一起。字典集合有一个继承于 DictionaryBase 的成员Dictionary，这个成员是一个IDictionary 接口，有自己的 Add()方法，该方法带有两个 object 参数。我们的实现代码带有一个 string 值(作为关键字)和一个 Animal 对象(作为与该关键字存储在一起的数据)。
- **Remove()**——带有一个关键字参数，而不是对象引用。带有指定关键字值的项被删除。

- **Indexer**——使用一个字符串关键字值，而不是一个索引，用于通过Dictionary 的继承成员来访问存储的项，这里仍需要进行数据类型转换。

基于 DictionaryBase 的集合和基于CollectionBase 的集合之间的另一个区别是 foreach 的工作方式略有区别。上一节的集合可以直接从集合中提取Animal 对象。使用foreach 和 DictionaryBase 派生类可以提供 DictionaryEntry 结构，这是在 System.Collections 名称空间中定义的另一个类型。要得到 Animal 对象本身，就必须使用这个结构的 Value 成员，也可以使用结构的 Key 成员得到相关的关键字。要使代码等价于前面的代码：

```
foreach (Animal myAnimal in animalCollection)
{
  Console.WriteLine("New {0} object added to custom collection, " +
                    "Name = {1}", myAnimal.ToString(), myAnimal.Name());
}
```

需要使用以下代码：

```
foreach (DictionaryEntry myEntry in animalCollection)
{
  Console.WriteLine("New {0} object added to custom collection, " +
                    "Name = {1}", myEntry.Value.ToString(),
                    ((Animal)myEntry.Value).Name);
}
```

重写这段代码，以便直接通过 foreach 提取 Animal 对象，这有许多方式，最简单的方式是实现一个迭代器。

11.1.6　迭代器

本章前面介绍过，IEnumerable 接口负责使用 foreach 循环。在 foreach 循环中使用定制类通常有很多优点，而不仅仅使用集合类，例如，本章前面几节介绍的集合类。

但是，重写使用 foreach 循环的方式，或者提供定制的实现方式，并不一定很简单。为了说明这一点，下面深入研究一下 foreach 循环。在 foreach 循环中，迭代集合 collectionObject 的过程如下：

(1) 调用 collectionObject.GetEnumerator()，返回一个 IEnumerator 引用。这个方法可以通过 IEnumerable 接口的实现代码来获得，但这是可选的。

(2) 调用所返回的 IEnumerator 接口的 MoveNext()方法。

(3) 如果 MoveNext()方法返回 true，就使用 IEnumerator 接口的 Current 属性获取对象的一个引用，用于 foreach 循环。

(4) 重复前面两步，直到 MoveNext()方法返回 false 为止，此时循环停止。

所以，为了在类中进行这些操作，必须重写几个方法，跟踪索引，维护Current 属性，以及执行其他一些操作，这要做许多工作。

一个较为简单的替代方法是使用迭代器。使用迭代器将有效地在后台生成许多代码，正确地完成所有的任务。而且，使用迭代器的语法掌握起来非常容易。

迭代器的定义是，它是一个代码块，按顺序提供了要在foreach 循环中使用的所有值。一般情况下，这个代码块是一个方法，但也可以使用属性访问器和其他代码块作为迭代器。这里为了简单起见，仅介绍方法。

无论代码块是什么，其返回类型都是有限制的。与期望正好相反，这个返回类型与所枚举的对象类型不同。例如，在表示 Animal 对象集合的类中，迭代器块的返回类型不可能是 Animal。两种可能的返回类型是前面提到的接口类型 IEnumerable 和 IEnumerator。使用这两个类型的场合是：

- 如果要迭代一个类，可使用方法 GetEnumerator()，其返回类型是 IEnumerator。
- 如果要迭代一个类成员，例如一个方法，则使用 IEnumerable。

在迭代器块中，使用 yield 关键字选择要在 foreach 循环中使用的值。其语法如下：

```
yield return value;
```

利用这个信息就足以建立一个非常简单的示例，如下所示：


```
public static IEnumerable SimpleList()
{
   yield return "string 1";
   yield return "string 2";
   yield return "string 3";
}

public static void Main(string[] args)
{
   foreach (string item in SimpleList())
      Console.WriteLine(item);

   Console.ReadKey();
}
```

代码段 SimpleIterators\Program.cs

为了亲手测试这些代码，应给 System.Collections 名称空间添加一个 using 语句，或者使用完全限定的 System.Collections.IEnumerable 接口。这些代码在本章下载代码的 SimpleIterators 项目中。

在此，静态方法 SimpleList()就是迭代器块。它是一个方法，所以使用 IEnumerable 返回类型。SimpleList()使用 yield 关键字为使用它的 foreach 块提供了 3 个值，每个值都输出到屏幕上，结果如图 11-3 所示。

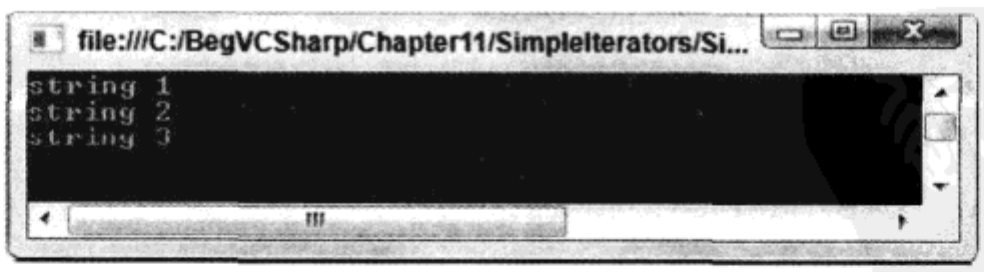


图　11-3

显然，这个迭代器并不是特别有用，但它允许查看执行过程，了解实现代码有多简单。看看代码，读者可能会疑惑代码是如何知道返回 string 类型的项。实际上，并没有返回 string 类型的项，而

是返回了object 类型的值。因为 object 是所有类型的基类，也就是说，可以从 yield 语句中返回任意类型。

但是，编译器非常聪明，所以我们可以把返回值解释为 foreach 循环需要的任何类型。这里代码需要 string 类型的值，所以这就是我们要使用的值。如果修改一行 yield 代码，让它返回一个整数，就会在 foreach 循环中出现一个错误类型转换异常。

对于迭代器，还有一点要注意。可以使用下面的语句中断信息返回 foreach 循环的过程：

```
yield break;
```

在遇到迭代器中的这个语句时，迭代器的处理会立即中断，就像 foreach 循环使用它一样。

下面是一个比较复杂但很有用的示例。在这个示例中，要实现一个迭代器，获取素数。

试一试：实现一个迭代器

(1) 在 C:\BegVCSharp\Chapter11 目录中创建一个新控制台应用程序 Ch11Ex03。

(2) 添加一个新类 Primes，修改代码，如下所示：


```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex03
{
  public class Primes
  {
    private long min;
    private long max;

    public Primes() : this(2, 100)
    {
    }

    public Primes(long minimum, long maximum)
    {
      if (min < 2)
        min = 2;
      else
        min = minimum;

      max = maximum;
    }

    public IEnumerator GetEnumerator()
    {
      for (long possiblePrime = min; possiblePrime <= max; possiblePrime++)
      {
        bool isPrime = true;
        for (long possibleFactor = 2; possibleFactor <=
          (long)Math.Floor(Math.Sqrt(possiblePrime)); possibleFactor++)
        {
```

```
                long remainderAfterDivision = possiblePrime % possibleFactor;
                if (remainderAfterDivision == 0)
                {
                   isPrime = false;
                   break;
                }
             }
             if (isPrime)
             {
                yield return possiblePrime;
             }
          }
       }
    }
}
```

代码段Ch11Ex03\Primes.cs

(3) 修改 Program.cs 中的代码，如下所示：


可从
wrox.com
下载源代码

```
static void Main(string[] args)
{
   Primes primesFrom2To1000 = new Primes(2, 1000);
   foreach (long i in primesFrom2To1000)
      Console.Write("{0} ", i);

   Console.ReadKey();
}
```

代码段Ch11Ex03\Program.cs

(4) 执行应用程序，结果如图 11-4 所示。

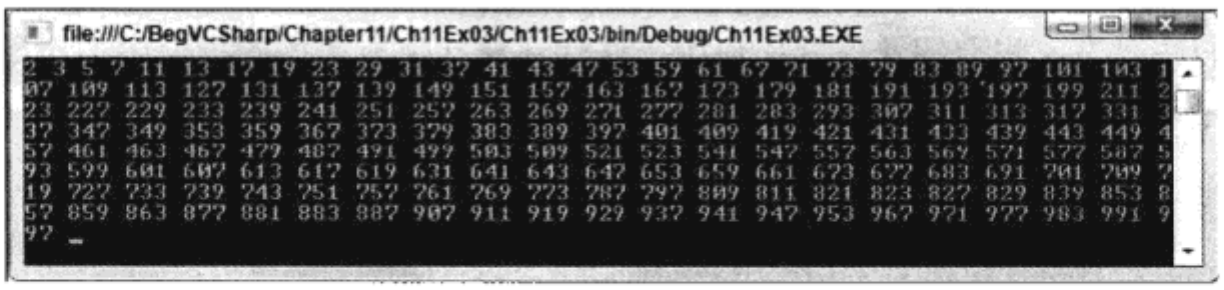


图　11-4

示例的说明

这个示例中的类可以枚举上下限之间的素数集合。封装素数的类利用迭代器提供了这个功能。

Primes 的代码开始时比较简单，用两个字段存储表示搜索范围的最大值和最小值，并使用构造函数设置这些值。注意，最小值是有限制的，它不能小于 2，这是有意义的，因为 2 是最小的素数。相关的代码则全部放在方法 GetEnumerator()中。该方法的签名满足迭代器块的规则，因为它返回 IEnumerator 类型：

```
public IEnumerator GetEnumerator()
{
```

为了提取上下限之间的素数，需要依次测试每个值，所以用一个 for 循环开始：

```
        for (long possiblePrime = min; possiblePrime <= max; possiblePrime++)
        {
```

由于我们不知道某个数是否是素数，所以先假定这个数是素数，再看看它是否是素数。为此，需要看看该数能否被 2 到该数平方根之间的所有数整除。如果能，则该数不是素数，于是测试下一个数。如果该数的确是素数，就使用 yield 把它传送给 foreach 循环。

```
            bool isPrime = true;
            for (long possibleFactor = 2; possibleFactor <=
               (long)Math.Floor(Math.Sqrt(possiblePrime)); possibleFactor++)
            {
               long remainderAfterDivision = possiblePrime % possibleFactor;
               if (remainderAfterDivision == 0)
               {
                  isPrime = false;
                  break;
               }
            }
            if (isPrime)
            {
               yield return possiblePrime;
            }
         }
      }
```

在这段代码中，有一个有趣的地方：如果把上下限设置为非常大的数，在执行应用程序时，就会发现，会一次显示一个结果，中间有暂停，而不是一次显示所有结果。这说明，无论代码在 yield 调用之间是否终止，迭代器代码都会一次返回一个结果。在后台，调用 yield 都会中断代码的执行，当请求另一个值时，也就是当使用迭代器的foreach 循环开始一个新的循环时，代码会恢复执行。

迭代器和集合

前面我们许诺过，将介绍迭代器如何用于迭代存储在字典类型的集合中的对象，无需处理 DictionaryItem 对象。下面是集合类 Animals：

```
public class Animals : DictionaryBase
{
   public void Add(string newID, Animal newAnimal)
   {
      Dictionary.Add(newID, newAnimal);
   }

   public void Remove(string animalID)
   {
      Dictionary.Remove(animalID);
   }

   public Animals()
   {
   }

   public Animal this[string animalID]
   {
```

```
        get
        {
           return (Animal)Dictionary[animalID];
        }
        set
        {
           Dictionary[animalID] = value;
        }
     }
  }
```

可以在这段代码中添加如下简单的迭代器，以便执行预期的操作：


```
public new IEnumerator GetEnumerator()
{
   foreach (object animal in Dictionary.Values)
      yield return (Animal)animal;
}
```

代码段 DictionaryAnimals\Animals.cs

现在可以使用下面的代码迭代集合中的 Animal 对象了：


```
foreach (Animal myAnimal in animalCollection)
{
   Console.WriteLine("New {0} object added to custom collection, " +
                     "Name = {1}", myAnimal.ToString(), myAnimal.Name);
}
```

代码段 DictionaryAnimals\Program.cs

在本章的下载代码中，这些代码位于 DictionaryAnimals 项目中。

11.1.7　深复制

第 9 章通过下面的 GetCopy()方法，介绍了如何使用受保护的方法 System.Object.MemberwiseClone()进行浅复制(shallow copy)。

```
public class Cloner
{
   public int Val;

   public Cloner(int newVal)
   {
      Val = newVal;
   }

   public object GetCopy()
   {
      return MemberwiseClone();
```

```
    }
}
```

假定有引用类型的字段，而不是值类型的字段(例如，对象)：

```
public class Content
{
   public int Val;
}

public class Cloner
{
   public Content MyContent = new Content();

   public Cloner(int newVal)
   {
      MyContent.Val = newVal;
   }

   public object GetCopy()
   {
      return MemberwiseClone();
   }
}
```

此时，通过GetCopy()得到的浅复制包括一个字段，它引用的对象与源对象相同。下面的代码使用这个 Cloner 类来说明浅复制引用类型的结果：

```
Cloner mySource = new Cloner(5);
Cloner myTarget = (Cloner)mySource.GetCopy();
Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);
mySource.MyContent.Val = 2;
Console.WriteLine("myTarget.MyContent.Val = {0}", myTarget.MyContent.Val);
```

第 4 行把一个值赋给 mySource.MyContent.Val，它是源对象中公共字段 MyContent 的公共字段 Val。这也改变了 myTarget.MyContent.Val 的值。这是因为 mySource.MyContent 引用了与 myTarget.MyContent 相同的对象实例。上述代码的输出结果如下：

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 2
```

为了解决这个问题，需要执行深复制。修改上面的 GetCopy()方法就可以进行深复制，但最好使用.NET Framework 的标准方式：实现ICloneable 接口，该接口有一个方法 Clone()：这个方法不带参数，返回一个 object 类型的结果，其签名和上面使用的 GetCopy()方法相同。

修改上面的类，以使用下面的深复制代码：

```
public class Content
{
   public int Val;
}

public class Cloner : ICloneable
{
```

```
    public Content MyContent = new Content();

    public Cloner(int newVal)
    {
      MyContent.Val = newVal;
    }

    public object Clone()
    {
      Cloner clonedCloner = new Cloner(MyContent.Val);
      return clonedCloner;
    }
}
```

其中使用包含在源 Cloner 对象中的 Content 对象(MyContent)的 Val 字段，创建一个新 Cloner 对象。这个字段是一个值类型，所以不需要深复制。

使用与上面类似的代码测试浅复制，但用 Clone()替代 GetCopy()，得到如下结果：

```
myTarget.MyContent.Val = 5
myTarget.MyContent.Val = 5
```

这次包含的对象是独立的。注意有时在比较复杂的对象系统中，调用 Clone()是一个递归过程。例如，如果 Cloner 类的 MyContent 字段也需要深复制，就要使用下面的代码：

```
public class Cloner : ICloneable
{
  public Content MyContent = new Content();

  ...

  public object Clone()
  {
    Cloner clonedCloner = new Cloner();
    clonedCloner.MyContent = MyContent.Clone();
    return clonedCloner;
  }
}
```

这里调用了默认的构造函数，以便简化创建一个新Cloner 对象的语法。为了使这段代码能正常工作，还需要在 Content 类上实现 ICloneable 接口。

11.1.8　给 CardLib 添加深复制

下面把上述内容付诸于实践：使用 ICloneable 接口，复制 Card、Cards 和 Deck 对象，这在某些扑克牌游戏中是有用的，因为在这些游戏中不需要让两副扑克牌引用一组相同的 Card 对象，但肯定会使一副扑克牌中的牌序与另一副牌的牌序相同。

在 Ch11CardLib 中，对 Card 类执行复制操作是很简单的，因为只需进行浅复制(Card 只包含值类型的数据，其形式为字段)。我们只需对类定义进行如下修改：

可从 wrox.com 下载源代码

```
public class Card : ICloneable
{
  public object Clone()
  {
```

```
        return MemberwiseClone();
    }
```

代码段 Ch11CardLib\Card.cs

ICloneable 接口的这段实现代码只是一个浅复制，无法确定在 Clone()方法中执行了什么操作，而这正是我们的目的。

接着，需要对 Cards 集合类实现 ICloneable 接口。这个过程稍复杂些，因为涉及到复制源集合中的每个 Card 对象，所以需要进行深复制：


```
public class Cards : CollectionBase, ICloneable
{
    public object Clone()
    {
        Cards newCards = new Cards();
        foreach (Card sourceCard in List)
        {
            newCards.Add(sourceCard.Clone() as Card);
        }
        return newCards;
    }
```

代码段 Ch11CardLib\Cards.cs

最后，需要在 Deck 类上实现 ICloneable 接口。这里存在一个小问题：因为 Deck 类无法修改它包含的扑克牌，所以没有洗牌。例如，无法修改有给定牌序的 Deck 实例。为了解决这个问题，为 Deck 类定义一个新的私有构造函数，在实例化 Deck 对象时，可以给该函数传送指定的 Cards 集合。所以，在这个类中执行复制的代码如下所示：


```
public class Deck : ICloneable
{
    public object Clone()
    {
        Deck newDeck = new Deck(cards.Clone() as Cards);
        return newDeck;
    }

    private Deck(Cards newCards)
    {
        cards = newCards;
    }
```

代码段 Ch11CardLib\Deck.cs

再次用一些简单的客户代码进行测试(与以前一样，这应放在客户项目的 Main()方法中，以便进行测试)：


```
Deck deck1 = new Deck();
Deck deck2 = (Deck)deck1.Clone();
Console.WriteLine("The first card in the original deck is: {0}",
```

```
                    deck1.GetCard(0));
Console.WriteLine("The first card in the cloned deck is: {0}",
                    deck2.GetCard(0));
deck1.Shuffle();
Console.WriteLine("Original deck shuffled.");
Console.WriteLine("The first card in the original deck is: {0}",
                    deck1.GetCard(0));
Console.WriteLine("The first card in the cloned deck is: {0}",
                    deck2.GetCard(0));
Console.ReadKey();
```

代码段 Ch11CardClient\Program.cs

其输出结果如图 11-5 所示。

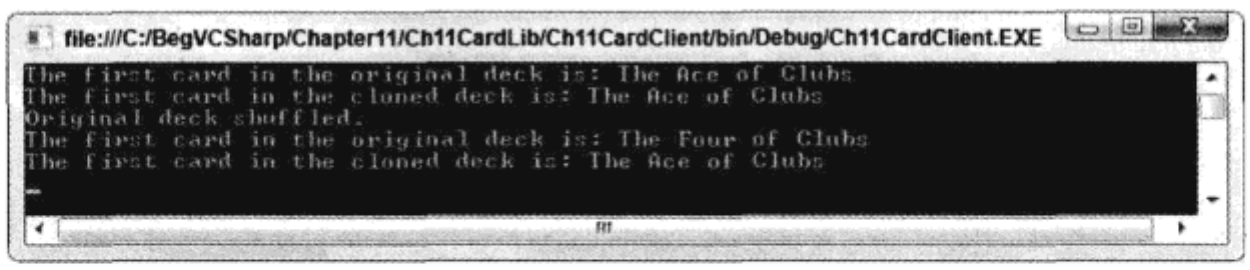


图　11-5

11.2　比较

本节介绍对象之间的两类比较：

- 类型比较
- 值比较

类型比较确定对象是什么，或者对象继承了什么，在 C#编程中，这是非常重要的。把对象传送给方法时，下一步要进行什么操作常常取决于对象的类型。本章和前面的章节都讨论过传送对象的内容，这里将介绍一些更有用的技巧。

值比较我们也见过许多，至少见过简单类型的值比较。在比较对象的值时，情况会变得较为复杂：必须从一开始就定义比较的含义，确定像>这样的运算符在类中会执行什么操作。这在集合中尤其重要，有时我们希望根据某个条件排列对象的顺序，例如按照字母顺序或者根据某个比较复杂的算法来排序。

11.2.1　类型比较

在比较对象时，常常需要了解它们的类型，才能确定是否可以进行值的比较。第 9 章介绍了 GetType()方法，所有的类都从 System.Object 中继承了这个方法，这个方法和 typeof()运算符一起使用，就可以确定对象的类型(并据此执行操作)：

```
if (myObj.GetType() == typeof(MyComplexClass))
{
   // myObj is an instance of the class MyComplexClass.
}
```

前面还提到 ToString()的默认实现方式，ToString()也是从 System.Object 继承来的，该方法可以提供对象类型的字符串表示。也可以比较这些字符串，但这是比较杂乱的方式。

本节将介绍比较值的一种简便方式：is 运算符。它可以提供可读性较高的代码，还可以检查基类。在介绍 is 运算符之前，需要了解处理值类型(与引用类型相反)时后台的一些常见操作：封箱(boxing)和拆箱(unboxing)。

1. 封箱和拆箱

第 8 章讨论了引用类型和值类型之间的区别，第 9 章通过比较结构(值类型)和类(引用类型)进行了说明。封箱(boxing)是把值类型转换为 System.Object 类型，或者转换为由值类型实现的接口类型。拆箱(unboxing)是相反的转换过程。

例如，下面的结构类型：

```
struct MyStruct
{
   public int Val;
}
```

可以把这种类型的结构放在 object 类型的变量中，以封箱它：

```
MyStruct valType1 = new MyStruct();
valType1.Val = 5;
object refType = valType1;
```

其中创建了一个类型为 MyStruct 的新变量(valType1)，并把一个值赋予这个结构的 Val 成员，然后把它封箱在 object 类型的变量(refType)中。

以这种方式封箱变量而创建的对象，包含值类型变量的一个副本的引用，而不包含源值类型变量的引用。要进行验证，可以修改源结构的内容，把对象中包含的结构拆箱到新变量中，检查其内容：

```
valType1.Val = 6;
MyStruct valType2 = (MyStruct)refType;
Console.WriteLine("valType2.Val = {0}", valType2.Val);
```

执行这段代码将得到如下输出结果：

```
valType2.Val = 5
```

但在把一个引用类型赋予对象时，将执行不同的操作。把 MyStruct 改为一个类(不考虑这个类名不合适的情况)，即可看到这种情形：

```
class MyStruct
{
   public int Val;
}
```

如果不修改上面的客户代码(再次忽略名称错误的变量)，就会得到如下输出结果：

```
valType2.Val = 6
```

也可以把值类型封箱到一个接口类型中，只要它们实现这个接口即可。例如，假定 MyStruct 类型实现 IMyInterface 接口，如下所示：

```
interface IMyInterface
{
}

struct MyStruct : IMyInterface
{
   public int Val;
}
```

接着把结构封箱到一个 IMyInterface 类型中，如下所示：

```
MyStruct valType1 = new MyStruct();
IMyInterface refType = valType1;
```

然后使用一般的数据类型转换语法拆箱它：

```
MyStruct ValType2 = (MyStruct)refType;
```

从这些示例中可以看出，封箱是在没有用户干涉的情况下进行的(即不需要编写任何代码)，但拆箱一个值需要进行显式转换，即需要进行数据类型转换(封箱是隐式的，所以不需要进行数据类型转换)。

读者可能想知道为什么要这么做。封箱非常有用，有两个非常重要的原因。首先，它允许在项的类型是 object 的集合(如 ArrayList)中使用值类型。其次，有一个内部机制允许在值类型上调用 object，例如 int 和结构。

最后需要注意的是，在访问值类型内容前，必须进行拆箱。

2. is 运算符

is 运算符并不是说明对象是某种类型的一种方式，而是可以检查对象是否是给定类型，或者是否可以转换为给定类型，如果是，这个运算符就返回 true。

在前面的示例中，有 Cow 和 Chicken 类，它们都继承于 Animal。使用 is 运算符比较 Animal 类型的对象，如果对象是这 3 种类型中的一种(不仅仅是 Animal)，is 运算符就返回 true。使用前面介绍的 GetType()方法和 typeof()运算符很难做到这一点。

is 运算符的语法如下：

```
<operand> is <type>
```

这个表达式的结果如下：

- 如果<type>是一个类类型，而<operand>也是该类型，或者它继承了该类型，或者它可以封箱到该类型中，则结果为 true。
- 如果<type>是一个接口类型，而<operand>也是该类型，或者它是实现该接口的类型，则结果为 true。
- 如果<type>是一个值类型，而<operand>也是该类型，或者它可以拆箱到该类型中，则结果为 true。

下面用几个示例说明如何使用该运算符。

试一试：使用 is 运算符

(1) 在 C:\BegVCSharp\Chapter11 目录中创建一个新控制台应用程序 Ch11Ex04。

(2) 修改 Program.cs 中的代码，如下所示：

可从 wrox.com 下载源代码

```
namespace Ch11Ex04
{
   class Checker
   {
      public void Check(object param1)
      {
         if (param1 is ClassA)
            Console.WriteLine("Variable can be converted to ClassA.");
         else
            Console.WriteLine("Variable can't be converted to ClassA.");
         if (param1 is IMyInterface)
            Console.WriteLine("Variable can be converted to IMyInterface.");
         else
            Console.WriteLine("Variable can't be converted to IMyInterface.");

         if (param1 is MyStruct)
            Console.WriteLine("Variable can be converted to MyStruct.");
         else
            Console.WriteLine("Variable can't be converted to MyStruct.");
      }
   }

   interface IMyInterface
   {
   }

   class ClassA : IMyInterface
   {
   }

   class ClassB : IMyInterface
   {
   }
   class ClassC
   {
   }

   class ClassD : ClassA
   {
   }

   struct MyStruct : IMyInterface
   {
   }

   class Program
```

```
    {
        static void Main(string[] args)
        {
            Checker check = new Checker();
            ClassA try1 = new ClassA();
            ClassB try2 = new ClassB();
            ClassC try3 = new ClassC();
            ClassD try4 = new ClassD();
            MyStruct try5 = new MyStruct();
            object try6 = try5;
            Console.WriteLine("Analyzing ClassA type variable:");
            check.Check(try1);
            Console.WriteLine("\nAnalyzing ClassB type variable:");
            check.Check(try2);
            Console.WriteLine("\nAnalyzing ClassC type variable:");
            check.Check(try3);
            Console.WriteLine("\nAnalyzing ClassD type variable:");
            check.Check(try4);
            Console.WriteLine("\nAnalyzing MyStruct type variable:");
            check.Check(try5);
            Console.WriteLine("\nAnalyzing boxed MyStruct type variable:");
            check.Check(try6);
            Console.ReadKey();
        }
    }
}
```

代码段 Ch11Ex04\Program.cs

(3) 运行代码，其结果如图 11-6 所示。

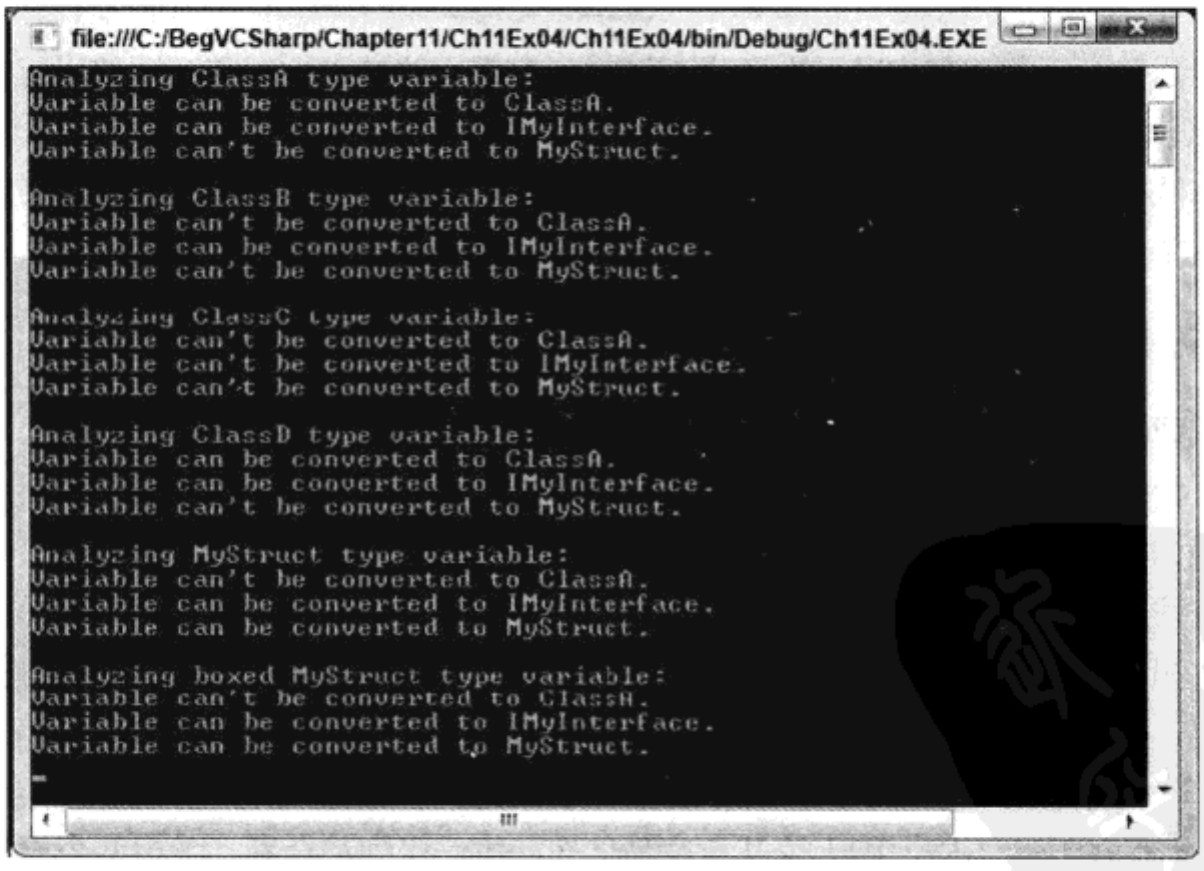


图　11-6

示例的说明

这个示例说明了使用 is 运算符的各种可能的结果。其中定义了 3 个类、一个接口和一个结构，并把它们用作一个类的方法的参数，这个类使用 is 运算符确定它们是否可以转换为 ClassA 类型、接口类型和结构类型。

只有 ClassA 和 ClassD(继承了 ClassA)类型与 ClassA 兼容。如果一个类型没有继承一个类，该类型不会与该类兼容。

ClassA、ClassB 和 MyStruct 类型都实现了 IMyInterface，所以它们都与 IMyInterface 类型兼容。ClassD 继承了 ClassA，所以它们两个也兼容。因此，只有 ClassC 是不兼容的。

最后，只有 MyStruct 类型的变量本身和该类型的封箱变量与 MyStruct 兼容，因为不能把引用类型转换为值类型(当然，我们不能拆箱以前封箱的变量)。

11.2.2　值比较

考虑两个表示人的 Person 对象，它们都有一个 Age 整型属性。下面要比较它们，看看哪个人年龄较大。为此可以使用以下代码：

```
if (person1.Age > person2.Age)
{
   ...
}
```

这是可以的，还有其他方法，例如，使用下面的语法：

```
if (person1 > person2)
{
   ...
}
```

可以使用运算符重载，如本节后面所述。这是一个强大的技术，但应谨慎使用。在上面的代码中，年龄的比较不是非常明显，该段代码还可以比较身高、体重、IQ 等。

另一个方法是使用 IComparable 和 IComparer 接口，它们可以用标准的方式定义比较对象的过程。这是由.NET Framework 中各种集合类提供的方式，是对集合中的对象进行排序的一种绝佳方式。

1. 运算符重载

运算符重载(operator overloading)可以对我们设计的类使用标准的运算符，例如+、>等。这称为重载，因为在使用特定的参数类型时，我们为这些运算符提供了自己的实现代码，其方式与重载方法相同，也是为同名的方法提供不同的参数。

运算符重载非常有用，因为我们可以在运算符重载的实现中执行需要的任何操作，这并不像“把这两个操作数相加”这么简单。稍后介绍一个进一步升级 CardLib 库的示例。我们将提供比较运算符的实现代码，比较两张牌，看看在一圈(扑克牌游戏中的一局)中哪张牌会赢。

因为在许多扑克牌游戏中，一圈取决于牌的花色，这并不像比较牌上的数字那样直接。如果第二张牌与第一张牌的花色不同，则无论其点数是什么，第一张牌都会赢。考虑两个操作数的顺序，就可以实现这种比较。也可以考虑“王牌”的花色，而王牌可以胜过其他的花色，即使该王牌的花色与第一张牌不同，也是如此。也就是说，card1 > card2 是 true(这表示如果 card1 是第一个出牌，

则 card1 胜过了 card2)，并不意味着 card2 > card1 是 false。如果 card1 和 card2 都不是王牌，且属于不同的花色，则这两个比较都是 true。

但我们先看看运算符重载的基本语法。要重载运算符，可给类添加运算符类型成员(它们必须是 static)。一些运算符有多种用途(如 - 运算符就有一元和二元两种功能)，因此我们还指定了要处理多少个操作数，以及这些操作数的类型。一般情况下，操作数的类型与定义运算符的类相同，但也可以定义处理混合类型的运算符，详见稍后的内容。

例如，考虑一个简单类型 AddClass1，如下所示：

```
public class AddClass1
{
   public int val;
}
```

这仅是 int 值的一个包装器(wrapper)，但可以用于说明原理。对于这个类，下面的代码不能编译：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
AddClass1 op3 = op1 + op2;
```

其错误是+运算符不能应用于AddClass1 类型的操作数，因为我们尚未定义要执行的操作。下面的代码则可执行，但得不到预期的结果：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass1 op2 = new AddClass1();
op2.val = 5;
bool op3 = op1 == op2;
```

其中，使用==二元运算符来比较 op1 和op2，看看它们是否引用同一个对象，而不是验证它们的值是否相等。在上述代码中，即使 op1.val 和op2.val 相等，op3 也是 false。

要重载+运算符，可使用下述代码：

```
public class AddClass1
{
   public int val;

   public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
   {
      AddClass1 returnVal = new AddClass1();
      returnVal.val = op1.val + op2.val;
      return returnVal;
   }
}
```

可以看出，运算符重载看起来与标准静态方法声明类似，但它们使用关键字 operator 和运算符本身，而不是一个方法名。现在可以成功地使用+运算符和这个类，如上面的示例所示：

```
AddClass1 op3 = op1 + op2;
```

重载所有的二元运算符都是一样的，一元运算符看起来也是类似的，但只有一个参数：

```
public class AddClass1
{
   public int val;

   public static AddClass1 operator +(AddClass1 op1, AddClass1 op2)
   {
      AddClass1 returnVal = new AddClass1();
      returnVal.val = op1.val + op2.val;
      return returnVal;
   }

   public static AddClass1 operator -(AddClass1 op1)
   {
      AddClass1 returnVal = new AddClass1();
      returnVal.val = -op1.val;
      return returnVal;
   }
}
```

这两个运算符处理的操作数的类型与类相同，返回值也是该类型，但考虑下面的类定义：

```
public class AddClass1
{
   public int val;

   public static AddClass3 operator +(AddClass1 op1, AddClass2 op2)
   {
      AddClass3 returnVal = new AddClass3();
      returnVal.val = op1.val + op2.val;
      return returnVal;
   }
}

public class AddClass2
{
   public int val;
}

public class AddClass3
{
   public int val;
}
```

下面的代码就可以执行：

```
AddClass1 op1 = new AddClass1();
op1.val = 5;
AddClass2 op2 = new AddClass2();
op2.val = 5;
AddClass3 op3 = op1 + op2;
```

可以酌情用这种方式混合类型。但要注意，如果把相同的运算符添加到 AddClass2 中，上面的代码就会失败，因为它弄不清要使用哪个运算符。因此，应注意不要把签名相同的运算符添加到多

个类中。

还要注意，如果混合了类型，操作数的顺序必须与运算符重载的参数顺序相同。如果使用了重载的运算符和顺序错误的操作数，操作就会失败。所以不能像下面这样使用运算符：

```
AddClass3 op3 = op2 + op1;
```

当然，除非提供了另一个重载运算符和倒序的参数：

```
public static AddClass3 operator +(AddClass2 op1, AddClass1 op2)
{
   AddClass3 returnVal = new AddClass3();
   returnVal.val = op1.val + op2.val;
   return returnVal;
}
```

可以重载下述运算符：

- **一元运算符：** +, - , !, ~, ++, - -, true, false
- **二元运算符：** +, - , *, /, %, &, |, ^, <<, >>
- **比较运算符：** ==, !=, <, >, <=, >=

如果重载 true 和 false 运算符，就可以在布尔表达式中使用类，例如，if (op1) {}。

不能重载赋值运算符，例如+=，但这些运算符使用与它们对应的简单运算符，例如+，所以不必担心它们。重载+表示+=像预期的那样执行。=运算符不能重载，因为它有一个基本的用途。但这个运算符与用户定义的转换运算符相关，详见下一节。

也不能重载&& 和 ||，但它们可以在计算中使用对应的运算符&和 |，所以重载&和 | 就足够了。

一些运算符如< 和> 必须成对重载。这就是说，不能重载 <，除非也重载了> 。在许多情况下，可以在这些运算符中调用其他运算符，以减少需要的代码数量(和可能发生的错误)，例如：

```
public class AddClass1
{
   public int val;

   public static bool operator >=(AddClass1 op1, AddClass1 op2)
   {
      return (op1.val >= op2.val);
   }

   public static bool operator <(AddClass1 op1, AddClass1 op2)
   {
      return !(op1 >= op2);
   }

   // Also need implementations for <= and > operators.
}
```

在较复杂的运算符定义中，这可以减少代码行数。这也意味着，如果后来决定修改这些运算符

的实现，需要改动的代码将较少。

这同样适用于==和!=，但对于这些运算符，常常需要重写 Object.Equals()和 Object.GetHash Code()，因为这两个函数也可以用于比较对象。重写这些方法，可以确保无论类的用户使用什么技术，都能得到相同的结果。这不太重要，但应增加进来，以保证其完整性。它需要下述非静态重写方法：

```
public class AddClass1
{
   public int val;

   public static bool operator ==(AddClass1 op1, AddClass1 op2)
   {
      return (op1.val == op2.val);
   }

   public static bool operator !=(AddClass1 op1, AddClass1 op2)
   {
      return !(op1 == op2);
   }

   public override bool Equals(object op1)
   {
      return val == ((AddClass1)op1).val;
   }

   public override int GetHashCode()
   {
      return val;
   }
}
```

GetHashCode()可根据其状态，获取对象实例的一个唯一的 int 值。这里使用 val 就可以了，因为它也是一个 int 值。

注意，Equals()使用 object 类型参数。我们需要使用这个签名，否则就将重载这个方法，而不是重写它。类的用户仍可以访问默认的实现代码。这样就必须使用数据类型转换得到需要的结果。这常常需要使用本章前面讨论的 is 运算符检查对象类型，代码如下所示：

```
public override bool Equals(object op1)
{
   if (op1 is AddClass1)
   {
      return val == ((AddClass1)op1).val;
   }
   else
   {
      throw new ArgumentException(
         "Cannot compare AddClass1 objects with objects of type "
         + op1.GetType().ToString());
   }
}
```

在这段代码中，如果传送给 Equals 的操作数的类型错误，或者不能转换为正确类型，就会抛出

一个异常。当然，这可能并不是我们希望的操作。我们要比较一个类型的对象和另一个类型的对象，此时需要更多的分支结构。另外，可能只允许对类型完全相同的两个对象进行比较，这需要对第一个 if 语句进行如下修改：

```
if (op1.GetType() == typeof(AddClass1))
```

2. 给 CardLib 添加运算符重载

现在再次升级 Ch11CardLib 项目，给 Card 类添加运算符重载。但首先给 Card 类添加额外的字段，指定某花色比其他花色大，使 A 有更高的级别。把这些字段指定为静态，因为设置了它们后，它们就可以应用到所有的 Card 对象上：


```
public class Card
{
   /// <summary>
   /// Flag for trump usage. If true, trumps are valued higher
   /// than cards of other suits.
   /// </summary>
   public static bool useTrumps = false;

   /// <summary>
   /// Trump suit to use if useTrumps is true.
   /// <summary>
   public static Suit trump = Suit.Club;

   /// <summary>
   /// Flag that determines whether aces are higher than kings or lower
   /// than deuces.
   /// <summary>
   public static bool isAceHigh = true;
```

代码段 Ch11CardLib\Card.cs

这些规则应用于应用程序中每个 Deck 的所有 Card 对象上。因此，两个 Deck 中的 Card 不可能遵守不同的规则。这适用于这个类库，但是确实可以做出这样的假设：如果一个应用程序要使用不同的规则，可以自行维护这些规则；例如，在切换牌时，设置 Card 的静态成员。

完成后，就要给 Deck 类再添加几个构造函数，以便用不同的特性初始化扑克牌：


```
   /// <summary>
   /// Nondefault constructor. Allows aces to be set high.
   /// <summary>
   public Deck(bool isAceHigh) : this()
   {
      Card.isAceHigh = isAceHigh;
   }

   /// <summary>
   /// Nondefault constructor. Allows a trump suit to be used.

   /// <summary>
   public Deck(bool useTrumps, Suit trump) : this()
```

```
{
   Card.useTrumps = useTrumps;
   Card.trump = trump;
}

/// <summary>
/// Nondefault constructor. Allows aces to be set high and a trump suit
/// to be used.
/// <summary>
public Deck(bool isAceHigh, bool useTrumps, Suit trump) : this()
{
   Card.isAceHigh = isAceHigh;
   Card.useTrumps = useTrumps;
   Card.trump = trump;
}
```

代码段Ch11CardLib\Deck.cs

每个构造函数都使用第 9 章介绍的: this()语法来定义，这样，无论如何，默认的构造函数总是会在非默认的构造函数之前调用，初始化扑克牌。

接着，给 Card 类添加运算符重载(和推荐的重写代码):

可从 wrox.com 下载源代码

```
public static bool operator ==(Card card1, Card card2)
{
   return (card1.suit == card2.suit) && (card1.rank == card2.rank);
}

public static bool operator !=(Card card1, Card card2)
{
   return !(card1 == card2);
}

public override bool Equals(object card)
{
   return this == (Card)card;
}
public override int GetHashCode()
{
   return 13*(int)rank + (int)suit;
}

public static bool operator >(Card card1, Card card2)
{
   if (card1.suit == card2.suit)
   {
      if (isAceHigh)
      {
         if (card1.rank == Rank.Ace)
         {
            if (card2.rank == Rank.Ace)
               return false;
            else
               return true;
         }
```

```
            else
            {
                if (card2.rank == Rank.Ace)
                    return false;
                else
                    return (card1.rank > card2.rank);
            }
        }
        else
        {
            return (card1.rank > card2.rank);
        }
    }
    else
    {
        if (useTrumps && (card2.suit == Card.trump))
            return false;
        else
            return true;
    }
}

public static bool operator <(Card card1, Card card2)
{
    return !(card1 >= card2);
}

public static bool operator >=(Card card1, Card card2)
{
    if (card1.suit == card2.suit)
    {
        if (isAceHigh)
        {
            if (card1.rank == Rank.Ace)
            {
                return true;
            }
            else
            {
                if (card2.rank == Rank.Ace)
                    return false;
                else
                    return (card1.rank >= card2.rank);
            }
        }
        else

        {
            return (card1.rank >= card2.rank);
        }
    }
    else
    {
        if (useTrumps && (card2.suit == Card.trump))
            return false;
```

```
        else
            return true;
    }
}

public static bool operator <=(Card card1, Card card2)
{
    return !(card1 > card2);
}
```

代码段 Ch11CardLib\Card.cs

这段代码没有什么需要特别关注的，只是>和>=重载运算符的代码比较长。如果单步执行>运算符的代码，就可以看到它的执行情况，明白为什么需要这些步骤。

比较两张牌 card1 和 card2，其中 card1 假定为先出的牌。如前所述，在使用王牌时，这是很重要的，因为王牌胜过其他牌，即使非王牌比较大，也是这样。当然，如果两张牌的花色相同，则王牌是否也是该花色就不重要了，所以这是我们要进行的第一个比较：

```
public static bool operator >(Card card1, Card card2)
{
   if (card1.suit == card2.suit)
   {
```

如果静态的 isAceHigh 标记为 true，就不能直接通过 Rank 枚举中的值比较牌的点数了。因为 A 的点数在这个枚举中是 1，比其他牌都小。此时就需要如下步骤：

- 如果第一张牌是 A，就检查第二张牌是否也是 A。如果是，则第一张牌就胜不过第二张牌。如果第二张牌不是 A，则第一张牌胜出：

```
if (isAceHigh)
{
   if (card1.rank == Rank.Ace)
   {
      if (card2.rank == Rank.Ace)
         return false;
      else
         return true;
   }
```

- 如果第一张牌不是 A，也需要检查第二张牌是否是 A。如果是，则第二张牌胜出；否则，就可以比较牌的点数，因为此时已不比较 A 了：

```
   else
   {
      if (card2.rank == Rank.Ace)
         return false;
      else
         return (card1.rank > card2.rank);
   }
}
```

- 另外，如果 A 不是最大的，就只需比较牌的点数：

```
        else
        {
            return (card1.rank > card2.rank);
        }
```

代码的其余部分主要考虑的是 card1 和 card2 花色不同的情况。其中静态 useTrumps 标记是非常重要的。如果这个标记是 true，且 card2 是王牌，则可以肯定，card1 不是王牌(因为这两张牌有不同的花色)，王牌总是胜出，所以 card2 比较大：

```
    else
    {
        if (useTrumps && (card2.suit == Card.trump))
            return false;
```

如果 card2 不是王牌(或者 useTrumps 是 false)，则 card1 胜出，因为它是最先出的牌：

```
        else
            return true;
    }
}
```

另有一个运算符(>=)使用与此类似的代码，除此之外的其他运算符都非常简单，所以不需要详细分析它们。

下面的简单客户代码测试这些运算符(把它放在客户项目的 Main()函数中进行测试，就像前面 CardLib 示例的客户代码那样)：


```
Card.isAceHigh = true;
Console.WriteLine("Aces are high.");
Card.useTrumps = true;
Card.trump = Suit.Club;
Console.WriteLine("Clubs are trumps.");

Card card1, card2, card3, card4, card5;
card1 = new Card(Suit.Club, Rank.Five);
card2 = new Card(Suit.Club, Rank.Five);
card3 = new Card(Suit.Club, Rank.Ace);
card4 = new Card(Suit.Heart, Rank.Ten);
card5 = new Card(Suit.Diamond, Rank.Ace);

Console.WriteLine("{0} == {1} ? {2}",
    card1.ToString(), card2.ToString(), card1 == card2);
Console.WriteLine("{0} != {1} ? {2}",
    card1.ToString(), card3.ToString(), card1 != card3);
Console.WriteLine("{0}.Equals({1}) ? {2}",
    card1.ToString(), card4.ToString(), card1.Equals(card4));
Console.WriteLine("Card.Equals({0}, {1}) ? {2}",
    card3.ToString(), card4.ToString(), Card.Equals(card3, card4));
Console.WriteLine("{0} > {1} ? {2}",
    card1.ToString(), card2.ToString(), card1 > card2);
Console.WriteLine("{0} <= {1} ? {2}",
    card1.ToString(), card3.ToString(), card1 <= card3);
Console.WriteLine("{0} > {1} ? {2}",
    card1.ToString(), card4.ToString(), card1 > card4);
```

```
Console.WriteLine("{0} > {1} ? {2}",
    card4.ToString(), card1.ToString(), card4 > card1);
Console.WriteLine("{0} > {1} ? {2}",
    card5.ToString(), card4.ToString(), card5 > card4);
Console.WriteLine("{0} > {1} ? {2}",
    card4.ToString(), card5.ToString(), card4 > card5);

Console.Readkey();
```

代码段 Ch11CardClient\Program.cs

其结果如图 11-7 所示。

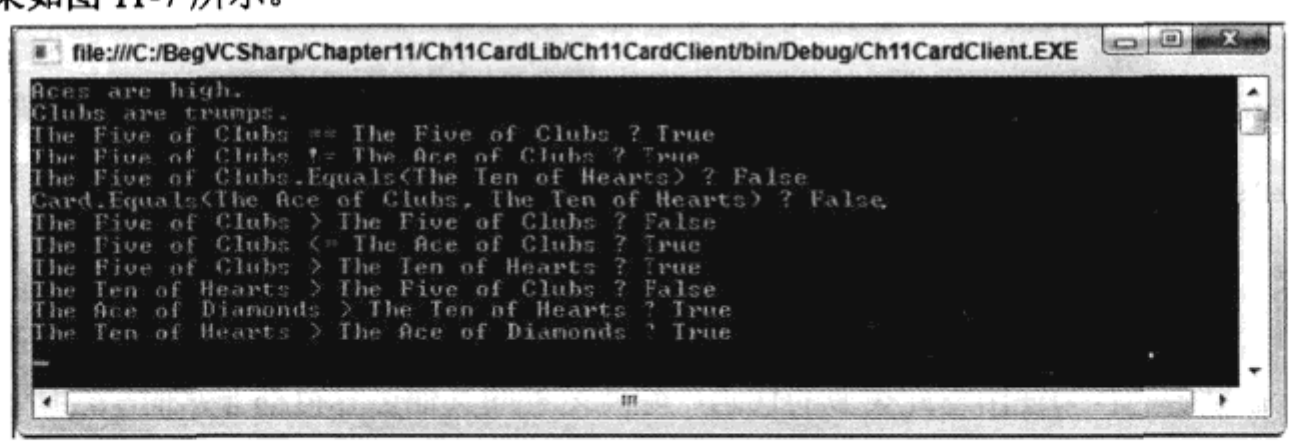


图　11-7

在两种情况下，在应用运算符时都考虑了指定的规则。这在输出结果的最后 4 行中尤其明显，说明王牌总是胜过其他牌。

3. IComparable 和 IComparer 接口

IComparable 和 IComparer 接口是.NET Framework 中比较对象的标准方式。这两个接口之间的差别如下：

- IComparable 在要比较的对象的类中实现，可以比较该对象和另一个对象。
- IComparer 在一个单独的类中实现，可以比较任意两个对象。

一般使用 IComparable 给出类的默认比较代码，使用其他类给出非默认的比较代码。

IComparable 提供了一个方法 CompareTo()，这个方法接受一个对象。例如，实现可以为实现方法传送一个 Person 对象，以便确定这个人是否比当前的人更年老还是更年轻。实际上，这个方法返回一个 int，所以也可以确定第二个人与当前的人的年龄差：

```
if (person1.CompareTo(person2) == 0)
{
   Console.WriteLine("Same age");
}
else if (person1.CompareTo(person2) > 0)
{
   Console.WriteLine("person 1 is Older");
}
else
{
   Console.WriteLine("person1 is Younger");
}
```

IComparer 也提供了一个方法 Compare()。这个方法接受两个对象，返回一个整型结果，这与 CompareTo()相同。对于支持 IComparer 的对象，可以使用下面的代码：

```
if (personComparer.Compare(person1, person2) == 0)
{
   Console.WriteLine("Same age");
}
else if (personComparer.Compare(person1, person2) > 0)
{
   Console.WriteLine("person 1 is Older");
}
else
{
   Console.WriteLine("person1 is Younger");
}
```

在这两种情况下，提供给方法的参数是 System.Object 类型。也就是说，可以比较其他任意类型的两个对象。所以，在返回结果之前，通常需要进行某种类型比较，如果使用了错误的类型，还会抛出异常。

.NET Framework 在类 Comparer 上提供了 IComparer 接口的默认实现方式，类 Comparer 位于 System.Collections 名称空间中，可以对简单类型以及支持 IComparable 接口的任意类型进行特定文化的比较。例如，可以通过下面的代码使用它：

```
string firstString = "First String";
string secondString = "Second String";
Console.WriteLine("Comparing '{0}' and '{1}', result: {2}",
   firstString, secondString,
   Comparer.Default.Compare(firstString, secondString));

int firstNumber = 35;
int secondNumber = 23;
Console.WriteLine("Comparing '{0}' and '{1}', result: {2}",
   firstNumber, secondNumber,
   Comparer.Default.Compare(firstNumber, secondNumber));
```

这里使用 Comparer.Default 静态成员获取 Comparer 类的一个实例，接着使用 Compare()方法比较前两个字符串，之后比较两个整数，结果如下：

```
Comparing 'First String' and 'Second String', result: -1
Comparing '35' and '23', result: 1
```

在字母表中，F 在 S 的前面，所以 F“小于”S，第一个比较的结果就是 - 1。同样，35 大于 23，所以结果是 1。注意这里的结果并未给出相差的幅度。

在使用 Comparer 时，必须使用可以比较的类型。例如，试图比较 firstString 和 firstNumber 就会生成一个异常。

下面是有关这个类的一些注意事项：

- 检查传送给 Comparer.Compare()的对象，看看它们是否支持 IComparable。如果支持，就使用该实现代码。
- 允许使用 null 值，它表示“小于”其他对象。

- 字符串根据当前文化来处理。要根据不同的文化(或语言)处理字符串，Comparer 类必须使用其构造函数进行实例化，以便传送指定所使用的文化的 System.Globalization.CultureInfo 对象。
- 字符串在处理时要区分大小写。如果要以不区分大小写的方式来处理它们，就需要使用 CaseInsensitiveComparer 类，该类以相同的方式工作。

4. 使用 IComparable 和 IComparer 接口对集合排序

许多集合类可以用对象的默认比较方式进行排序，或者用定制方法来排序。ArrayList 就是一个示例，它包含方法 Sort()，这个方法使用时可以不带参数，此时使用默认的比较方式，也可以给它传送 IComparer 接口，以比较对象对。

在给 ArrayList 填充了简单类型时，例如整数或字符串，就会进行默认的比较。对于自己的类，必须在类定义中实现 IComparable，或者创建一个支持 IComparer 的类，来进行比较。

注意，System.Collection 名称空间中的一些类，包括 CollectionBase，都没有提供排序方法。如果要对派生于这个类的集合排序，就必须多做一些工作，自己给内部的 List 集合排序。

下面的示例说明如何使用默认的和非默认的比较方式给列表排序。

试一试：给列表排序

(1) 在 C:\BegVCSharp\Chapter11 目录中创建一个新控制台应用程序 Ch11Ex05。

(2) 添加一个新类 Person，修改代码，如下所示：


```
namespace Ch11Ex05
{
   class Person : IComparable
   {
      public string Name;
      public int Age;

      public Person(string name, int age)
      {
         Name = name;
         Age = age;
      }

      public int CompareTo(object obj)
      {
         if (obj is Person)
         {
            Person otherPerson = obj as Person;
            return this.Age - otherPerson.Age;
         }
         else
         {
            throw new ArgumentException(
               "Object to compare to is not a Person object.");
         }
      }
   }
```

```
}
```

代码段 Ch11Ex05\Person.cs

(3) 添加一个新类 PersonComparerName，修改代码，如下所示：


```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex05
{
   public class PersonComparerName : IComparer
   {
      public static IComparer Default = new PersonComparerName();

      public int Compare(object x, object y)
      {
         if (x is Person && y is Person)
         {
            return Comparer.Default.Compare(
               ((Person)x).Name, ((Person)y).Name);
         }
         else
         {
            throw new ArgumentException(
               "One or both objects to compare are not Person objects.");
         }
      }
   }
}
```

代码段 Ch11Ex05\PersonComparerName.cs

(4) 修改 Program.cs 中的代码，如下所示：


```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch11Ex05
{
   class Program
   {
      static void Main(string[] args)
      {
         ArrayList list = new ArrayList();
         list.Add(new Person("Jim", 30));
         list.Add(new Person("Bob", 25));
         list.Add(new Person("Bert", 27));
```

```
list.Add(new Person("Ernie", 22));

Console.WriteLine("Unsorted people:");
for (int i = 0; i < list.Count; i++)
{
   Console.WriteLine("{0} ({1})",
      (list[i] as Person).Name, (list[i] as Person).Age);
}
Console.WriteLine();

Console.WriteLine(
   "People sorted with default comparer (by age):");
list.Sort();
for (int i = 0; i < list.Count; i++)
{
   Console.WriteLine("{0} ({1})",
      (list[i] as Person).Name, (list[i] as Person).Age);
}
Console.WriteLine();

Console.WriteLine(
   "People sorted with nondefault comparer (by name):");
list.Sort(PersonComparerName.Default);
for (int i = 0; i < list.Count; i++)
{
   Console.WriteLine("{0} ({1})",
      (list[i] as Person).Name, (list[i] as Person).Age);
}

Console.ReadKey();
        }
    }
}
```

代码段 Ch11Ex05\Program.cs

(5) 执行代码，结果如图 11-8 所示。

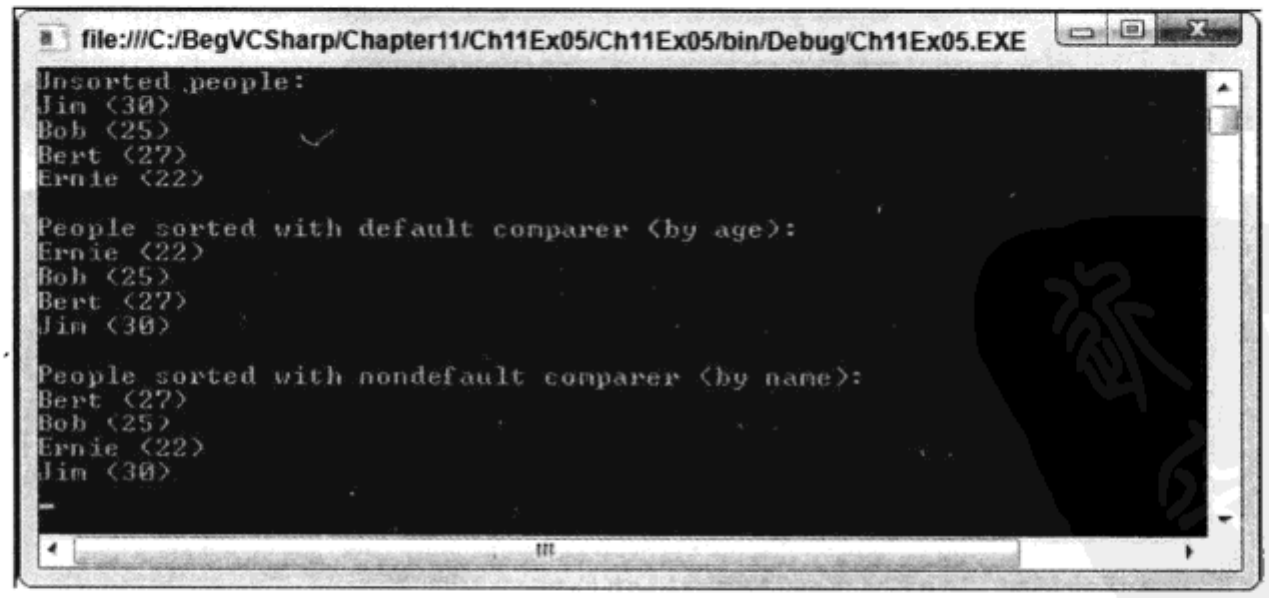


图　11-8

示例的说明

在这个示例中，包含 Person 对象的 ArrayList 用两种不同的方式排序。调用不带参数的 ArrayList.Sort()方法，将使用默认的比较方式，即使用Person 类中的CompareTo()方法(因为这个类实现了 IComparable)：

```
public int CompareTo(object obj)
{
   if (obj is Person)
   {
      Person otherPerson = obj as Person;
      return this.Age - otherPerson.Age;
   }
   else
   {
      throw new ArgumentException(
         "Object to compare to is not a Person object.");
   }
}
```

这个方法首先检查其参数是否能与 Person 对象比较，即该对象是否能转换为 Person 对象。如果遇到问题，就抛出一个异常。否则，就比较两个 Person 对象的 Age 属性。

接着，使用实现了 IComparer 的 PersonComparerName 类，执行非默认的比较排序。这个类有一个公共的静态字段，以方便使用：

```
public static IComparer Default = new PersonComparerName();
```

它可以使用 PersonComparerName.Default 获取一个实例，就像前面的 Comparer 类一样。这个类的 CompareTo()方法如下：

```
public int Compare(object x, object y)
{
   if (x is Person && y is Person)
   {
      return Comparer.Default.Compare(
         ((Person)x).Name, ((Person)y).Name);
   }
   else
   {
      throw new ArgumentException(
         "One or both objects to compare are not Person objects.");
   }
}
```

这里也是首先检查参数，看看它们是否是 Person 对象，如果不是，就抛出一个异常；如果是，就使用默认的 Comparer 对象比较两个 Person 对象的字符串字段 Name。

11.3　转换

到目前为止，在需要把一种类型转换为另一种类型时，使用的都是类型转换。而这并不是唯一

的方式。在计算过程中，int 可以隐式转换为 long 或 double，采用相同的方式还可以定义所创建的类(隐式或显式)转换为其他类的方式。为此，可以重载转换运算符，其方式与本章前面重载其他运算符的方式相同。本节第一部分就介绍重载方式。本节还将介绍另一个有用的运算符：as 运算符，它一般适用于引用类型的转换。

11.3.1 重载转换运算符

除了重载如上所述的数学运算符之外，还可以定义类型之间的隐式和显式转换。如果要在不相关的类型之间转换，这是必须的，例如，如果在类型之间没有继承关系，也没有共享接口，这就是必须的。

下面定义 ConvClass1 和ConvClass2 之间的隐式转换，即编写下列代码：

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = op1;
```

另外，还可以定义一个显式转换：

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = (Convclass2)op1;
```

例如，考虑下面的代码：

```
public class ConvClass1
{
   public int val;

   public static implicit operator ConvClass2(Convclass1 op1)
   {
      ConvClass2 returnVal = new ConvClass2();
      returnVal.val = op1.val;
      return returnVal;
   }
}

public class ConvClass2
{
   public double val;

   public static explicit operator ConvClass1(ConvClass2 op1)
   {
      ConvClass1 returnVal = new ConvClass1();
      checked {returnVal.val = (int)op1.val;};
      return returnVal;
   }
}
```

其中，ConvClass1 包含一个 int 值，ConvClass2 包含一个 double 值。int 值可以隐式转换为 double 值，所以可以在 ConvClass1 和 ConvClass2 之间定义一个隐式转换。但反过来就不行了，应把 ConvClass2 和 ConvClass1 之间的转换定义为显式转换。

在代码中，用关键字 implicit 和 explicit 来指定这些转换，如上所示。对于这些类，下面的代码就很好：

```
ConvClass1 op1 = new ConvClass1();
op1.val = 3;
ConvClass2 op2 = op1;
```

但反向的转换需要进行下述显式数据类型转换：

```
ConvClass2 op1 = new ConvClass2();
op1.val = 3e15;
ConvClass1 op2 = (ConvClass1)op1;
```

如果在显式转换中使用了checked 关键字，则上述代码将产生一个异常，因为op1 的 val 属性值太大，不能放在 op2 的 val 属性中。

11.3.2　as 运算符

as 运算符使用下面的语法，把一种类型转换为指定的引用类型：

```
<operand> as <type>
```

这只适用于下列情况：

- <operand> 的类型是<type>类型
- <operand> 可以隐式转换为<type>类型
- <operand> 可以封箱到<type>类型中

如果不能从<operand>转换为<type>，则表达式的结果就是 null。

基类到派生类的转换可以使用显式转换来进行，但这并不总是有效的。考虑前面示例中的两个类 ClassA 和 ClassD，其中 ClassD 派生于 ClassA：

```
class ClassA : IMyInterface
{
}

class ClassD : ClassA
{
}
```

下面的代码使用 as 运算符把 obj1 中存储的 ClassA 实例转换为 ClassD 类型：

```
ClassA obj1 = new ClassA();
ClassD obj2 = obj1 as ClassD;
```

则 obj2 的结果为 null。

还可以使用多态性把 ClassD 实例存储在 ClassA 类型的变量中。下面的代码演示了这一点，ClassA 类型的变量包含 ClassD 类型的实例，使用 as 运算符把 ClassA 类型的变量转换为 ClassD 类型。

```
ClassD obj1 = new ClassD();
ClassA obj2 = obj1;
ClassD obj3 = obj2 as ClassD;
```

这次 obj3 最后包含与 obj1 相同的对象引用，而不是 null。

因此，as 运算符非常有用，因为下面使用简单类型转换的代码会抛出一个异常：

```
ClassA obj1 = new ClassA();
ClassD obj2 = (ClassD)obj1;
```

与此代码等价的 as 代码会把 null 值赋予 obj2，不会抛出异常。这表示，下面的代码(使用本章前面开发的两个类：Animal 和派生于 Animal 的一个类 Cow)在 C#应用程序中是很常见的：

```
public void MilkCow(Animal myAnimal)
{
   Cow myCow = myAnimal as Cow;
   if (myCow != null)
   {
      myCow.Milk();
   }
   else
   {
      Console.WriteLine("{0} isn't a cow, and so can't be milked.",
          myAnimal.Name);
   }
}
```

这要比检查异常要简单得多！

11.4　小结

本章介绍的许多技巧都可以使OOP 应用程序更强大、更有趣。尽管这些技巧要花一定的时间才能掌握，但它们可以使类更容易使用，简化了编写其他代码的任务。

本章介绍的每个论题都有许多用途。读者可能在几乎所有应用程序中见过某种形式的集合，如果要处理类型相同的一组对象，则创建类型安全的集合可以使任务更容易完成。介绍了集合后，我们又介绍了如何添加索引符和迭代器，以访问集合中的对象。

比较和转换是另一个很费时的领域。我们介绍了各种比较方式，以及封箱和拆箱的一些基本功能，还讨论了如何重载比较和转换运算符，如何利用列表排序把事物链接在一起。

第 12 章将介绍一个全新的内容：泛型，通过它们可以创建自动定制自身的类，动态地处理所选的类型。这对于集合来说非常有用，还会介绍如何使用泛型集合极大地简化本章的许多代码。

11.5　练习

(1) 创建一个集合类 People ，它是下述 Person 类的集合，该集合中的项可以通过一个字符串索引符来访问，该字符串索引符是人名，与 Person.Name 属性相同：

```
public class Person
{
   private string name;
   private int age;

   public string Name
   {
      get
      {
         return name;
      }
      set
      {
```

```
            name = value;
        }
    }

    public int Age
    {
        get
        {
            return age;
        }
        set
        {
            age = value;
        }
    }
}
```

(2) 扩展上一题中的 Person 类，重载>、<、>=和<=运算符，比较 Person 实例的 Age 属性。

(3) 给 People 类添加 GetOldest()方法，使用练习(2)中定义的重载运算符，返回其 Age 属性值为最大的 Person 对象数组(1 个或多个对象，因为对于这个属性而言，多个项可以有相同的值)。

(4) 在 People 类上实现 ICloneable 接口，提供深度复制功能。

(5) 给 People 类添加一个迭代器，在下面的 foreach 循环中获取所有成员的年龄：

```
foreach(int age in myPeople.Ages)
{
  //Display ages.
}
```

附录 A 给出了练习答案。

11.6　本章要点

主　题	重 要 概 念
定义集合	集合是可以包含其他类的实例的类。要定义集合，可以从 CollectionBase 中派生，或者自己实现集合接口，例如 IEnumerable、ICollection 和 IList。一般需要为集合定义一个索引器，以使用 collection.[index]语法来访问集合成员
字典	也可以定义关键字值集合，即字典，字典中的每一项都有一个关联的键。在字典中，键可以用于标识一项，而无需使用该项的索引。定义字典时，可以实现 IDictionary，或者从 DictionaryBase 中派生类
迭代器	可以实现一个迭代器，来控制循环代码如何在其循环过程中获取值。要迭代一个类，需要实现 GetEnumerator()方法，其返回类型是 IEnumerator。要迭代类的成员，例如方法，可以使用 IEnumerable 返回类型。在迭代器的代码块中，使用 yield 关键字返回值
类型比较	使用 GetType()方法可以获得对象的类型，使用 typeof()运算符可以获得类的类型。可以比较这些类型值。还可以使用 is 运算符确定对象是否与某个类类型兼容
值比较	如果希望类的实例可以用标准的 C#运算符进行比较，就必须在类定义中重载这些运算符。对于其他类型的值比较，可以使用实现了 IComparable 或 IComparer 接口的类。这些接口特别适用于集合的排序
as 运算符	可以使用 as 运算符把一个值转换为引用类型。如果不能进行转换，as 运算符就返回 null 值

第12章

泛　型

本章内容：

- 泛型的含义
- 如何使用.NET Framework 提供的一些泛型类
- 如何定义自己的泛型
- 变体如何与泛型一起工作

C#第 1 版中最受诟病的一个方面是缺乏对泛型(generics)的支持。C++中的泛型(在该语言中称为模板)很早就被公认为是完成任务的最佳方式。它可以在编译期间由一个类型定义派生出许多特定的类型，这节省了大量的时间和精力。不知道是什么原因，泛型没有纳入 C#的第 1 版，C#也因此受到很多批评。也许是因为泛型是一种很难掌握的技术，也许是开发人员觉得不需要泛型。幸好，C# 2.0 版中加入了泛型。泛型并不是真的很难掌握，只是需要用略微不同的方式处理而已。

本章首先介绍泛型的概念，先学习泛型的抽象术语，因为学习泛型的概念对高效使用它是至关重要的。

接着讨论.NET Framework 中的一些泛型类型，这有助于更好地理解其功能和强大之处，以及在代码中需要使用的新语法。然后定义自己的泛型类型，包括泛型类、接口、方法和委托。还要介绍进一步定制泛型类型的其他技术：default 关键字和类型约束。

最后讨论协变(covariance)和抗变(contravariance)，这是 C# 4 新增的两种形式的变体，在使用泛型类时提供了更大的灵活性。

12.1　泛型的概念

为了介绍泛型的概念，说明它们为什么这么有用，先回忆一下第11 章中的集合类。基本集合可以包含在类似 ArrayList 这样的类中，但这些集合是没有类型化的，所以需要把 object 项转换为集合中实际存储的对象类型。继承自 System.Object 的任何对象都可以存储在 ArrayList 中，所以要特别仔细。假定包含在集合中的某些类型可能导致抛出异常，代码逻辑崩溃。前面介绍的技术可以处理

这个问题，包括检查对象类型所需的代码。

但是，更好的解决办法是一开始就使用强类型化的集合类。这种集合类派生于 CollectionBase，并可以拥有自己的方法，来添加、删除和访问集合的成员，但它可能把集合成员限制为派生于某种基本类型，或者必须支持某个接口。这会带来一个问题。每次创建需要包含在集合中的新类时，就必须执行下列任务之一：

- 使用某个集合类，该类已经定义为可以包含新类型的项。
- 创建一个新的集合类，它可以包含新类型的项，实现所有需要的方法。

一般情况下，新的类型需要额外的功能，我们常常并不需要新的集合类，创建集合类也会花费大量时间。

另一方面，泛型类大大简化了这个问题。泛型类是以实例化过程中提供的类型或类为基础建立的，可以毫不费力地对对象进行强类型化。对于集合，创建“T 类型对象的集合”十分简单，只需编写一行代码即可。不使用下面的代码：

```
CollectionClass items = new CollectionClass();
items.Add(new ItemClass());
```

而是使用：

```
CollectionClass<ItemClass> items = new CollectionClass<ItemClass>();
items.Add(new ItemClass());
```

尖括号语法就是把类型参数传送给泛型类型的方式。在上面的代码中，应把 CollectionClass <ItemClass>看作 ItemClass 的 CollectionClass。当然，本章后面会详细探讨这个语法。

前面的泛型只涉及到集合，实际上泛型非常适合于这个领域，本章在后面介绍 System.Collections.Generic 名称空间时会提及。创建一个泛型类，就可以生成一些方法，它们的签名可以强类型化为我们需要的任何类型，该类型甚至可以是值类型或引用类型，处理各自的操作。还可以把用于实例化泛型类的类型限制为支持某个给定的接口，或派生自某种类型，只允许使用类型的一个子集。泛型并不限于类，还可以创建泛型接口、泛型方法(可以在非泛型类上定义)，甚至泛型委托。这将极大地提高代码的灵活性，正确使用泛型可以显著缩短开发时间。

那么该如何实现泛型呢？通常，在创建类时，它会编译为一个类型，然后在代码中使用。读者可能认为，在创建泛型类时，它必须被编译为许多类型，才能进行实例化。幸好并不是这样：在.NET 中，类有无限多个。在后台，.NET 运行库允许在需要时动态生成泛型类。在通过实例化来请求生成之前，B 的某个泛型类 A 甚至不存在。

对于熟悉C++或者对C++感兴趣的读者来说，这是C++模板和 C#泛型类的一个区别。在 C++中，编译器可以检测出在哪里使用了模板的某个特定类型，例如，模板 B 的 A 类型，然后编译需要的代码，来创建这个类型。而在C#中，所有操作都在运行期间进行。

总之，泛型允许灵活地创建类型，处理一种或多种特定类型的对象，这些类型是在实例化时确定的，否则就使用泛型类型。下面看看如何在实际中使用它们。

12.2 使用泛型

在探讨如何创建自己的泛型类型之前，先介绍.NET Framework 提供的泛型，包括 System.Collections.Generic 名称空间中的类型，这个名称空间已在前面的代码中出现过多次，因为默认情况下它包含在控制台应用程序中。我们还没有使用过这个名称空间中的类型，但下面就要使用了。本节将讨论这个名称空间中的类型，以及如何使用它们创建强类型化的集合，改进已有集合的功能。

首先论述另一个较简单的泛型类型，即可空类型(nullable type)，它解决了值类型的一个小问题。

12.2.1 可空类型

在前面的章节中，介绍了值类型(大多数基本类型，例如，int、double和所有的结构)区别于引用类型(string 和所有的类)的一种方式：值类型必须包含一个值，它们可以在声明之后、赋值之前，在未赋值的状态下存在，但不能以任何方式使用。而引用类型可以是 null。

有时让值类型为空是很有用的(尤其是处理数据库时)，泛型使用 System.Nullable<T>类型提供了使值类型为空的一种方式。例如：

```
System.Nullable<int> nullableInt;
```

这行代码声明了一个变量 nullableInt，它可以拥有 int 变量能包含的任意值，还可以拥有值 null。所以可以编写如下的代码：

```
nullableInt = null;
```

如果 nullableInt 是一个 int 类型的变量，上面的代码是不能编译的。

前面的赋值等价于：

```
nullableInt = new System.Nullable<int>();
```

与其他任何变量一样，无论是初始化为 null(使用上面的语法)，还是通过给它赋值来初始化，都不能在初始化之前使用它。

可以像测试引用类型一样，测试可空类型，看看它们是否为 null：

```
if (nullableInt == null)
{
   ...
}
```

另外，可以使用 HasValue 属性：

```
if (nullableInt.HasValue)
{
   ...
}
```

这不适用于引用类型，即使引用类型有一个 HasValue 属性，也不能使用这种方法，因为引用类型的变量值为 null，就表示不存在对象，当然就不能通过对象来访问这个属性，否则会抛出一个

异常。

使用 Value 属性可以查看可空类型的值。如果 HasValue 是 true，就说明 Value 属性有一个非空值。但如果 HasValue 是 false，就说明变量被赋予了 null，访问 Value 属性会抛出 System.InvalidOperationException 类型的异常。

可空类型非常有用，以致于修改了C#语法。声明可空类型的变量不使用上述语法，而是使用下面的语法：

```
int? nullableInt;
```

int ?是 System.Nullable<int>的缩写，但更便于读取。在后面的章节中就使用了这个语法。

1. 运算符和可空类型

对于简单类型如 int，可以使用+、－等运算符来处理值。而对于可空类型，这是没有区别的：包含在可空类型中的值会隐式转换为需要的类型，使用适当的运算符。这也适用于结构和自己提供的运算符。例如：

```
int? op1 = 5;
int? result = op1 * 2;
```

注意，其中 result 变量的类型也是 int?。下面的代码不会被编译：

```
int? op1 = 5;
int result = op1 * 2;
```

为了使上面的代码正常工作，必须进行显式转换：

```
int? op1 = 5;
int result = (int)op1 * 2;
```

或者通过 Value 属性访问值，需要的代码如下：

```
int? op1 = 5;
int result = op1.Value * 2;
```

只要 op1 有一个值，上面的代码就可以正常运行。如果 op1 是 null，就会生成 System.InvalidOperationException 类型的异常。

这就引出了下一个问题：当运算等式中的一个或两个值是 null 时，例如，下面代码中的 op1，会发生什么情况？

```
int? op1 = null;
int? op2 = 5;
int? result = op1 * op2;
```

答案是：对于除了 bool?之外的所有简单可空类型，该操作的结果是 null，可以把它解释为“不能计算”。对于结构，可以定义自己的运算符来处理这种情况(详见本章后面的内容)。对于 bool?，为&和 | 定义的运算符会得到非空返回值，如表 12-1 所示。

表　12-1

op1	op2	op1 & op2	op1 | op2
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

这些运算符的结果十分符合逻辑，如果不需要知道其中一个操作数的值，就可以计算出结果，则该操作数是否为 null 就不重要。

2. ??运算符

为了进一步减少处理可空类型所需的代码量，使可空变量的处理变得更简单，可以使用??运算符。这个运算符称为空接合运算符(null coalescing operator)，是一个二元运算符，允许给可能等于 null 的表达式提供另一个值。如果第一个操作数不是 null，该运算符就等于第一个操作数，否则，该运算符就等于第二个操作数。下面的两个表达式的作用是相同的：

```
op1 ?? op2
op1 == null ? op2 : op1
```

在这两行代码中，op1 可以是任意可空表达式，包括引用类型和更重要的可空类型。因此，如果可空类型是 null，就可以使用??运算符提供要使用的默认值，如下所示：

```
int? op1 = null;
int result = op1 * 2 ?? 5;
```

在这个示例中，op1 是 null，所以 op1*2 也是 null。但是，??运算符检测到这个情况，并把值 5 赋予 result。这里要特别注意，在结果中放入 int 类型的变量 result 不需要显式转换。??运算符会自动处理这个转换。还可以把??等式的结果放在 int?中：

```
int? result = op1 * 2 ?? 5;
```

在处理可空变量时，??运算符有许多用途，它也是一种提供默认值的便捷方式，不需要使用 if 结构中的代码块或容易引起混淆的三元运算符。

在下面的示例中，将介绍可空类型 Vector。

试一试：可空类型

(1) 在 C:\BegVCSharp\Chapter12 目录中创建一个新控制台应用程序项目 Ch12Ex01。

(2) 在文件 Vector.cs 中添加一个新类 Vector。

(3) 修改 Vector.cs 中的代码，如下所示：

可从
wrox.com
下载源代码

```
public class Vector
{
   public double? R = null;
   public double? Theta = null;

   public double? ThetaRadians
   {
      get
      {
         // Convert degrees to radians.
         return (Theta * Math.PI / 180.0);
      }
   }

   public Vector(double? r, double? theta)
   {
      // Normalize.
      if (r < 0)
      {
         r = -r;
         theta += 180;
      }
      theta = theta % 360;

      // Assign fields.
      R = r;
      Theta = theta;
   }

public static Vector operator +(Vector op1, Vector op2)
{
   try
   {
      // Get (x, y) coordinates for new vector.
      double newX = op1.R.Value * Math.Sin(op1.ThetaRadians.Value)
         + op2.R.Value * Math.Sin(op2.ThetaRadians.Value);
      double newY = op1.R.Value * Math.Cos(op1.ThetaRadians.Value)
         + op2.R.Value * Math.Cos(op2.ThetaRadians.Value);

      // Convert to (r, theta).
      double newR = Math.Sqrt(newX * newX + newY * newY);
      double newTheta = Math.Atan2(newX, newY) * 180.0 / Math.PI;

      // Return result.
      return new Vector(newR, newTheta);
   }
   catch
   {
      // Return "null" vector.
      return new Vector(null, null);
   }
}

public static Vector operator -(Vector op1)
{
```

```
        return new Vector(-op1.R, op1.Theta);
    }

    public static Vector operator -(Vector op1, Vector op2)
    {
        return op1 + (-op2);
    }

    public override string ToString()
    {
        // Get string representation of coordinates.
        string rString = R.HasValue ? R.ToString() : "null";
        string thetaString = Theta.HasValue ? Theta.ToString() : "null";

        // Return (r, theta) string.
        return string.Format("({0}, {1})", rString, thetaString);
    }
}
```

代码段 Ch12Ex01\Vector.cs

(4) 修改 Program.cs 中的代码，如下所示：


```
class Program
{
  static void Main(string[] args)
  {
    Vector v1 = GetVector("vector1");
    Vector v2 = GetVector("vector1");
    Console.WriteLine("{0} + {1} = {2}", v1, v2, v1 + v2);
    Console.WriteLine("{0} - {1} = {2}", v1, v2, v1 - v2);
    Console.ReadKey();
}
static Vector GetVector(string name)
{
    Console.WriteLine("Input {0} magnitude:", name);
    double? r = GetNullableDouble();
    Console.WriteLine("Input {0} angle (in degrees):", name);
    double? theta = GetNullableDouble();
    return new Vector(r, theta);
}

static double? GetNullableDouble()
{
    double? result;
    string userInput = Console.ReadLine();
    try
    {
        result = double.Parse(userInput);
    }
    catch
    {
        result = null;
    }
    return result;
```

```
    }
}
```

代码段 Ch12Ex01\Program.cs

(5) 执行应用程序，给两个矢量(vector)输入值，示例输出结果如图 12-1 所示。

图　12-1

(6) 再次执行应用程序，这次跳过四个值中的至少一个，示例输出结果如图 12-2 所示。

示例的说明

在这个示例中，创建了一个类Vector，它表示带极坐标(有一个幅值和一个角度)的矢量，如图 12-3 所示。

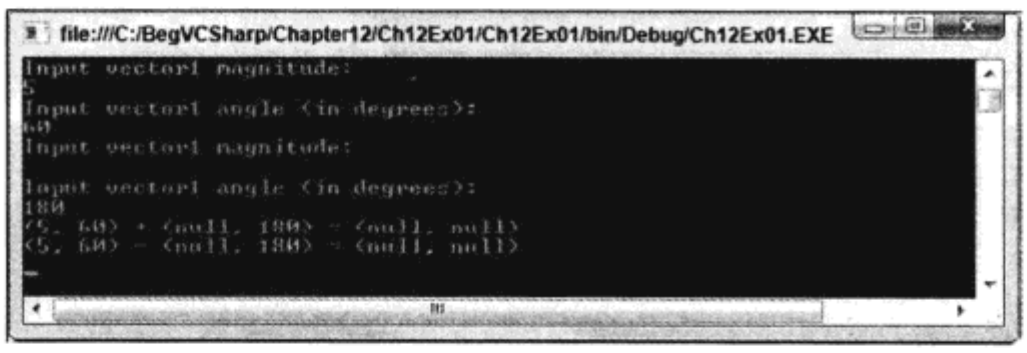


图　12-2

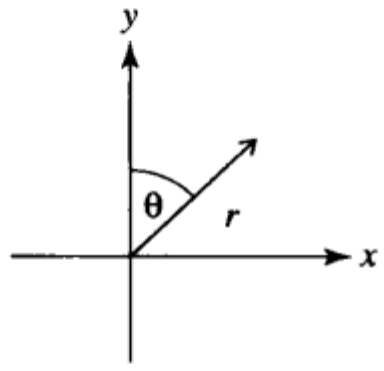


图　12-3

坐标 r 和 θ 在代码中用公共字段 R 和 Theta 表示，其中 Theta 的单位是度(°)。ThetaRadians 用于获取 Theta 的弧度值，这是必需的，因为 Math 类在其静态方法中使用弧度。R 和 Theta 的类型都是 double?，所以它们可以为空。

可从 wrox.com 下载源代码

```
public class Vector
{
   public double? R = null;
   public double? Theta = null;

   public double? ThetaRadians
   {
      get
      {
         // Convert degrees to radians.
         return (Theta * Math.PI / 180.0);
      } ,
   }
```

代码段 Ch12Ex01\Vector.cs

Vector 的构造函数标准化 R 和 Theta 的初始值，然后赋予公共字段。

```
public Vector(double? r, double? theta)
{
   // Normalize.
   if (r < 0)
   {
      r = -r;
      theta += 180;
   }
   theta = theta % 360;

   // Assign fields.
   R = r;
   Theta = theta;
}
```

Vector 类的主要功能是使用运算符重载对矢量进行相加和相减运算，这需要一些非常基本的三角函数知识，这里不解释它们。在代码中，重要的是，如果在获取 R 或 ThetaRadians 的 Value 属性时抛出了异常，即其中一个是 null，就返回“空”矢量。

```
public static Vector operator +(Vector op1, Vector op2)
{
   try
   {
      // Get (x, y) coordinates for new vector.
      ...
   }
   catch
   {
      // Return "null" vector.
      return new Vector(null, null);
   }
}
```

如果组成矢量的一个坐标是 null，该矢量就是无效的，这里用 R 和 Theta 都可为 null 的 Vector 类来表示。Vector 类的其他代码重写了其他运算符，以便扩展相加的功能，使之包含相减操作，再重写 ToString()，获取 Vector 对象的字符串表示。

Program.cs 中的代码测试 Vector 类，让用户初始化两个矢量，再对它们进行相加和相减。如果用户省略了某个值，该值就解释为 null，应用前面提及的规则。

12.2.2 System.Collections.Generic 名称空间

实际上，本书前面的每个应用程序都包含如下名称空间：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

System 名称空间包含.NET 应用程序使用的大多数基本类型。System.Text 名称空间包含与字符串处理和编码相关的类型，System.Linq 名称空间将从第 23 章开始介绍。但 System.Collections.Generic 名称空间包含什么类型？为什么要在默认情况下把它包含在控制台应用程序中？

这个名称空间包含用于处理集合的泛型类型，使用得非常频繁，用 using 语句配置它，使用时就不必添加限定符了。

本章前面提到过这些泛型类型，下面将予以介绍，它们可以使工作更容易完成，可以毫不费力地创建强类型化的集合类。表 12-2 描述了本节要介绍的 System.Collections. Generics 名称空间中的两个类型，本章后面还会详细阐述这个名称空间中的更多类型。

表 12-2

类 型	说 明
List<T>	T 类型对象的集合
Dictionary<K, V>	与 K 类型的键值相关的 V 类型的项的集合

后面还会介绍和这些类一起使用的各种接口和委托。

1. List<T>

List<T>泛型集合类型更加快捷、更易于使用；这样，就不必像上一章那样，从 CollectionBase 中派生一个类，然后实现需要的方法。它的另一个好处是正常情况下需要实现的许多方法(例如，Add())已经自动实现了。

创建 T 类型对象的集合需要如下代码：

```
List<T> myCollection = new List<T>();
```

这就足够了。没有定义类、实现方法和进行其他操作。还可以把 List<T>对象传送给构造函数，在集合中设置项的起始列表。使用这个语法实例化的对象将支持表 12-3 中的方法和属性(其中，提供给 List<T>泛型的类型是 T)。

表 12-3

成 员	说 明
int Count	该属性给出集合中项的个数
void Add(T item)	把一个项添加到集合中
void AddRange(IEnumerable<T>)	把多个项添加到集合中
IList<T> AsReadOnly()	给集合返回一个只读接口
int Capacity	获取或设置集合可以包含的项数
void Clear()	删除集合中的所有项
bool Contains(T item)	确定 item 是否包含在集合中
void CopyTo(T[] array, int index)	把集合中的项复制到数组 array 中，从数组的索引 index 开始
IEnumerator<T> GetEnumerator()	获取一个 IEnumerator<T>实例，用于迭代集合。注意，返回的接口强类型化为 T，所以在 foreach 循环中不需要类型转换

(续表)

成 员	说 明
int IndexOf(T item)	获取 item 的索引，如果集合中并未包含该项，就返回 -1
void Insert(int index, T item)	把 item 插入到集合的指定索引位置上
bool Remove(T item)	从集合中删除第一个 item，并返回 true；如果 item 不包含在集合中，就返回 false
void RemoveAt(int index)	从集合中删除索引 index 处的项

List<T>还有一个 Item 属性，允许进行类似于数组的访问，如下所示：

```
T itemAtIndex2 = myCollectionOfT[2];
```

这个类还支持其他几个方法，但只要掌握了上述知识，就完全可以开始使用该类了。下面的示例将介绍如何在实际中使用 List<T>。

试一试：使用 List<T>

(1) 在 C:\BegVCSharp\Chapter12 目录中创建一个新控制台应用程序 Ch12Ex02。

(2) 在 Solution Explorer 窗口中右击项目名称，选择 Add | Existing Item 选项。

(3) 在 C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01 目录中选择 Animal.cs、Cow.cs 和 Chicken.cs 文件，单击 Add 按钮。

(4) 修改这 3 个文件中的名称空间声明，如下所示：


```
namespace Ch12Ex02
```

Code snippets Ch12Ex02\Animal.cs、Ch12Ex02\Cow.cs 和 Ch12Ex02\Chicken.cs

(5) 修改 Program.cs 中的代码，如下所示：


```
static void Main(string[] args)
{
   List<Animal> animalCollection = new List<Animal>();
   animalCollection.Add(new Cow("Jack"));
   animalCollection.Add(new Chicken("Vera"));
   foreach (Animal myAnimal in animalCollection)
   {
      myAnimal.Feed();
   }
   Console.ReadKey();
}
```

代码段 Ch12Ex02\Program.cs

(6) 执行应用程序，结果与第 11 章的 Ch11Ex02 的结果相同。

示例的说明

这个示例与 Ch11Ex02 只有两个区别。第一个区别是下面的代码：

```
Animals animalCollection = new Animals();
```

被替换为：

```
List<Animal> animalCollection = new List<Animal>();
```

第二个区别比较重要：项目中不再有 Animals 集合类。通过使用泛型的集合类，前面为创建这个类所做的工作现在用一行代码即可完成。

获得相同效果的另一个方法是不修改 Program.cs 中的代码，而是使用 Animals 的如下定义：

```
public class Animals : List<Animal>
{
}
```

这么做的优点是，能较容易地看懂 Program.cs 中的代码，还可以在合适时给 Animals 类添加额外的成员。

为什么不从CollectionBase 中派生类？这是一个很好的问题。实际上，在许多情况下，我们都不会从CollectionBase 中派生类。知道内部工作原理肯定是件好事，因为 List<T>以相同的方式工作，但CollectionBase 主要用于向后兼容。使用CollectionBase 的唯一场合是要更多地控制向类的用户展示的成员。例如，如果希望集合类的 Add()方法使用内部访问修饰符，则使用 CollectionBase 是最佳选择。

也可以把要使用的初始容量(作为 int)传递给 List<T>的构造函数，或者传递给使用 IEnumerable<T>接口的初始项列表。支持这个接口的类包括 List<T>。

2. 对泛型列表进行排序和搜索

对泛型列表进行排序与对其他列表进行排序是一样的。第 11 章介绍了如何使用 IComparer 和 IComparable 接口比较两个对象，然后对该类型的对象列表排序。这里唯一的区别是，可以使用泛型接口 IComparer<T>和 IComparable<T>，它们提供了略有区别、且针对特定类型的方法。表 12-4 列出了它们之间的区别。

表 12-4

泛 型 方 法	非泛型方法	区 别
int IComparable<T>. CompareTo(T otherObj)	int IComparable. CompareTo(object, otherObj)	在泛型版本中是强类型化的
bool IComparable<T>. Equals(T otherObj)	N/A	在非泛型接口中不存在，可以使用 object.Equals()替代
int IComparer<T>. Compare(TobjectA,TobjectB)	int IComparer. Compare(objectobjectA,objectobjectB)	、在泛型版本中是强类型化的

(续表)

泛 型 方 法	非泛型方法	区 别
bool IComparer<T>. Equals(T objectA, T objectB)	N/A	在非泛型接口中不存在，可以改用 object.Equals()
int IComparer<T>. GetHashCode (T objectA)	N/A	在非泛型接口中不存在，可以改用继承的 object. GetHashCode ()

要对 List<T>排序，可以在要排序的类型上提供 IComparable<T>接口，或者提供 IComparer<T>接口。另外，还可以提供泛型委托，作为排序方法。从了解工作原理的角度来看，这非常有趣，因为实现上述接口并不比实现其非泛型版本更麻烦。

一般情况下，给列表排序需要有一个方法来比较两个 T 类型的对象。要在列表中搜索，也需要一个方法来检查 T 类型的对象，看看它是否满足某个条件。定义这样的方法很简单，这里给出两个可以使用的泛型委托类型：

- Comparison<T>：这个委托类型用于排序方法，其返回类型和参数如下：

```
int method (T objectA, T objectB)
```

- Predicate<T>：这个委托类型用于搜索方法，其返回类型和参数如下：

```
bool method(T targetObject)
```

可以定义任意多个这样的方法，使用它们实现 List<T>的搜索和排序方法。下面的示例进行了演示。

试一试：List<T>的搜索和排序

(1) 在 C:\BegVCSharp\Chapter12 目录中创建一个新控制台应用程序 Ch12Ex03。

(2) 在 Solution Explorer 窗口中右击项目名称，选择 Add | Add Existing Item 选项。

(3) 在 C:\BegVCSharp\Chapter12\Ch12Ex01\Ch12Ex01 目录中选择 Vector.cs 文件，单击 Add 按钮。

(4) 修改这个文件中的名称空间声明，如下所示：

```
namespace Ch12Ex03
```

(5) 添加一个新类 Vectors。

(6) 修改 Vectors.cs 中的代码，如下所示：


```
public class Vectors : List<Vector>
{
   public Vectors()
   {
   }

   public Vectors(IEnumerable<Vector> initialItems)
   {
      foreach (Vector vector in initialItems)
```

```
        {
            Add(vector);
        }
    }

    public string Sum()
    {
        StringBuilder sb = new StringBuilder();
        Vector currentPoint = new Vector(0.0, 0.0);
        sb.Append("origin");
        foreach (Vector vector in this)
        {
            sb.AppendFormat(" + {0}", vector);
            currentPoint += vector;
        }
        sb.AppendFormat(" = {0}", currentPoint);
        return sb.ToString();
    }
}
```

代码段Ch12Ex03\Vector.cs

(7) 添加一个新类 VectorDelegates。

(8) 修改 VectorDelegates.cs 中的代码，如下所示：


```
public static class VectorDelegates
{
    public static int Compare(Vector x, Vector y)
    {
        if (x.R > y.R)
        {
            return 1;
        }
        else if (x.R < y.R)
        {
            return -1;
        }
        return 0;
    }

    public static bool TopRightQuadrant(Vector target)
    {
        if (target.Theta >= 0.0 && target.Theta <= 90.0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

代码段Ch12Ex03\VectorDelegates.cs

(9) 修改 Program.cs 中的代码，如下所示：


```
static void Main(string[] args)
{
   Vectors route = new Vectors();
   route.Add(new Vector(2.0, 90.0));
   route.Add(new Vector(1.0, 180.0));
   route.Add(new Vector(0.5, 45.0));
   route.Add(new Vector(2.5, 315.0));

   Console.WriteLine(route.Sum());

   Comparison<Vector> sorter = new Comparison<Vector>
    (VectorDelegates.Compare);
   route.Sort(sorter);
   Console.WriteLine(route.Sum());

   Predicate<Vector> searcher =
      new Predicate<Vector>(VectorDelegates.TopRightQuadrant);
   Vectors topRightQuadrantRoute = new Vectors(route.FindAll(searcher));
   Console.WriteLine(topRightQuadrantRoute.Sum());

   Console.ReadKey();
}
```

代码段 Ch12Ex03\Program.cs

(10) 执行应用程序，结果如图 12-4 所示。

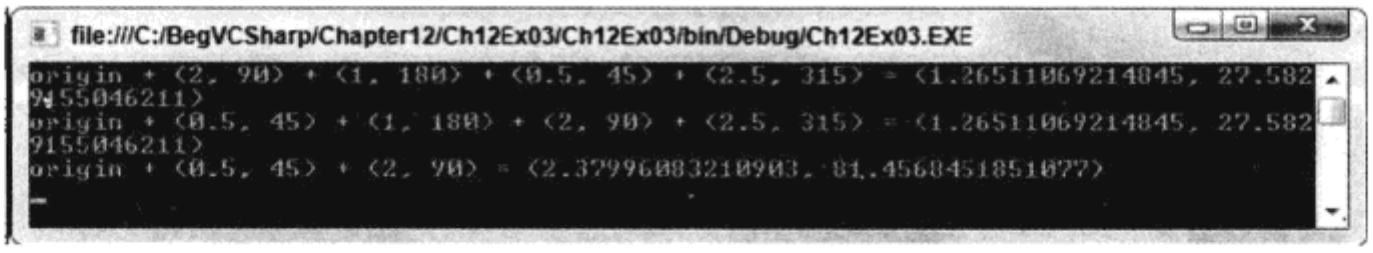


图　12-4

示例的说明

在这个示例中，为 Ch12Ex01 中的 Vector 类创建了一个集合类 Vectors。可以只使用 List <Vector> 类型的变量，但因为需要其他功能，所以使用了一个新类 Vectors，它派生自 List <Vector>，允许添加需要的其他成员。

该类有一个成员 Sum()，依次返回每个矢量的字符串列表，并在最后把它们加在一起(使用源类 Vector 的重载+运算符)。每个矢量都可以看作“方向+距离”，所以这个矢量列表构成了一条有端点的路径。


```
public string Sum()
{
   StringBuilder sb = new StringBuilder();
   Vector currentPoint = new Vector(0.0, 0.0);
   sb.Append("origin");
   foreach (Vector vector in this)
   {
      sb.AppendFormat(" + {0}", vector);
```

```
            currentPoint += vector;
        }
        sb.AppendFormat(" = {0}", currentPoint);
        return sb.ToString();
    }
```

代码段 Ch12Ex03\Vector.cs

这个方法使用 System.Text 名称空间中的简便的 StringBuilder 类来构建响应字符串。这个类包含 Append()和 AppendFormat()等成员(这里使用)，所以很容易组合字符串，其性能也比串联各个字符串要高。使用这个类的 ToString()方法即可获得最终的字符串。

本例还创建了两个用作委托的方法，作为 VectorDelegates 的静态成员。Compare()用于比较(排序)，TopRightQuadrant()用于搜索。稍后在讨论 Program.cs 中的代码时介绍它们。

Main()中的代码首先初始化 Vectors 集合，给它添加几个 Vector 对象：


```
Vectors route = new Vectors();
route.Add(new Vector(2.0, 90.0));
route.Add(new Vector(1.0, 180.0));
route.Add(new Vector(0.5, 45.0));
route.Add(new Vector(2.5, 315.0));
```

代码段 Ch12Ex03\Program.cs

如前所述，Vectors.Sum()方法用于输出集合中的项，这次是按照其初始顺序输出：

```
Console.WriteLine(route.Sum());
```

接着，创建第一个委托 sorter，这个委托属于 Comparison<Vector>类型，因此可以赋予带如下返回类型和参数的方法：

```
int method(Vector objectA, Vector objectB)
```

它匹配 VectorDelegates.Compare()，该方法就是赋予委托的方法。

```
Comparison<Vector> sorter = new Comparison<Vector>
    (VectorDelegates.Compare);
```

Compare()比较两个矢量的大小，如下所示：

```
public static int Compare(Vector x, Vector y)
{
    if (x.R > y.R)
    {
        return 1;
    }
    else if (x.R < y.R)
    {
        return -1;
    }
    return 0;
}
```

这样就可以按大小对矢量排序了：

```
route.Sort(sorter);
Console.WriteLine(route.Sum());
```

应用程序的输出结果符合我们的预期——汇总的结果是一样的，因为无论用什么顺序执行各个步骤，“矢量路径”的端点都是相同的。

然后，进行搜索，获取集合中的一个矢量子集。这需要使用 VectorDelegates.TopRight Quadrant() 来实现：

```
public static bool TopRightQuadrant(Vector target)
{
   if (target.Theta >= 0.0 && target.Theta <= 90.0)
   {
      return true;
   }
   else
   {
      return false;
   }
}
```

如果方法的 Vector 参数值是介于 0° ～90° 之间的 Theta 值，该方法就返回 true，也就是说，它在前面的排序图中指向上或右。

在 Main()方法中，通过 Predicate<Vector>类型的委托使用这个方法，如下所示：

```
Predicate<Vector> searcher =
   new Predicate<Vector>(VectorDelegates.TopRightQuadrant);
Vectors topRightQuadrantRoute = new Vectors(route.FindAll(searcher));
Console.WriteLine(topRightQuadrantRoute.Sum());
```

这需要在 Vectors 中定义构造函数：

```
public Vectors(IEnumerable<Vector> initialItems)
{
   foreach (Vector vector in initialItems)
   {
      Add(vector);
   }
}
```

其中，使用 IEnumerable<Vector>的接口初始化了一个新的 Vectors 集合，这是必需的，因为 List<Vector>.FindAll()返回一个 List<Vector>实例，而不是 Vectors 实例。

搜索的结果是，只返回 Vector 对象的一个子集，所以汇总的结果不同(这正是我们希望的)。使用这些泛型委托类型来排序和搜索泛型集合需要一段时间才能习惯，但代码更流畅、更高效了，代码的结构更富逻辑性。最好花点时间研究本节介绍的技术。

另外，在这个示例中，注意下面的代码：

```
Comparison<Vector> sorter = new Comparison<Vector>(
   VectorDelegates.Compare);
route.Sort(sorter);
```

可以简化为：

```
route.Sort(VectorDelegates.Compare);
```

这样就不需要隐式引用 Comparison<Vector>类型了。实际上，仍会创建这个类型的一个实例，但它是隐式创建的。显然，Sort()方法需要这个类型的实例才能工作，但编译器会认识到这一点，在我们提供的方法中自动创建该类型的实例。此时，对 VectorDelegates.Compare()的引用(没有括号)称为方法组。在许多情况下，都可以使用方法组以这种方式隐式地创建委托，使代码变得更容易读取。

3. Dictionary<K, V>

这个类型可以定义键/值对的集合。与本章前面介绍的其他泛型集合类型不同，这个类需要实例化两个类型，分别用于键和值，以表示集合中的各个项。

实例化 Dictionary<K, V>对象后，就可以像在继承自 DictionaryBase 的类上那样，对它执行相同的操作，但要使用已有的类型安全的方法和属性。例如，可以使用强类型化的 Add()方法添加键/值对。

```
Dictionary<string, int> things = new Dictionary<string, int>();
things.Add("Green Things", 29);
things.Add("Blue Things", 94);
things.Add("Yellow Things", 34);
things.Add("Red Things", 52);
things.Add("Brown Things", 27);
```

可以使用 Keys 和 Values 属性迭代集合中的键和值：

```
foreach (string key in things.Keys)
{
  Console.WriteLine(key);
}

foreach (int value in things.Values)
{
  Console.WriteLine(value);
}
```

还可以迭代集合中的各个项，把每个项作为一个 KeyValuePair<K, V>实例来获取，这与第 11 章介绍的 DictionaryEntry 对象十分相似：

```
foreach (KeyValuePair<string, int> thing in things)
{
  Console.WriteLine("{0} = {1}", thing.Key, thing.Value);
}
```

对于 Dictionary<K, V>要注意的一点是，每个项的键都必须是唯一的。如果要添加的项的键与已有项的键相同，就会抛出 ArgumentException 异常。所以，Dictionary<K, V>允许把 IComparer<K>接口传递给其构造函数，如果要把自己的类用作键，且它们不支持 IComparable 或 IComparable<K>接口，或者要使用非默认的过程比较对象，就必须把 IComparer<K>接口传递给其构造函数。例如，在上面的示例中，可以使用不区分大小写的方法来比较字符串键：

```
Dictionary<string, int> things =
  new Dictionary<string, int>(StringComparer.CurrentCultureIgnoreCase);
```

如果使用下面的键，就会得到一个异常：

```
things.Add("Green Things", 29);
things.Add("Green things", 94);
```

也可以给构造函数传递初始容量(使用 int)或项的集合(使用 IDictionary<K,V>接口)。

4. 修改 CardLib，以使用泛型集合类

对前几章创建的 CardLib 项目可以进行简单的修改，即修改 Cards 集合类，以使用一个泛型集合类，这将减少许多代码行。对 Cards 的类定义需要做如下修改：


```
public class Cards : List<Card>, ICloneable
{
   ...
}
```

代码段 Ch12CardLib\Cards.cs

还可以删除 Cards 的所有方法，但 Clone()和 CopyTo()除外，因为 Clone()是 ICloneable 需要的方法，而 List<Card>提供的 CopyTo()版本处理的是 Card 对象数组，而不是 Cards 集合。需要对 Clone()做一些轻微的修改，因为 List<T>没有定义 List 属性：

```
public object Clone()
{
   Cards newCards = new Cards();
   foreach (Card sourceCard in this)
   {
      newCards.Add(sourceCard.Clone() as Card);
   }
   return newCards;
}
```

这里没有列出代码，因为这是很简单的修改，CardLib 的更新版本为 Ch12CardLib，它和第 11 章的客户代码包含在本章的下载代码中。

12.3 定义泛型类型

利用前面介绍的泛型知识，足以创建自己的泛型了。前面的许多代码都涉及到泛型类型，您还看到了多个使用泛型语法的实例。本节将定义如下内容：

- 泛型类
- 泛型接口
- 泛型方法
- 泛型委托

在定义泛型类型的过程中，还将讨论处理如下问题的一些更高级技术：

- default 关键字
- 约束类型

- 从泛型类中继承
- 泛型运算符

12.3.1　定义泛型类

要创建泛型类，只需在类定义中包含尖括号语法：

```
class MyGenericClass<T>
{
   ...
}
```

其中 T 可以是任意标识符，只要遵循通常的 C#命名规则即可，例如，不以数字开头等。但一般只使用 T。泛型类可以在其定义中包含任意多个类型，它们用逗号分隔开，例如：

```
class MyGenericClass<T1, T2, T3>
{
   ...
}
```

定义了这些类型之后，就可以在类定义中像使用其他类型那样使用它们。可以把它们用作成员变量的类型、属性或方法等成员的返回类型以及方法变元(argument)的参数类型(parameter type)等。例如：

```
class MyGenericClass<T1, T2, T3>
{
   private T1 innerT1Object;

   public MyGenericClass(T1 item)
   {
      innerT1Object = item;
   }

   public T1 InnerT1Object
   {
      get
      {
         return innerT1Object;
      }
   }
}
```

其中，类型 T1 的对象可以传递给构造函数，只能通过 InnerT1Object 属性对这个对象进行只读访问。注意，不能假定类提供了什么类型。例如，下面的代码就不会编译：

```
class MyGenericClass<T1, T2, T3>
{
   private T1 innerT1Object;

   public MyGenericClass()
   {
      innerT1Object = new T1();
   }
```

```
    public T1 InnerT1Object
    {
      get
      {
        return innerT1Object;
      }
    }
}
```

我们不知道 T1 是什么，也就不能使用它的构造函数，它甚至可能没有构造函数，或者没有可公共访问的默认构造函数。如果不使用涉及本节后面介绍的高级技术的复杂代码，则只能对 T1 进行如下假设：可以把它看作继承自 System.Object 的类型或可以封箱到 System.Object 中的类型。

显然，这意味着不能对这个类型的实例进行非常有趣的操作，或者对为 MyGenericClass 泛型类提供的其他类型进行有趣的操作。不使用反射(这是用于在运行期间检查类型的高级技术，本章不介绍它)，就只能使用下面的代码：

```
public string GetAllTypesAsString()
{
  return "T1 = " + typeof(T1).ToString()
    + ", T2 = " + typeof(T2).ToString()
    + ", T3 = " + typeof(T3).ToString();
}
```

可以做一些其他工作，尤其是对集合进行操作，因为处理对象组是非常简单的，不需要对对象类型进行任何假设，这是为什么存在本章前面介绍的泛型集合类的一个原因。

另一个需要注意的限制是，在比较为泛型类型提供的类型值和 null 时，只能使用运算符==和!=。例如，下面的代码会正常工作：

```
public bool Compare(T1 op1, T1 op2)
{
  if (op1 != null && op2 != null)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

其中，如果T1是一个值类型，则总是假定它是非空的，于是在上面的代码中，Compare 总是返回 true。但是，下面试图比较两个变元 op1 和 op2 的代码将不能编译：

```
public bool Compare(T1 op1, T1 op2)
{
  if (op1 == op2)
  {
    return true;
  }
  else
  {
```

```
            return false;
        }
    }
```

其原因是这段代码假定T1 支持==运算符。这说明，要对泛型进行实际的操作，需要更多地了解类中使用的类型。

1. default 关键字

要确定用于创建泛型类实例的类型，需要了解一个最基本的情况：它们是引用类型还是值类型。若不知道这个情况，就不能用下面的代码赋予 null 值：

```
public MyGenericClass()
{
   innerT1Object = null;
}
```

如果T1 是值类型，则 innerT1Object 不能取 null 值，所以这段代码不会编译。幸好，开发人员考虑到了这个问题，使用 default 关键字(本书前面在 switch 结构中使用过它)的新用法解决了它。该新用法如下：

```
public MyGenericClass()
{
   innerT1Object = default(T1);
}
```

其结果是，如果 innerT1Object 是引用类型，就给它赋予 null 值；如果它是值类型，就给它赋予默认值。对于数字类型，这个默认值是 0；而结构根据其各个成员的类型，以相同的方式初始化为 0 或 null。default 关键字允许对必须使用的类型进行更多的操作，但为了更进一步，还需要限制所提供的类型。

2. 约束类型

前面用于泛型类的类型称为无绑定(unbounded)类型，因为没有对它们进行任何约束。而通过约束(constraining)类型，可以限制可用于实例化泛型类的类型，这有许多方式。例如，可以把类型限制为继承自某个类型。回顾前面使用的 Animal、Cow 和 Chicken 类，您可以把一个类型限制为 Animal 或继承自 Animal，则下面的代码是正确的：

```
MyGenericClass<Cow> = new MyGenericClass<Cow>();
```

但下面的代码不能编译：

```
MyGenericClass<string> = new MyGenericClass<string>();
```

在类定义中，这可以使用 where 关键字来实现：

```
class MyGenericClass<T> where T : constraint
{
   ...
}
```

其中 constraint 定义了约束。可以用这种方式提供许多约束，各个约束间用逗号分隔开：

```
class MyGenericClass<T> where T : constraint1, constraint2
{
   ...
}
```

还可以使用多个 where 语句，定义泛型类需要的任意类型或所有类型上的约束：

```
class MyGenericClass<T1, T2> where T1 : constraint1 where T2 : constraint2
{
   ...
}
```

约束必须出现在继承说明符的后面：

```
class MyGenericClass<T1, T2> : MyBaseClass, IMyInterface
   where T1 : constraint1 where T2 : constraint2
{
   ...
}
```

表 12-5 中列出了一些可用的约束。

表 12-5

约 束	定 义	用 法 示 例
struct	类型必须是值类型	在类中，需要值类型才能起作用，例如，T 类型的成员变量是 0，表示某种含义
class	类型必须是引用类型	在类中，需要引用类型才能起作用，例如，T 类型的成员变量是 null，表示某种含义
base-class	类型必须是基类或继承自基类。可以给这个约束提供任意类名	在类中，需要继承自基类的某种基本功能，才能起作用
interface	类型必须是接口或实现了接口	在类中，需要接口公开的某种基本功能，才能起作用
new()	类型必须有一个公共的无参构造函数	在类中，需要能实例化 T 类型的变量，例如在构造函数中实例化

如果new()用作约束，它就必须是为类型指定的最后一个约束。

可以通过 base-class 约束，把一个类型参数用作另一个类型参数的约束，如下所示：

```
class MyGenericClass<T1, T2> where T2 : T1
{
   ...
}
```

其中，T2 必须与 T1 的类型相同，或者继承自 T1。这称为裸类型约束(naked type constraint)，表示一个泛型类型参数用作另一个类型参数的约束。

类型约束不能循环，例如：

```
class MyGenericClass<T1, T2> where T2 : T1 where T1 : T2
{
   ...
}
```

这段代码不能编译。下面的示例将定义和使用一个泛型类，该类使用前面几章介绍的 Animal 类系列。

试一试：定义泛型类

(1) 在 C:\BegVCSharp\Chapter12 目录中创建一个新的控制台应用程序 Ch12Ex04。

(2) 在 Solution Explorer 窗口中右击项目名称，选择 Add | Add Existing Item 选项。

(3) 从 C:\BegVCSharp\Chapter12\Ch12Ex02\Ch12Ex02 目录中选择 Animal.cs、Cow.cs 和 Chicken.cs 文件，单击 Add 按钮。

(4) 在已经添加的文件中修改名称空间声明，如下所示：


```
namespace Ch12Ex04
```

Code snippets Ch12Ex04\Animal.cs、Ch12Ex04\Cow.cs 和 Ch12Ex04\Chicken.cs

(5) 修改 Animal.cs，如下所示：


```
public abstract class Animal
{
   ...

   public abstract void MakeANoise();
}
```

代码段 Ch12Ex04\Animal.cs

(6) 修改 Chicken.cs，如下所示：


```
public class Chicken : Animal
{
   ...

   public override void MakeANoise()
   {
      Console.WriteLine("{0} says 'cluck!'", name);
   }
}
```

代码段 Ch12Ex04\Chicken.cs

(7) 修改 Cow.cs，如下所示：


```
public class Cow : Animal
{
   ...
```

```
    public override void MakeANoise()
    {
      Console.WriteLine("{0} says 'moo!'", name);
    }
  }
```

代码段 Ch12Ex04\Cow.cs

(8) 添加一个新类 SuperCow，并修改 SuperCow.cs 中的代码，如下所示：


```
public class SuperCow : Cow
{
  public void Fly()
  {
    Console.WriteLine("{0} is flying!", name);
  }

  public SuperCow(string newName) : base(newName)
  {
  }

  public override void MakeANoise()
  {
    Console.WriteLine("{0} says 'here I come to save the day!'", name);
  }
}
```

代码段 Ch12Ex04\SuperCow.cs

(9) 添加一个新类 Farm，并修改 Farm.cs 中的代码，如下所示：


```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ch12Ex04
{
  public class Farm<T> : IEnumerable<T>
    where T : Animal
  {
    private List<T> animals = new List<T>();

    public List<T> Animals
    {
      get
      {
        return animals;
      }
    }

    public IEnumerator<T> GetEnumerator()
    {
      return animals.GetEnumerator();
```

```
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
       return animals.GetEnumerator();
    }

    public void MakeNoises()
    {
       foreach (T animal in animals)
       {
          animal.MakeANoise();
       }
    }
    public void FeedTheAnimals()
    {
       foreach (T animal in animals)
       {
          animal.Feed();
       }
    }
    public Farm<Cow> GetCows()
    {
       Farm<Cow> cowFarm = new Farm<Cow>();
       foreach (T animal in animals)
       {
          if (animal is Cow)
          {
             cowFarm.Animals.Add(animal as Cow);
          }
          return cowFarm;
       }
    }
}
```

代码段Ch12Ex04\Farm.cs

(10) 修改 Program.cs，如下所示：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   Farm<Animal> farm = new Farm<Animal>();
   farm.Animals.Add(new Cow("Jack"));
   farm.Animals.Add(new Chicken("Vera"));
   farm.Animals.Add(new Chicken("Sally"));
   farm.Animals.Add(new SuperCow("Kevin"));
   farm.MakeNoises();

   Farm<Cow> dairyFarm = farm.GetCows();
   dairyFarm.FeedTheAnimals();

   foreach (Cow cow in dairyFarm)
   {
```

```
            if (cow is SuperCow)
            {
               (cow as SuperCow).Fly();
            }
         }
         Console.ReadKey();
      }
```

代码段 Ch12Ex04\Program.cs

(11) 执行应用程序，结果如图 12-5 所示。

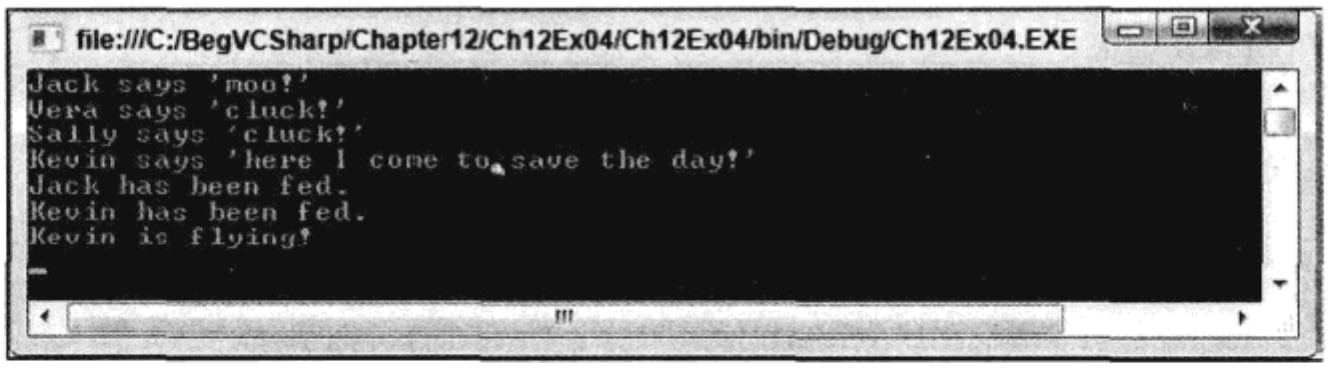


图　12-5

示例的说明

在这个示例中，创建了一个泛型类 Farm<T>，它没有继承泛型 List 类，而是将泛型 list 类作为公共属性公开，该 List 的类型由传送给 Farm<T>的类型参数 T 确定，且被约束为 Animal，或者继承自 Animal。


```
public class Farm<T> : IEnumerable<T>
   where T : Animal
{
   private List<T> animals = new List<T>();

   public List<T> Animals
   {
      get
      {
         return animals;
      }
   }
```

代码段 Ch12Ex04\Farm.cs

Farm<T>还实现了 IEnumerable<T>，其中，T 传递给这个泛型接口，因此也以相同的方式进行了约束。实现这个接口，就可以迭代包含在 Farm<T>中的项，而无需显式迭代 Farm<T>.Animals。很容易就能做到这一点，只需返回Animals 公开的枚举器即可，该枚举器是一个 List<T>类，也实现了 IEnumerable<T>。

```
   public IEnumerator<T> GetEnumerator()
   {
      return animals.GetEnumerator();
   }
```

因为 IEnumerable<T>继承自 IEnumerable，所以还需要实现 IEnumerable.GetEnumerator ()：

```
IEnumerator IEnumerable.GetEnumerator()
{
   return animals.GetEnumerator();
}
```

之后，Farm<T>包含的两个方法利用了抽象类 Animal 的方法：

```
public void MakeNoises()
{
   foreach (T animal in animals)
   {
      animal.MakeANoise();
   }
}

public void FeedTheAnimals()
{
   foreach (T animal in animals)
   {
      animal.Feed();
   }
}
```

T 被约束为 Animal，所以这段代码会正确编译——无论 T 是什么，都可以访问这些方法。

下一个方法 GetCows()更加有趣。这个方法提取了集合中类型为 Cow(或继承自 Cow，例如，新的 SuperCow 类)的所有项：

```
public Farm<Cow> GetCows()
{
   Farm<Cow> cowFarm = new Farm<Cow>();
   foreach (T animal in animals)
   {
      if (animal is Cow)
      {
         cowFarm.Animals.Add(animal as Cow);
      }
   }
   return cowFarm;
}
```

有趣的是，这个方法似乎有点浪费。如果以后希望有同一系列的其他方法，如 GetChickens()，也需要显式实现它们。在使用许多类型的系统中，需要更多的方法。一个较好的解决方案是使用泛型方法，详见本章后面的内容。

Program.cs 中的客户代码测试了Form 的各个方法，它并没有包含前面列出的许多代码，所以不需要深入探讨这些代码。

3. 从泛型类中继承

上面示例中的Farm<T>类以及本章前面介绍的其他几个类都继承自一个泛型类型。在 Farm<T>中，这个类型是一个接口 IEnumerable<T>。这里 Farm<T>在 T 上提供的约束也会在 IEnumerable<T>

中使用的 T 上添加一个额外的约束。这可以用于限制未约束的类型，但是需要遵循一些规则。

首先，如果某个类型所继承的基类型中受到了约束，该类型就不能“解除约束”。也就是说，类型 T 在所继承的基类型中使用时，该类型必须受到至少与基类型相同的约束。例如，下面的代码是正确的：

```
class SuperFarm<T> : Farm<T>
      where T : SuperCow
{
}
```

因为 T 在 Farm<T>中被约束为 Animal，把它约束为 SuperCow，就是把 T 约束为这些值的一个子集，所以这段代码可以正常运行。但是，不会编译以下代码：

```
class SuperFarm<T> : Farm<T>
      where T : struct
{
}
```

可以肯定地说，提供给 SuperFarm<T>的类型 T 不能转换为可由 Farm<T>使用的 T，所以代码不会编译。

甚至对于约束为超集的情况，也会出现相同的问题：

```
class SuperFarm<T> : Farm<T>
      where T : class
{
}
```

即使 SuperFarm<T>允许有像 Animal 这样的类型，Farm<T>中也不允许有满足类约束的其他类型。否则编译就会失败。这个规则适用于本章前面介绍的所有约束类型。

另外，如果继承了一个泛型类型，就必须提供所有必须的类型信息，这可以使用其他泛型类型参数的形式来提供，如上所述，也可以显式提供。这也适用于继承了泛型类型的非泛型类。例如：

```
public class Cards : List<Card>, ICloneable
{
}
```

这是可行的，但下面的代码会失败：

```
public class Cards : List<T>, ICloneable
{
}
```

因为没有提供 T 的信息，所以不能编译。

如果给泛型类型提供了参数，例如，上面的 List<Card>，就可以把类型引用为“关闭”。同样，继承 List<T>，就是继承一个“打开”的泛型类型。

4. 泛型运算符

在C#中，可以像其他方法一样进行运算符的重写，这也可以在泛型类中实现此类重写。例如，可以在Farm<T>中定义如下隐式的转换运算符：

```
public static implicit operator List<Animal>(Farm<T> farm)
{
   List<Animal> result = new List<Animal>();
   foreach (T animal in farm)
   {
      result.Add(animal);
   }
   return result;
}
```

这样，如果需要，就可以在 Farm<T>中把 Animal 对象直接作为 List<Animal>来访问。例如，使用下面的运算符添加两个 Farm<T>实例，这是很方便的：

```
public static Farm<T> operator +(Farm<T> farm1, List<T> farm2)
{
   Farm<T> result = new Farm<T>();

   foreach (T animal in farm1)
   {
      result.Animals.Add(animal);
   }
   foreach (T animal in farm2)
   {
      if (!result.Animals.Contains(animal))
      {
         result.Animals.Add(animal);
      }
   }
   return result;
}

public static Farm<T> operator +(List<T> farm1, Farm<T> farm2)
{
   return farm2 + farm1;
}
```

接着可以添加 Farm<Animal>和 Farm<Cow>的实例，如下所示：

```
Farm<Animal> newFarm = farm + dairyFarm;
```

在这行代码中，dairyFarm(是 Farm<Cow>的实例)隐式转换为 List<Animal>，List<Animal>可以在 Farm<T>中由重载运算符+使用。

读者可能认为，使用下面的代码也可以做到：

```
public static Farm<T> operator +(Farm<T> farm1, Farm<T> farm2)
{
...
}
```

但是，Farm<Cow>不能转换为 Farm<Animal>，所以汇总会失败。为了更进一步，可以使用下面的转换运算符来解决这个问题：

```
public static implicit operator Farm<Animal>(Farm<T> farm)
{
   Farm <Animal> result = new Farm <Animal>();
   foreach (T animal in farm)
   {
      result.Animals.Add(animal);
   }
   return result;
}
```

使用这个运算符，Farm<T>的实例(如 Farm<Cow>)就可以转换为 Farm<Animal>的实例，这解决了上面的问题。所以，可以使用上面列出的两种方法，但是后者更适合，因为它比较简单。

5. 泛型结构

前几章说过，结构实际上与类相同，只有一些微小的区别，而且结构是值类型，不是引用类型。所以，可以用与泛型类相同的方式来创建泛型结构。例如：

```
public struct MyStruct<T1, T2>
{
   public T1 item1;
   public T2 item2;
}
```

12.3.2 定义泛型接口

前面介绍了几个泛型接口，它们都位于 Systems.Collections.Generic 名称空间中，例如，上一个示例中使用的 IEnumerable<T>。定义泛型接口与定义泛型类所用的技术相同，例如：

```
interface MyFarmingInterface<T>
   where T : Animal
{
   bool AttemptToBreed(T animal1, T animal2);

   T OldestInHerd  {get;}
}
```

其中，泛型参数 T 用作 AttemptToBreed()的两个变元的类型和 OldestInHerd 属性的类型。

其继承规则与类相同。如果继承了一个基泛型接口，就必须遵循“保持基接口泛型类型参数的约束”等规则。

12.3.3 定义泛型方法

在上一个示例中提到了方法 GetCows()，在讨论这个示例时也提到，可以使用泛型方法得到这个方法的更一般形式。本节将说明如何达到这一目标。在泛型方法中，返回类型和/或参数类型由泛型类型参数来确定。例如：

```
public T GetDefault<T>()
{
```

```
    return default(T);
}
```

这个小示例使用本章前面介绍的 default 关键字，为类型 T 返回默认值。这个方法的调用如下所示：

```
int myDefaultInt = GetDefault<T>();
```

在调用该方法时提供了类型参数 T。

这个 T 与用于给类提供泛型类型参数的类型差异极大。实际上，可以通过非泛型类来实现泛型方法：

```
public class Defaulter
{
   public T GetDefault<T>()
   {
      return default(T);
   }
}
```

但如果类是泛型的，就必须为泛型方法类型使用不同的标识符。下面的代码不会编译：

```
public class Defaulter<T>
{
   public T GetDefault<T>()
   {
      return default(T);
   }
}
```

必须重命名方法或类使用的类型 T。

泛型方法参数可以采用与类相同的方式使用约束，在此可以使用任意的类类型参数，例如：

```
public class Defaulter<T1>
{
   public T2 GetDefault<T2>()
      where T2 : T1
   {
      return default(T2);
   }
}
```

其中，为方法提供的类型 T2 必须与给类提供的 T1 相同，或者继承自 T1。这是约束泛型方法的常用方式。

在前面的 Farm<T>类中，可以包含下面的方法(在 Ch12Ex04 的下载代码中包含它们，但已注释掉)。

```
public Farm<U> GetSpecies<U>() where U : T
{
   Farm<U> speciesFarm = new Farm<U>();
   foreach (T animal in animals)
   {
      if (animal is U)
```

```
        {
            speciesFarm.Animals.Add(animal as U);
        }
    }
    return speciesFarm;
}
```

这可以替代 GetCows()和相同类型的其他方法。这里使用的泛型类型参数 U 由 T 约束，T 又由 Farm<T>类约束为 Animal。因此，如果愿意，可以把 T 的实例视为 Animal 的实例。

在 Ch12Ex04 的客户代码 Program.cs 中，使用这个新方法需要进行一处修改：

```
Farm<Cow> dairyFarm = farm.GetSpecies<Cow>();
```

也可以编写如下代码：

```
Farm<Chicken> dairyFarm = farm.GetSpecies<Chicken>();
```

或者继承了 Animal 的其他类。

这里要注意，如果某个方法有泛型类型参数，会改变该方法的签名。也就是说，该方法有几个重载，它们仅在泛型类型参数上有区别。例如：

```
public void ProcessT<T>(T op1)
{
    ...
}

public void ProcessT<T, U>(T op1)
{
    ...
}
```

使用哪个方法取决于调用方法时指定的泛型类型参数的个数。

12.3.4 定义泛型委托

最后一个要介绍的泛型类型是泛型委托。本章前面在介绍如何排序和搜索泛型列表时曾介绍过它们，即分别为此使用了 Comparison<T>和 Predicate<T>委托。

第 6 章介绍了如何使用方法的参数和返回类型、delegate 关键字和委托名来定义委托，例如：

```
public delegate int MyDelegate(int op1, int op2);
```

要定义泛型委托，只需声明和使用一个或多个泛型类型参数，例如：

```
public delegate T1 MyDelegate<T1, T2>(T2 op1, T2 op2) where T1 : T2;
```

可以看出，也可以在这里使用约束。第 13 章将更详细地介绍委托，了解在常见的 C#编程技术即“事件”中如何使用它们。

12.4 变体

变体(variance)是协变(covariance)和抗变(contravariance)的统称，这两个概念在.NET 4 中引入。

实际上，它们已经存在不短的时间了(在.NET 2.0 中就可以使用)，但直到.NET 4，仍然很难实现它们，因为它们需要定制的编译过程。

要掌握这些术语的含义，最简单的方式是把它们与多态性进行比较。多态性允许把派生类型的对象放在基类型的变量中，例如：

```
Cow myCow = new Cow("Geronimo");
Animal myAnimal = myCow;
```

其中把 Cow 类型的对象放在 Animal 类型的变量中，这是可行的，因为 Cow 派生自 Animal。但是，这不适用于接口，也就是说，下面的代码不能工作：

```
IMethaneProducer<Cow> cowMethaneProducer = myCow;
IMethaneProducer<Animal> animalMethaneProducer = cowMethaneProducer;
```

假定 Cow 支持 IMethaneProducer<Cow>接口，第一行代码就没有问题。但是，第二行代码预先假定两个接口类型有某种关系，但实际上这种关系不存在，所以无法把一种类型转换为另一种类型。肯定无法使用本章前面介绍的技术，因为泛型类型的所有类型参数都是不变的。但是可以在泛型接口和泛型委托上定义变体类型参数，以适合上述代码演示的情形。

为了使上述代码工作，IMethaneProducer<T>接口的类型参数 T 必须是协变的。有了协变的类型参数，就可以在 IMethaneProducer<Cow>和 IMethaneProducer<Animal>之间建立继承关系，这样一种类型的变量就可以包含另一种类型的值，这与多态性类似(但稍复杂些)。

为了完成对变体的介绍，需要看看变体的另一面：抗变。抗变和协变是类似的，但方向相反。抗变不能像协变那样，把泛型接口值放在使用基类型的变量中，而可以把该接口放在使用派生类型的变量中，例如：

```
IGrassMuncher<Cow> cowGrassMuncher = myCow;
IGrassMuncher<SuperCow> superCowGrassMuncher = cowGrassMuncher;
```

初看起来似乎有点古怪，因为不能通过多态性完成相同的功能。但是这在一些情况下是一项有效的技术，如“抗变”一节所述。

下面两节将介绍如何在泛型类型中实现变体，以及.NET Framework 如何使用变体简化编程。

本节所有代码都包含在演示项目 VarianceDemo 中，可供使用。

12.4.1　协变

要把泛型类型参数定义为协变，可以在类型定义中使用 out 关键字，如下面的示例所示：

```
public interface IMethaneProducer<out T>
{
...
}
```

对于接口定义，协变类型参数只能用作方法的返回值或属性 get 访问器。

说明协变用途的一个很好的例子在.NET Framework 中，即前面使用的 IEnumerable<T>接口。在

这个接口中，项类型 T 定义为协变，这表示可以把支持 IEnumerable<Cow>的对象放在IEnumerable<Animal>类型的变量中。

因此下面的代码是有效的：

```
static void Main(string[] args)
{
    List<Cow> cows = new List<Cow>();
    cows.Add(new Cow("Geronimo"));
    cows.Add(new SuperCow("Tonto"));
    ListAnimals(cows);
    Console.ReadKey();
}

static void ListAnimals(IEnumerable<Animal> animals)
{
    foreach (Animal animal in animals)
    {
        Console.WriteLine(animal.ToString());
    }
}
```

其中 cows 变量的类型是 List<Cow>，它支持 IEnumerable<Cow>接口。通过协变，这个变量可以传送给需要 IEnumerable<Animal>类型的参数的方法。回想一下 foreach 循环的工作方式，就知道 GetEnumerator()方法用于获取 IEnumerator<T>的一个枚举器，该枚举器的 Current 属性用于访问项。IEnumerator<T>还把其类型参数定义为协变，这表示可以把它用作参数的 get 访问器，而且一切都运转良好。

12.4.2 抗变

要把泛型类型参数定义为抗变，可以在类型定义中使用 in 关键字：

```
public interface IGrassMuncher<in T>
{
    ...
}
```

对于接口定义，抗变类型参数只能用作方法参数，不能用作返回类型。

理解这一点的最佳方式是列举一个在.NET Framework 中使用抗变的例子。带有抗变类型参数的一个接口是前面用过的 IComparer<T>。可以给 Animal 实现这个接口，如下所示：

```
public class AnimalNameLengthComparer : IComparer<Animal>
{
    public int Compare(Animal x, Animal y)
    {
        return x.Name.Length.CompareTo(y.Name.Length);
    }
}
```

这个比较器按名称的长度比较动物，所以可以使用它对 List<Animal>的实例排序。通过抗变，还可以使用它对 List<Cow>的实例排序，尽管 List<Cow>.Sort()方法需要 IComparer<Cow>的实例。

```
List<Cow> cows = new List<Cow>();
```

```
cows.Add(new Cow("Geronimo"));
cows.Add(new SuperCow("Tonto"));
cows.Add(new Cow("Gerald"));
cows.Add(new Cow("Phil"));
cows.Sort(new AnimalNameLengthComparer());
```

大多数情况下，抗变都会发生——它在.NET Framework 中可帮助执行这种排序操作。.NET 4 中这两种变体的优点是，您可以在需要时使用本节介绍的技术实现它。

12.5 小结

本章学习了如何在 C#中使用泛型类型；如何创建自己的泛型类型，包括类、接口、方法和委托；如何使用结构，包括创建可空类型，使用 System.Collecitons.Generic 名称空间中的类。

泛型是 C#中一项功能极其强大的新技术，使用它们创建的类可以同时达到多种目的，并可以在许多不同的情况下使用。即使没有必要创建自己的泛型类型，也可以使用泛型集合类。

第 13 章将研究其他基本知识，探讨事件，继续对基本 C#语言的讨论。

12.6 练习

(1) 下面哪些元素可以是泛型?

a. 类

b. 方法

c. 属性

d. 运算符重载

e. 结构

f. 枚举

(2) 扩展 Ch12Ex01 中的 Vector 类，使*运算符返回两个矢量的点积(dot product)。

两个矢量的点积定义为两个矢量的大小与两个矢量之间夹角余弦的乘积。

(3) 下面的代码存在什么错误？请加以修改。

```
public class Instantiator<T>
{
   public T instance;

   public Instantiator()
   {
      instance = new T();
   }
}
```

(4) 下面的代码存在什么错误？请加以修改。

```
public class StringGetter<T>
{
   public string GetString<T>(T item)
   {
      return item.ToString();
   }
}
```

(5) 创建一个泛型类 ShortCollection<T>，它实现了 IList<T>，包含一个项集合及集合的最大容量。这个最大容量应是一个整数，并可以提供给 ShortCollection<T>的构造函数，或者默认为 10。构造函数还应通过 List<T>参数获取项的最初列表。该类与 Collection<T>的功能相同，但如果试图给集合添加太多的项，或者传递给构造函数的 List<T>包含太多的项，就会抛出 IndexOutOfRange-Exception 类型的异常。

(6) 下面的代码可以进行编译吗？试说明原因？

```
public interface IMethaneProducer<out T>
{
   void BelchAt(T target);
}
```

附录 A 给出了练习答案。

12.7 本章要点

主 题	重 要 概 念
使用泛型类型	泛型类型需要一个或多个类型参数才能工作。在声明变量时，传送需要的类型参数，就可以把泛型类型用作变量的类型。为此，应把逗号分隔的类型名列表放在尖括号中
可空类型	可空类型可以使用指定值类型的任意值或 null 值。使用 Nullable<T>或 T?语法，可以声明可空类型的变量
??运算符	空接合运算符返回第一个操作数的值，如果第一个操作数是 null，就返回第二个操作数的值
泛型集合	泛型集合非常有用，因为它们内置了强类型化功能。可以使用 List<T>、Collection<T>和 Dictionary<K, V>等集合类型，它们还提供了泛型接口。为了针对泛型集合进行排序和搜索，应使用 IComparer<T>和 IComparable<T>接口
定义泛型类	泛型类型的定义十分类似于其他类型，但在指定类型名时需要添加泛型类型参数。与使用泛型类型一样，也需要把这些参数指定为逗号分隔的列表，并放在尖括号中。在使用类型名的地方都可以使用泛型类型参数，例如可以在方法的返回值和参数中使用它们
泛型类型的参数约束	为了高效地在泛型类型代码中使用泛型类型参数，可以在使用类型时约束可以提供的类型。可以根据基类、所支持的接口、是否必须是值类型或引用类型以及是否支持无参数的构造函数等，来约束类型参数。如果没有这些约束，就必须使用 default 关键字来实例化泛型类型的变量

(续表)

主　题	重 要 概 念
其他泛型类型	除类之外，还可以定义泛型接口、委托和方法
变体	变体是类似于多态性的一个概念，但应用于类型参数。它允许使用一个泛型类型替代另一个泛型类型，这些泛型类型仅在所使用的泛型类型参数上有区别。协变允许在两种类型之间转换，其中目标类型有一个类型参数，它是源类型的类型参数的基类。抗变允许进行相反的转换。协变类型参数用 out 参数定义，只能用作返回类型和属性 get 访问器的类型。抗变类型参数用 in 参数定义，只能用作方法的参数

第13章

其他 OOP 技术

本章内容：

- ::运算符
- 全局名称空间限定符
- 如何创建定制异常
- 如何使用事件
- 如何使用匿名方法

本章将介绍前面未涉及的内容，继续对 C#语言的讨论。并不是说这些技术没用，它们只是不适合放在前面的主题中讨论而已。

本章还将对前面几章构建的 CardLib 代码进行最后的修改，并使用 CardLib 来创建扑克牌游戏。

13.1 ::运算符和全局名称空间限定符

::运算符提供了另一种访问名称空间中类型的方式。如果要使用一个名称空间的别名，但该别名与实际名称空间层次结构之间的界限不清晰，这将是必要的。在那种情况下，名称空间层次结构优先于名称空间别名。为了阐明其含义，考虑下列代码：

```
using MyNamespaceAlias = MyRootNamespace.MyNestedNamespace;

namespace MyRootNamespace
{
   namespace MyNamespaceAlias
   {
      public class MyClass
      {
      }
   }

   namespace MyNestedNamespace
```

```
    {
        public class MyClass
        {
        }
    }
}
```

MyRootNamespace 中的代码使用下面的代码引用一个类：

```
MyNamespaceAlias.MyClass
```

这行代码表示的类是 MyRootNamespace.MyNamespaceAlias.MyClass 类，而不是 MyRootNamespace.MyNestedNamespace.MyClass 类。也就是说，MyRootNamespace.MyNamespaceAlias 名称空间隐藏了由 using 语句定义的别名，该别名指向 MyRootNamespace. MyNestedNamespace 名称空间。仍然可以访问这个名称空间以及其中包含的类，但需要使用不同的语法：

```
MyNestedNamespace.MyClass
```

另外，还可以使用::运算符：

```
MyNamespaceAlias::MyClass
```

使用这个运算符会迫使编译器使用由 using 语句定义的别名，因此代码指向 MyRootNamespace.MyNestedNamespace.MyClass。

::运算符还可以和 global 关键字一起使用，它实际上是顶级根名称空间的别名。这有助于更清晰地说明要指向哪个名称空间，如下所示：

```
global::System.Collections.Generic.List<int>
```

这是希望使用的类，即 List<T>泛型集合类。它肯定不是用下列代码定义的类：

```
namespace MyRootNamespace
{
   namespace System
   {
      namespace Collections
      {
         namespace Generic
         {
            class List<T>
            {
            }
         }
      }
   }
}
```

当然，应避免使名称空间的名称与已有的.NET 名称空间相同，但这个问题只在大型项目中才会出现，尤其是作为大型开发队伍中的一员进行开发时，此类问题就更严重。使用::运算符和 global 关键字可能是访问所需类型的唯一方式。

13.2　定制异常

第 7 章讨论了异常，以及如何使用 try...catch...finally 块处理它们。我们还论述了几个标准的.NET 异常，包括异常的基类 System.Exception。在应用程序中，有时也可以从这个基类中派生自己的异常类，并使用它们，而不是使用标准的异常。这样就可以把更具体的信息发送给捕获该异常的代码，让处理异常的捕获代码更有针对性。例如，可以给异常类添加一个新属性，以便访问某些底层信息，这样异常的接收代码就可以做出必要的改变，或者仅给出异常起因的更多信息。

定义了异常类后，就可以使用 Debug | Exceptions 对话框中的 Add 按钮，把它添加到 VS 可以识别的异常列表中，然后定义与异常相关的操作，如第 7 章所述。

在 System 名称空间中有两个基本的异常类 ApplicationException 和 SystemException，它们派生于 Exception。SystemException 用作.NET Framework 预定义的异常的基类，ApplicationException 由开发人员用于派生自己的异常类。但最近的最佳实践方式是不从这个类中派生异常，而应使用 Exception。ApplicationException 类在未来可能会被废弃。

给 CardLib 添加定制异常

定制异常的用法最好通过升级 CardLib 项目来说明。如果试图访问索引小于 0 或大于 51 的扑克牌，Deck.GetCard()方法目前就会抛出一个标准的.NET 异常，但下面改为使用一个定制异常。

首先，需要在 BegVCSharp\Chapter13 目录中创建一个新的类库项目 Ch13CardLib，像以前一样把类从 Ch12CardLib 中复制过来，并把名称空间改为 Ch13CardLib。接着定义该异常。方法是使用在新类文件 CardOutOfRange Exception.cs 中定义的一个新类，这个新类是使用 Project | Add Class 添加到 Ch13CardLib 项目中的：

可从 wrox.com 下载源代码

```
public class CardOutOfRangeException : Exception
{
   private Cards deckContents;

   public Cards DeckContents
   {
      get
      {
         return deckContents;
      }
   }

   public CardOutOfRangeException(Cards sourceDeckContents) :
       base("There are only 52 cards in the deck.")
   {
     deckContents = sourceDeckContents;
   }
}
```

代码段 Ch13CardLib\CardOutOfRangeException.cs

这个类的构造函数需要使用 Cards 类的一个实例，它允许通过 DeckContents 属性来访问这个 Cards 对象，为 Exception 基构造函数提供合适的错误信息，使该错误信息可以通过类的 Message 属性得到。

接着，在 Deck.cs 中添加抛出该异常的代码(替换原来的标准异常)：

可从 wrox.com 下载源代码

```
public Card GetCard(int cardNum)
{
   if (cardNum >= 0 && cardNum <= 51)
      return cards[cardNum];
   else
      throw new CardOutOfRangeException(cards.Clone() as Cards);
}
```

代码段 Ch13CardLib\Deck.cs

DeckContents 属性是通过对 Deck 对象的当前内容(其形式是一个 Cards 对象)进行深度复制来初始化的。这表示，此时的内容是异常抛出时的内容，所以以后对 Deck 内容的修改不会丢失这些信息。

要进行测试，使用下面的客户代码(在本章的下载代码的 Ch13CardClient 中)：

可从 wrox.com 下载源代码

```
Deck deck1 = new Deck();
try
{
   Card myCard = deck1.GetCard(60);
}
catch (CardOutOfRangeException e)
{
   Console.WriteLine(e.Message);
   Console.WriteLine(e.DeckContents[0]);
}
Console.ReadKey();
```

代码段 Ch13CardClient\Program.cs

结果如图 13-1 所示。

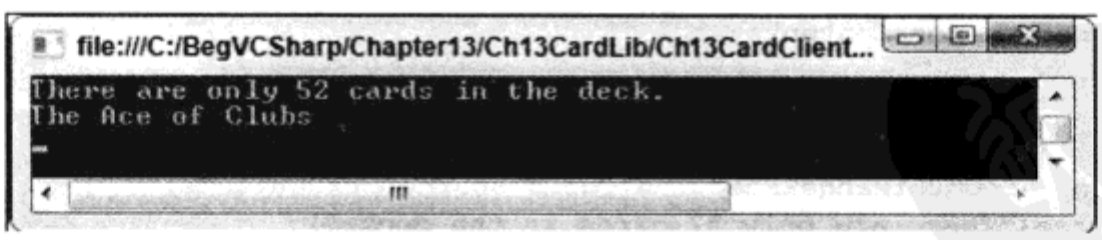


图　13-1

其中捕获代码把异常的 Message 属性写到屏幕上。我们还通过 DeckContents 显示了 Cards 对象中的第一张牌，以证明可以通过定制的异常对象访问 Cards 集合。

13.3　事件

本节主要讨论.NET 中最常用的 OOP 技术：事件。像往常一样，先介绍基础知识，分析事件到底是什么。之后讨论几个简单的事件，看看使用它们可以做什么。然后论述如何创建和使用自己的事件。

本章的最后介绍如何给 CardLib 类库添加一个事件，使该类库更完整。另外，因为这是在介绍一些更高级论题之前的最后一部分，我们还将创建一个使用该类库的有趣的扑克牌游戏应用程序。

13.3.1　事件的含义

事件类似于异常，因为它们都由对象引发(抛出)，我们可以提供代码来处理事件。但它们也有几个重要的区别。最重要的区别是并没有与 try...catch 类似的结构来处理事件，而必须订阅(subscribe)它们。订阅一个事件的含义是提供代码，在事件发生时执行这些代码，它们称为事件处理程序。

单个事件可供多个处理程序订阅，在该事件发生时，这些处理程序都会被调用，其中包括引发该事件的对象所在的类中的事件处理程序，但事件处理程序也可能在其他类中。

事件处理程序本身都是简单的方法。对事件处理方法的唯一限制是它必须匹配于事件所要求的返回类型和参数。这个限制是事件定义的一部分，由一个委托指定。

在事件中使用委托是非常有用的。第 6 章对此已进行了论述，读者可以温习这一部分，复习一下委托是什么以及如何使用它们。

基本处理过程如下所示：首先，应用程序创建一个可以引发事件的对象。例如，假定一个即时消息传送(instant messaging)应用程序创建的对象表示一个远程用户的连接。当接收到通过该连接从远程用户传送来的信息时，这个连接对象会引发一个事件，如图 13-2 所示。

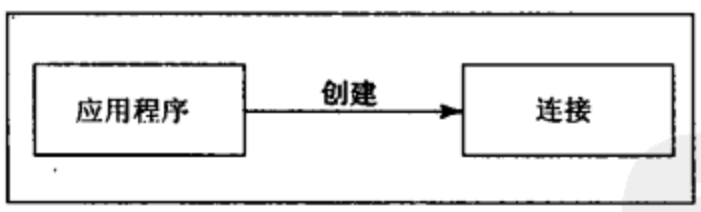


图　13-2

接着，应用程序订阅事件。为此，即时消息传送应用程序将定义一个方法，该方法可以与事件指定的委托类型一起使用，把这个方法的一个引用传送给事件，而事件的处理方法可以是另一个对象的方法，假定是表示显示设备的对象，当接收到信息时，该方法将显示即时消息，如图 13-3 所示。

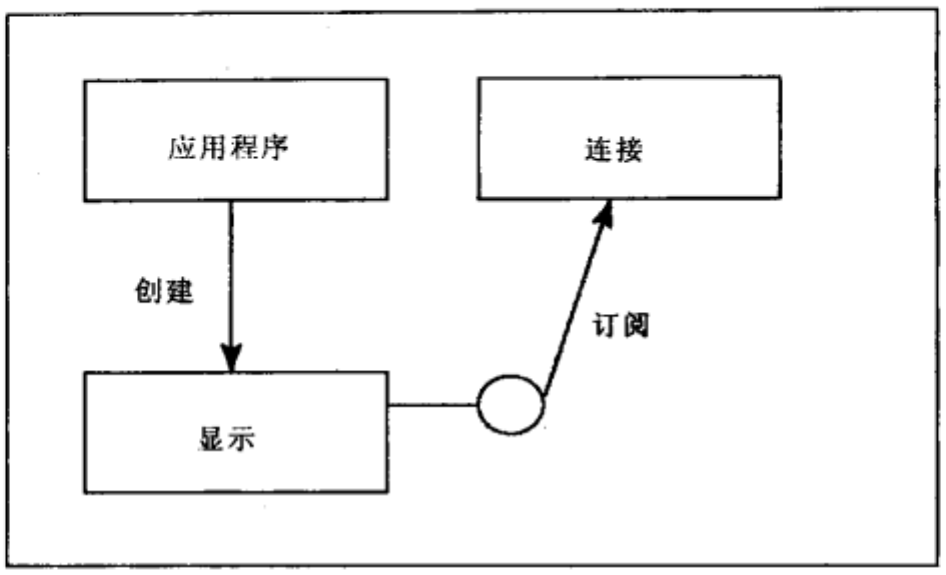


图　13-3

引发事件后，就通知订阅器。当接收到通过连接对象传来的即时消息时，就调用显示设备对象上的事件处理方法。因为我们使用的是一个标准方法，所以引发事件的对象可以通过参数传送任何相关的信息，这样就大大增加了事件的通用性。在本例中，一个参数是即时消息的文本，事件处理程序可以在显示设备对象上显示它，如图 13-4 所示。

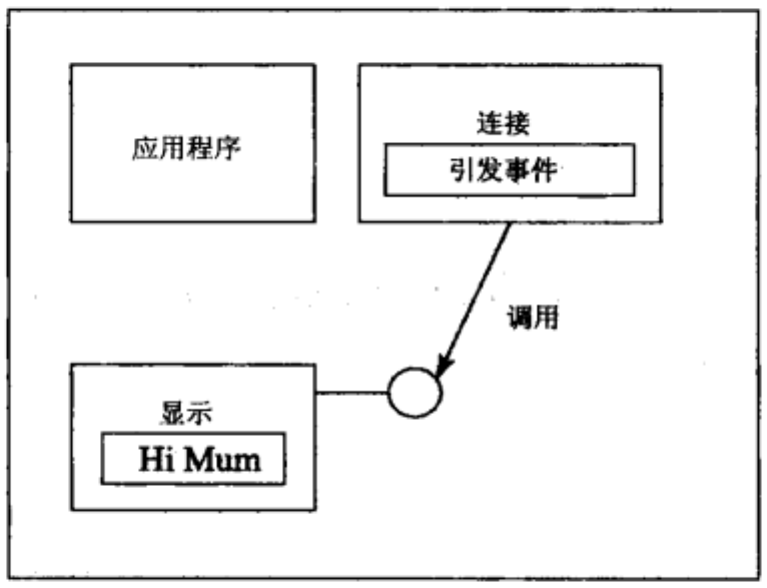


图　13-4

13.3.2　处理事件

如前所述，要处理事件，需要提供一个事件处理方法来订阅事件，该方法的返回类型和参数应该匹配事件指定的委托。下面的示例使用一个简单的计时器对象引发事件，调用一个处理方法。

试一试：处理事件

(1) 在 C:\BegVCSharp\Chapter13 目录中下创建一个新的控制台应用程序 Ch13Ex01。

(2) 修改 Program.cs 中的代码，如下所示：


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Timers;

namespace Ch13Ex01
{
   class Program
   {
      static int counter = 0;

      static string displayString =
                        "This string will appear one letter at a time. ";
      static void Main(string[] args)
      {
         Timer myTimer = new Timer(100);
         myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
         myTimer.Start();
         Console.ReadKey();
      }

      static void WriteChar(object source, ElapsedEventArgs e)
      {
         Console.Write(displayString[counter++ % displayString.Length]);
      }
   }
}
```

代码段 Ch13Ex01\Program.cs

(3) 运行应用程序(启动后，按回车键将终止程序的执行)，在经过短暂运行后，将显示如图 13-5 所示的结果。

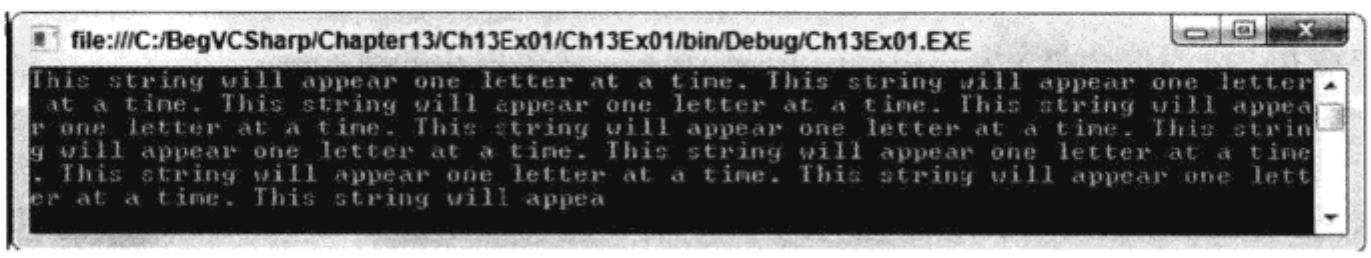


图　13-5

示例的说明

用于引发事件的对象是 System.Timers.Timer 类的一个实例。使用一个时间段(以毫秒为单位)来初始化该对象。当使用 Start()方法启动 Timer 对象时，就引发一系列事件，根据指定的时间段来引发事件。Main()用 100 毫秒初始化 Timer 对象，所以在启动该对象后，1 秒钟内将引发 10 次事件：

```
static void Main(string[] args)
{
   Timer myTimer = new Timer(100);
```

Timer 对象有一个 Elapsed 事件，这个事件要求事件处理程序必须匹配 System.Timers.ElapsedEventHandler 委托类型的返回类型和参数，该委托是.NET Framework 中定义的标准委托之一，指定了返回类型和参数：

```
void functionName(object source, ElapsedEventArgs e);
```

Timer 对象的第一个参数是它本身的引用，第二个参数则是 ElapsedEventArgs 对象的一个实例。现在可以不考虑这些参数，后面将论述它们。

在代码中，有一个匹配该返回类型和参数的方法：

```
static void WriteChar(object source, ElapsedEventArgs e)
{
  Console.Write(displayString[counter++ % displayString.Length]);
}
```

这个方法使用 Class1 的两个静态字段 counter 和 displayString 来显示一个字符。每次调用方法时，显示的字符都不相同。

下一个任务是把这个处理程序与事件关联起来——即订阅它。为此，可以使用+=运算符，给事件添加一个处理程序，其形式是使用事件处理方法初始化的一个新委托实例：

```
static void Main(string[] args)
{
  Timer myTimer = new Timer(100);
  myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

这个命令(使用有点古怪的语法，专用于委托)在列表中添加一个处理程序，当引发 Elapsed 事件时，就会调用该处理程序。可以给这个列表添加任意多个处理程序，只要它们满足指定的条件即可。当引发事件时，会依次调用每个处理程序。

Main()剩下的任务是启动计时器：

```
  myTimer.Start();
```

我们不想在处理完任何事件前终止应用程序，所以要让 Main()函数一直执行。最简单的方式是请求用户输入，因为这个命令要在用户按下回车键后，才会停止处理。

```
  Console.ReadKey();
```

在这里，Main()中的处理会停止，但 Timer 对象中的处理将继续。当该对象引发事件时，就调用 WriteChar()方法，同时该方法运行 Console.ReadLine()语句。

注意，可以使用上一章介绍的方法组概念简化添加事件处理程序的语法：

```
myTimer.Elapsed += WriteChar;
```

最终结果是相同的，但不必显式指定委托类型，编译器会根据使用事件的上下文来指定它。但是，许多程序员不喜欢这个语法，因为它降低了可读性——不再能一眼看出使用了什么委托类型。如果喜欢，就可以使用这个语法。但为了清晰起见，本章使用的所有委托都显式指定。

13.3.3 定义事件

接着论述如何定义和使用自己的事件。我们将使用本节前言介绍的即时消息传送应用程序示例，并创建一个 Connection 对象，该对象引发由 Display 对象处理的事件。

试一试：定义事件

(1) 在 C:\BegVCSharp\Chapter13 目录中创建一个新控制台应用程序 Ch13Ex02。

(2) 添加一个新类 Connection，并修改 Connection.cs，如下所示：

可从 wrox.com 下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Timers;

namespace Ch13Ex02
{
   public delegate void MessageHandler(string messageText);

   public class Connection
   {
      public event MessageHandler MessageArrived;
      private Timer pollTimer;

      public Connection()
      {
         pollTimer = new Timer(100);
         pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
      }

      public void Connect()
      {
         pollTimer.Start();
      }

      public void Disconnect()
      {
         pollTimer.Stop();
      }

      private static Random random = new Random();

      private void CheckForMessage(object source, ElapsedEventArgs e)
      {
         Console.WriteLine("Checking for new messages.");
         if ((random.Next(9) == 0) && (MessageArrived != null))
         {
            MessageArrived("Hello Mum!");
         }
      }
   }
}
```

代码段 Ch13Ex02\Connection.cs

(3) 添加一个新类 Display，并修改 Display.cs，如下：

可从 wrox.com 下载源代码

```
namespace Ch13Ex02
{
   public class Display
   {
      public void DisplayMessage(string message)
```

```
        {
            Console.WriteLine("Message arrived: {0}", message);
        }
    }
}
```

代码段 Ch13Ex02\Display.cs

(4) 修改 Program.cs 中的代码，如下所示：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
    Connection myConnection = new Connection();
    Display myDisplay = new Display();
    myConnection.MessageArrived +=
          new MessageHandler (myDisplay.DisplayMessage);
    myConnection.Connect();
    Console.ReadKey();
}
```

代码段 Ch13Ex02\Program.cs

(5) 运行应用程序，其结果如图 13-6 所示。

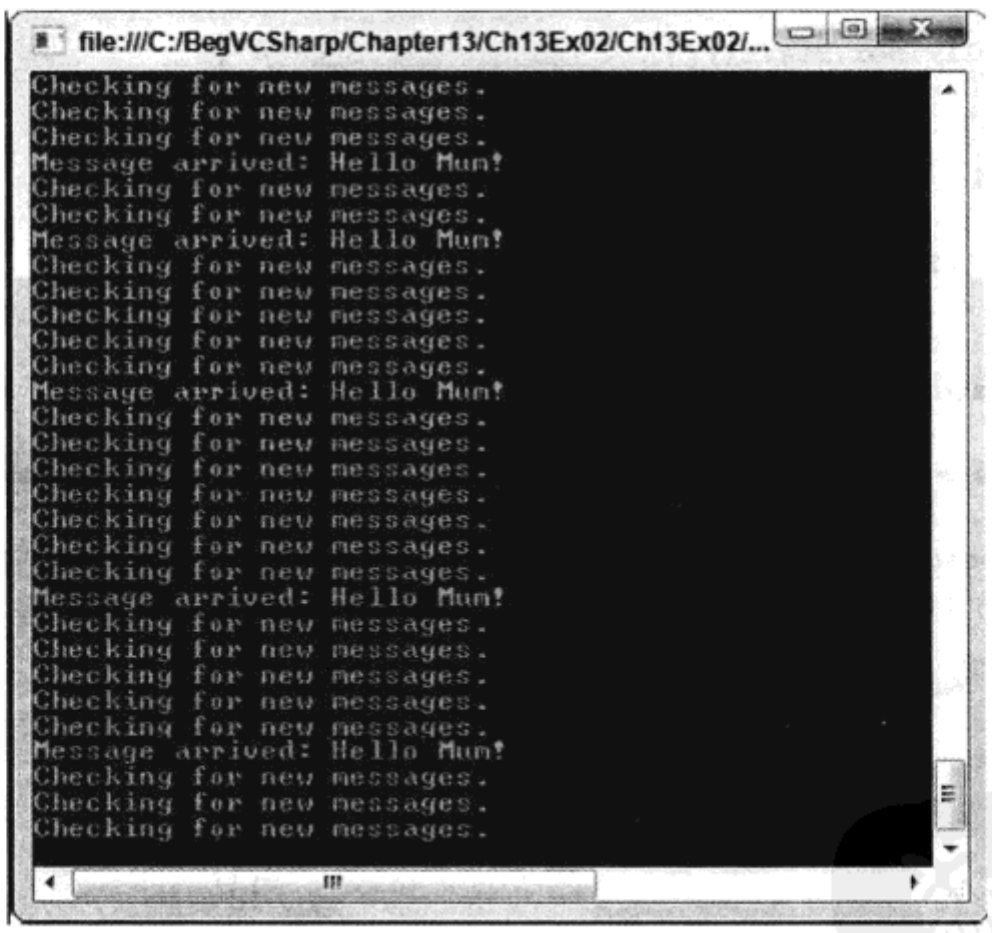


图　13-6

示例的说明

在这个应用程序中，大部分工作是由 Connection 类完成的。这个类的实例使用如本章第一个示例中所示的 Timer 对象，在类的构造函数中初始化它，并通过 Connect()和 Disconnect()访问它的状态(可访问和禁止访问)：

```
public class Connection
   {
      private Timer pollTimer;

      public Connection()
      {
         pollTimer = new Timer(100);
         pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
      }

      public void Connect()
      {
         pollTimer.Start();
      }

      public void Disconnect()
      {
         pollTimer.Stop();
      }

      ...
   }
```

在构造函数中，我们还以与第一个示例相同的方式注册了Elapsed 事件的一个事件处理程序。每当调用这个处理程序方法 CheckForMessage()的次数达到 10 次后，就会引发一个事件。在分析它的代码前，先看看事件的定义。

在定义事件前，必须先定义一个委托类型，以用于该事件，这个委托类型指定了事件处理方法必须拥有的返回类型和参数。为此，我们使用标准的委托语法，在 Ch13Ex02 名称空间中将该委托定义为公共类型，使该类型可供外部代码使用：

```
namespace Ch13Ex02
{
   public delegate void MessageHandler(string messageText);
```

这个委托类型称为 MessageHandler，是 void 函数的签名，它有一个 string 参数。使用这个参数可以把 Connection 对象收到的即时消息发送给 Display 对象。定义了委托 (或者找到合适的现有委托)后，就可以把事件本身定义为 Connection 类的一个成员：

```
public class Connection
{
   public event MessageHandler MessageArrived;
```

给事件命名(这里使用名称 MessageArrived)，在声明时，使用 event 关键字，并指定要使用的委托类型(前面定义的 MessageHandler 委托类型)。以这种方式声明了事件后，就可以引发它，方法是按名称来调用它，就好像它是一个其返回类型和参数是由委托指定的方法一样。例如，使用下面的代码引发这个事件：

```
MessageArrived("This is a message.");
```

如果定义该委托时不包含任何参数，就可以使用下面的代码：

```
MessageArrived();
```

如果定义了较多参数，就需要用比较多的代码来引发事件。CheckForMessage()方法如下所示：

```
private static Random random = new Random();
private void CheckForMessage(object source, ElapsedEventArgs e)
{
    Console.WriteLine("Checking for new messages.");
    if ((random.Next(9) == 0) && (MessageArrived != null))
    {
        MessageArrived("Hello Mum!");
    }
}
```

使用前面几章中的 Random 类实例，生成一个 0～9 之间的随机数，如果该随机数为 0，就引发一个事件，它的发生几率为 10%。这类似于轮流检测连接，看看是否接收到消息，不可能每次检测时，都没有接收到消息。为了把计时器与 Connection 的实例分隔开，使用了 Random 类的一个私有静态实例。

注意，这里还提供了其他逻辑。只有表达式 MessageArrived != null 为 true，才引发一个事件。这个表达式也使用了委托语法，但语法略有不同，其含义是“事件是否有订阅者？”。如果没有订阅者，MessageArrived 就是 null，也就不会引发事件。

订阅事件的类是 Display，它包含一个方法 DisplayMessage()，其定义如下所示：

```
public class Display
{
  public void DisplayMessage(string message)
  {
    Console.WriteLine("Message arrived: {0}", message);
  }
}
```

这个方法匹配于委托类型(而且是公共的，如果类不是生成该事件的类，则其事件处理程序就必须是公共的)，所以可以用它来响应 MessageArrived 事件。

剩下的是 Main()中的代码初始化了Connection 和Display 类的实例，把它们关联起来，开始执行任务。这里需要的代码类似于第一个示例中的代码：

```
static void Main(string[] args)
{
  Connection myConnection = new Connection();
  Display myDisplay = new Display();
  myConnection.MessageArrived +=
        new MessageHandler(myDisplay.DisplayMessage);
  myConnection.Connect();
  Console.ReadKey();
}
```

再次调用 Console.ReadKey()，当开始执行 Connection 对象的 Connect()方法时，暂停 Main()的处理。

1. 多用途的事件处理程序

前面 Timer.Elapsed 事件的委托包含了事件处理程序中常见的两类参数，如下所示：

- object source——引发事件的对象的引用
- ElapsedEventArgse——由事件传送的参数

在这个事件(以及许多其他的事件)中使用 Object 类型参数的原因是，我们常常要为由不同对象引发的几个相同事件使用同一个事件处理程序，但仍要指定哪个对象生成了事件。

要说明这一点，下面将扩展上一个示例。

试一试：使用多用途的事件处理程序

(1) 在 C:\BegVCSharp\Chapter13 目录中创建一个新控制台应用程序 Ch13Ex03。

(2) 复制 Ch13Ex02 中 Program.cs、Connection.cs 和 Display.cs 的代码，并把每个文件中的 Ch13Ex02 名称空间改成 Ch13Ex03。

(3) 添加一个新类 MessageArrivedEventArgs，修改 MessageArrivedEventArgs.cs，如下所示：


可从 wrox.com 下载源代码

```
namespace Ch13Ex03
{
   public class MessageArrivedEventArgs : EventArgs
   {
      private string message;

      public string Message
      {
         get
         {
            return message;
         }
      }

      public MessageArrivedEventArgs()
      {
         message = "No message sent.";
      }

      public MessageArrivedEventArgs(string newMessage)
      {
         message = newMessage;
      }
   }
}
```

代码段 Ch13Ex03\MessageArrivedEventArgs.cs

(4) 修改 Connection.cs，如下所示：


可从 wrox.com 下载源代码

```
namespace Ch13Ex03
{
   public delegate void MessageHandler(Connection source,
                                       MessageArrivedEventArgs e);
```

```
public class Connection
{
   public event MessageHandler MessageArrived;

   private string name { get; set; }
   ...

   private void CheckForMessage(object source, EventArgs e)
   {
      Console.WriteLine("Checking for new messages.");
      if ((random.Next(9) == 0) && (MessageArrived != null))
      {
         MessageArrived(this, new MessageArrivedEventArgs("Hello Mum!"));
      }
   }

   ...

  }
}
```

代码段 Ch13Ex03\Connection.cs

(5) 修改 Display.cs(包括事件参数类型)，如下所示：

可从 wrox.com 下载源代码

```
public void DisplayMessage(Connection source, MessageArrivedEventArgs e)
{
   Console.WriteLine("Message arrived from: {0}", source.Name);
   Console.WriteLine("Message Text: {0}", e.Message);
}
```

代码段 Ch13Ex03\Display.cs

(6) 修改 Program.cs，如下所示：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
   Connection myConnection1 = new Connection();
   myConnection1.Name = "First connection.";
   Connection myConnection2 = new Connection();
   myConnection2.Name = "Second connection.";
   Display myDisplay = new Display();
   myConnection1.MessageArrived +=
               new MessageHandler(myDisplay.DisplayMessage);
   myConnection2.MessageArrived +=
               new MessageHandler(myDisplay.DisplayMessage);
   myConnection1.Connect();
   myConnection2.Connect();
   Console.ReadKey();
}
```

代码段 Ch13Ex03\Program.cs

(7) 运行应用程序，其结果如图 13-7 所示。

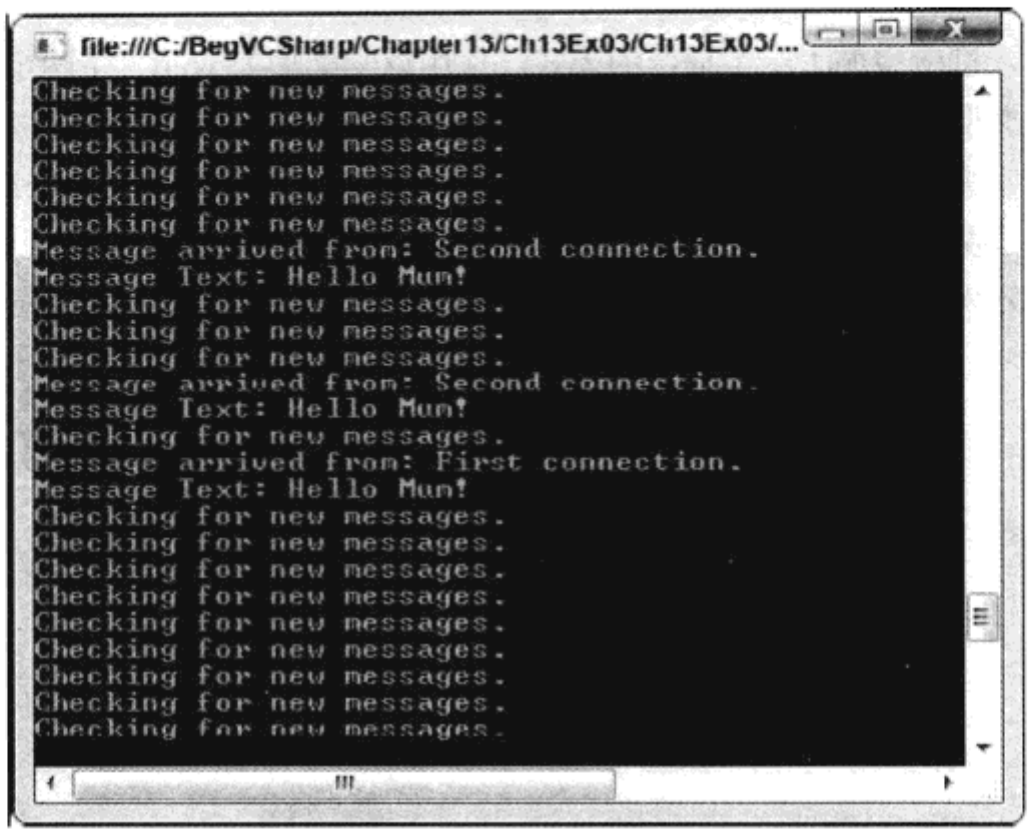


图　13-7

示例的说明

发送一个引发事件的对象引用，将其作为事件处理程序的一个参数，就可以为不同的对象定制处理程序的响应。利用该引用可以访问源对象，包括它的属性。

通过发送包含在派生于 System.EventArgs(与 ElapsedEventArgs 相同)的类中的参数，就可以将其他必要信息提供为参数(例如，MessageArrivedEventArgs 类上的 Message 参数)。

另外，这些参数也将得益于多态性。可以为 MessageArrived 事件定义一个处理程序，如下所示：

```
public void DisplayMessage(object source, EventArgs e)
{
  Console.WriteLine("Message arrived from: {0}",
                    ((Connection)source).Name);
  Console.WriteLine("Message Text: {0}",
                     ((MessageArrivedEventArgs)e).Message);
}
```

修改 Connection.cs 中的委托定义，如下所示：

```
public delegate void MessageHandler(object source, EventArgs e);
```

这个应用程序将像以前那样执行，但 DisplayMessage()方法变得更加通用(至少从理论上讲是这样的——需要使用更多实现代码，才能满足生产环境的质量要求)。这个处理程序还可以处理其他事件，例如 Timer.Elapsed 事件，但必须修改处理程序的内部代码，这样，在引发这个事件时，发送过来的参数才会得到正确处理(以这种方式把它们转换为 Connection 和 MessageArrivedEventArgs 对象，会抛出一个异常，所以这里应使用 as 运算符，检查 null 值)。

2. EventHandler 和泛型 EventHandler<T>类型

大多数情况下，都应遵循上一节提出的模式，使用返回类型为 void、带两个参数的事件处理程序。第一个参数的类型是 object，是事件源。第二个参数的类型派生于 System.EventArgs，包含任意事件变元。这非常常见，所以.NET 提供了两个委托类型 EventHandler 和 EventHandler<T>，以便于定义事件。它们都是委托，使用标准的事件处理模式。泛型版本允许指定要使用的事件变元的类型。

所以在前面的示例中，可以不定义自己的 MessageHandler 泛型类型，而是定义 MessageArrived 事件，如下所示：

```
public class Connection
{
   public event EventHandler MessageArrived;

   ...
}
```

甚或：

```
public class Connection
{
   public event EventHandler<MessageArrivedEventArgs> MessageArrived;

   ...
}
```

这显然是件好事，因为它简化了代码。

3. 返回值和事件处理程序

前面的所有事件处理程序都使用 void 类型的返回值。可以为事件提供返回类型，但这会出问题。这是因为引发给定的事件，可能会调用好几个事件处理程序。如果这些处理程序都返回一个值，那么我们该使用哪个返回值？

系统处理这个问题的方式是，只允许访问由事件处理程序最后返回的那个值，也就是最后一个订阅该事件的处理程序返回的值。这个功能在某些情况下是有用的，但最好使用 void 类型的事件处理程序，且避免使用 out 类型的参数(如果使用 out 参数，参数返回的值的源头就是不清楚的)。

4. 匿名方法

除了定义事件处理方法之外，还可以使用匿名方法(anonymous method)。匿名方法实际上并非传统意义上的方法，它不是某个类上的方法，而纯粹是为用作委托目的而创建的。

要创建匿名方法，需要使用下面的代码：

```
delegate(parameters)
{
   // Anonymous method code.
};
```

其中 parameters 是一个参数列表，这些参数匹配正在实例化的委托类型，由匿名方法的代码使用，例如：

```
delegate(Connection source, MessageArrivedEventArgs e)
{
    // Anonymous method code matching MessageHandler event in Ch13Ex03.
};
```

使用这段代码可以完全绕过 Ch13Ex03 中的 DisplayMessage()方法：

```
myConnection1.MessageArrived +=
  delegate(Connection source, MessageArrivedEventArgs e)
  {
     Console.WriteLine("Message arrived from: {0}", source.Name);
     Console.WriteLine("Message Text: {0}", e.Message);
  };
```

对于匿名方法要注意，对于包含它们的代码块来说，它们是局部的，可以访问这个区域内的局部变量。如果使用这样一个变量，它就成为外部变量(outer variable)。外部变量在超出作用域时，是不会删除的，这与其他局部变量不同，在使用它们的匿名方法被销毁时，外部变量才会删除。这比我们希望的时间晚一些，所以要格外小心。如果外部变量占用了大量内存，或者使用的资源在其他方面是比较昂贵的(例如资源在数量上是有限的)，就可能导致内存或性能问题。

13.4　扩展和使用 CardLib

前面介绍了事件的定义和使用，现在就可以在 Ch13CardLib 中使用它们了。当使用 GetCard 获得 Deck 对象中的最后一个 Card 对象时，就将引发的事件 LastCardDrawn 添加到该类库中。这个事件允许订阅者(subscriber)自动重新洗牌，停止客户要求的处理。为这个事件定义的委托(LastCardDrawnHandler)需要为 Deck 对象提供一个引用，这样无论处理程序在什么地方，都可以访问 Shuffle()方法。在 Deck.cs 中添加以下代码：

可从 wrox.com 下载源代码

```
namespace Ch13CardLib
{
   public delegate void LastCardDrawnHandler(Deck currentDeck);
```

代码段 Ch13CardLib\Deck.cs

定义和引发事件的代码比较简单，如下所示：

```
public event LastCardDrawnHandler LastCardDrawn;

...

public Card GetCard(int cardNum)
{
   if (cardNum >= 0 && cardNum <= 51)
   {
      if ((cardNum == 51) && (LastCardDrawn != null))
         LastCardDrawn(this);
      return cards[cardNum];
   }
   else
```

```
            throw new CardOutOfRangeException((Cards)cards.Clone());
        }
```

这是把事件添加到 Deck 类定义所需要的所有代码。

CardLib 的扑克牌游戏客户程序

在开发了 CardLib 库后，就可以使用它了。在结束讲述 C#和.NET Framework 中 OOP 技术的这个部分前，我们将编写扑克牌应用程序的基本代码，其中将使用我们熟悉的扑克牌类。

与前面的章节一样，我们将在 Ch13CardLib 解决方案中添加一个客户控制台应用程序，添加一个 Ch13CardLib 项目的引用，使之成为启动项目。这个应用程序叫作 Ch13CardClient。

首先在 Ch13CardClient 的一个新文件 Player.cs 中创建一个新类 Player，这个类包含两个自动属性：Name(字符串) 和 PlayHand(Cards 类型)。这些属性有私有的 get 访问器。但是 PlayHand 属性仍可以对其内容进行写入访问，这样就可以修改玩家手中的扑克牌。

我们还把默认的构造函数设置为私有，以隐藏它，并提供了一个公共的非默认构造函数，该函数接受 Player 实例中属性 Name 的初始值。

最后，提供一个 bool 类型的方法 HasWon()。如果玩家手中的扑克牌花色都相同(一个简单的取胜条件，但并没有什么意义)，该方法就返回 true。

Player.cs 的代码如下所示：


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch13CardLib;

namespace Ch13CardClient
{
    public class Player
    {
        public string Name { get; private set; }

        public Cards PlayHand { get; private set; }

        private Player()
        {
        }

        public Player(string name)
        {
            Name = name;
            PlayHand = new Cards();
        }

        public bool HasWon()
        {
            bool won = true;
            Suit match = PlayHand[0].suit;
            for (int i = 1; i < PlayHand.Count; i++)
            {
                won &= PlayHand[i].suit == match;
```

```
            }
            return won;
        }
    }
}
```

代码段 Ch13CardClient\Player.cs

接着定义一个处理扑克牌游戏的类 Game，这个类在 Ch13CardClient 项目的 Game.cs 文件中。这个类有 4 个私有成员字段：

- playDeck——Deck 类型的变量，包含要使用的一副扑克牌
- currentCard——一个 int 值，用作下一张要翻开的扑克牌的指针
- players——一个 Player 对象数组，表示玩家
- discardedCards——Cards 集合，表示玩家扔掉的扑克牌，但还没有放回整副牌中。

这个类的默认构造函数初始化了存储在 playDeck 中的 Deck，并洗牌，把 currentCard 指针变量设置为 0(playDeck 中的第一张牌)，并关联了 playDeck.LastCardDrawn 事件的处理程序 Reshuffle()。这个处理程序将洗牌，初始化 discardedCards 集合，并把 currentCard 重置为 0，准备从新的一副牌中读取扑克牌。

Game 类还包含两个实用方法：SetPlayers()可以设置游戏的玩家(Player 对象数组)，DealHands()可以处理玩家手中的牌(每个玩家有 7 张牌)。玩家的数量限制为 2～7 人，确保每个玩家有足够多的牌。

最后，PlayGame()方法包含游戏逻辑。在分析了 Program.cs 中的代码后介绍这个方法，Game.cs 的剩余代码如下所示：


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ch13CardLib;

namespace Ch13CardClient
{
   public class Game
   {
      private int currentCard;
      private Deck playDeck;
      private Player[] players;
      private Cards discardedCards;

      public Game()
      {
         currentCard = 0;
         playDeck = new Deck(true);
         playDeck.LastCardDrawn += new LastCardDrawnHandler(Reshuffle);
         playDeck.Shuffle();
         discardedCards = new Cards();
      }

      private void Reshuffle(Deck currentDeck)
```

```
{
   Console.WriteLine("Discarded cards reshuffled into deck.");
   currentDeck.Shuffle();
   discardedCards.Clear();
   currentCard = 0;
}

public void SetPlayers(Player[] newPlayers)
{
   if (newPlayers.Length > 7)
     throw new ArgumentException("A maximum of 7 players may play this" +
                                 " game.");

   if (newPlayers.Length < 2)
     throw new ArgumentException("A minimum of 2 players may play this" +
                                 " game.");

   players = newPlayers;
}

private void DealHands()
{
   for (int p = 0; p < players.Length; p++)
   {
      for (int c = 0; c < 7; c++)
      {
         players[p].PlayHand.Add(playDeck.GetCard(currentCard++));
      }
   }
}
```

```
      public int PlayGame()
      {
         // Code to follow.
      }
   }
}
```

代码段 Ch13CardClient\Game.cs

Program.cs 包含 Main()方法，它启动和运行游戏。这个方法执行以下步骤：

(1) 显示引导画面。

(2) 提示用户输入 2～7 个玩家。

(3) 建立一个 Player 对象数组。

(4) 给每个玩家起个名字，用于初始化数组中的一个 Player 对象。

(5) 创建一个 Game 对象，使用 SetPlayers()方法指定玩家。

(6) 使用 PlayGame()方法启动游戏。

(7) PlayGame()的 int 返回值用于显示一条获胜消息(返回的值是 Player 对象数组中获胜的玩家的索引)。

这个方法的代码(为了清晰起见，加了一些注释)如下所示：

可从
wrox.com
下载源代码

```
static void Main(string[] args)
{
   // Display introduction.
   Console.WriteLine("KarliCards: a new and exciting card game.");
   Console.WriteLine("To win you must have 7 cards of the same suit in" +
                     " your hand.");
   Console.WriteLine();

   // Prompt for number of players.
   bool inputOK = false;
   int choice = -1;
   do
   {
      Console.WriteLine("How many players (2-7)?");
      string input = Console.ReadLine();
      try
      {
         // Attempt to convert input into a valid number of players.
         choice = Convert.ToInt32(input);
         if ((choice >= 2) && (choice <= 7))
            inputOK = true;
      }
      catch
      {
         // Ignore failed conversions, just continue prompting.
      }
   } while (inputOK == false);

   // Initialize array of Player objects.
   Player[] players = new Player[choice];

   // Get player names.
   for (int p = 0; p < players.Length; p++)
   {
      Console.WriteLine("Player {0}, enter your name:", p + 1);
      string playerName = Console.ReadLine();
      players[p] = new Player(playerName);
   }

   // Start game.
   Game newGame = new Game();
   newGame.SetPlayers(players);
   int whoWon = newGame.PlayGame();

   // Display winning player.
   Console.WriteLine("{0} has won the game!", players[whoWon].Name);
}
```

代码段 Ch13CardClient\Program.cs

接着看看应用程序的主体 PlayGame()。由于篇幅所限，这里不准备详细讲解这个方法，而只是加注了一些注释，使之更容易理解。实际上，这些代码都不复杂，仅是较多而已。

每个玩家都可以查看手中的牌和桌面上的一张翻开的牌。他们可以拾取这张牌，或者翻开一张

新牌。在拾取一张牌后，玩家必须扔掉一张牌，如果他们拾取了桌面上的那张牌，就必须用另一张牌替换桌面上的那张牌，或者把扔掉的那张牌放在桌面上那张牌的上面(把扔掉的那张牌添加到 discardedCards 集合中)。

在分析这段代码时，一个关键的问题是 Card 对象的处理方式。必须清楚，这些对象定义为引用类型，而不是值类型(使用结构)。给定的 Card 对象似乎同时存在于多个地方，因为引用可以存在于 Deck 对象、Player 对象的 hand 字段、discardedCards 集合和 playCard 对象(桌面上的当前牌)中。这样就很方便地跟踪扑克牌，特别是可以用于从一副牌中拾取一张新牌。如果牌不在任何玩家的手中，也不在 discardedCards 集合中，才能接受该牌。

代码如下所示：

```
public int PlayGame()
{
  // Only play if players exist.
  if (players == null)
    return -1;

  // Deal initial hands.
  DealHands();

  // Initialize game vars, including an initial card to place on the
  // table: playCard.
  bool GameWon = false;
  int currentPlayer;
  Card playCard = playDeck.GetCard(currentCard++);
  discardedCards.Add(playCard);

// Main game loop, continues until GameWon == true.
do
{
  // Loop through players in each game round.
  for (currentPlayer = 0; currentPlayer < players.Length;
       currentPlayer++)
  {
    // Write out current player, player hand, and the card on the
    // table.
    Console.WriteLine("{0}'s turn.", players[currentPlayer].Name);
    Console.WriteLine("Current hand:");
    foreach (Card card in players[currentPlayer].PlayHand)
    {
      Console.WriteLine(card);
    }
    Console.WriteLine("Card in play: {0}", playCard);

    // Prompt player to pick up card on table or draw a new one.
    bool inputOK = false;
    do
    {
     Console.WriteLine("Press T to take card in play or D to " +
```

```
                    "draw:");
string input = Console.ReadLine();
if (input.ToLower() == "t")
{
   // Add card from table to player hand.
   Console.WriteLine("Drawn: {0}", playCard);
   // Remove from discarded cards if possible (if deck
   // is reshuffled it won't be there any more)
   if (discardedCards.Contains(playCard))
   {
      discardedCards.Remove(playCard);
   }
   players[currentPlayer].PlayHand.Add(playCard);
   inputOK = true;
}
if (input.ToLower() == "d")
{
   // Add new card from deck to player hand.
   Card newCard;
   // Only add card if it isn't already in a player hand
   // or in the discard pile
   bool cardIsAvailable;
   do
   {
      newCard = playDeck.GetCard(currentCard++);
      // Check if card is in discard pile
      cardIsAvailable = !discardedCards.Contains(newCard);
      if (cardIsAvailable)
      {
         // Loop through all player hands to see if newCard is
         // already in a hand.
         foreach (Player testPlayer in players)
         {
            if (testPlayer.PlayHand.Contains(newCard))
            {
               cardIsAvailable = false;
               break;
            }
         }
      }
   } while (!cardIsAvailable);
   // Add the card found to player hand.
   Console.WriteLine("Drawn: {0}", newCard);
   players[currentPlayer].PlayHand.Add(newCard);
   inputOK = true;
  }
} while (inputOK == false);

// Display new hand with cards numbered.
Console.WriteLine("New hand:");
```

```
for (int i = 0; i < players[currentPlayer].PlayHand.Count; i++)
{
   Console.WriteLine("{0}: {1}", i + 1,
                     players[currentPlayer].PlayHand[i]);
}

// Prompt player for a card to discard.
inputOK = false;
int choice = -1;
do
{
   Console.WriteLine("Choose card to discard:");
   string input = Console.ReadLine();
   try
   {
      // Attempt to convert input into a valid card number.
      choice = Convert.ToInt32(input);
      if ((choice > 0) && (choice <= 8))
         inputOK = true;
   }
   catch
   {
      // Ignore failed conversions, just continue prompting.
   }
} while (inputOK == false);

// Place reference to removed card in playCard (place the card
// on the table), then remove card from player hand and add
// to discarded card pile.
playCard = players[currentPlayer].PlayHand[choice - 1];
players[currentPlayer].PlayHand.RemoveAt(choice - 1);
discardedCards.Add(playCard);
Console.WriteLine("Discarding: {0}", playCard);

// Space out text for players
Console.WriteLine();

// Check to see if player has won the game, and exit the player
// loop if so.
GameWon = players[currentPlayer].HasWon();
if (GameWon == true)
   break;
   }
} while (GameWon == false);

// End game, noting the winning player.
return currentPlayer;
}
```

图 13-8 显示了一个正在进行的游戏。

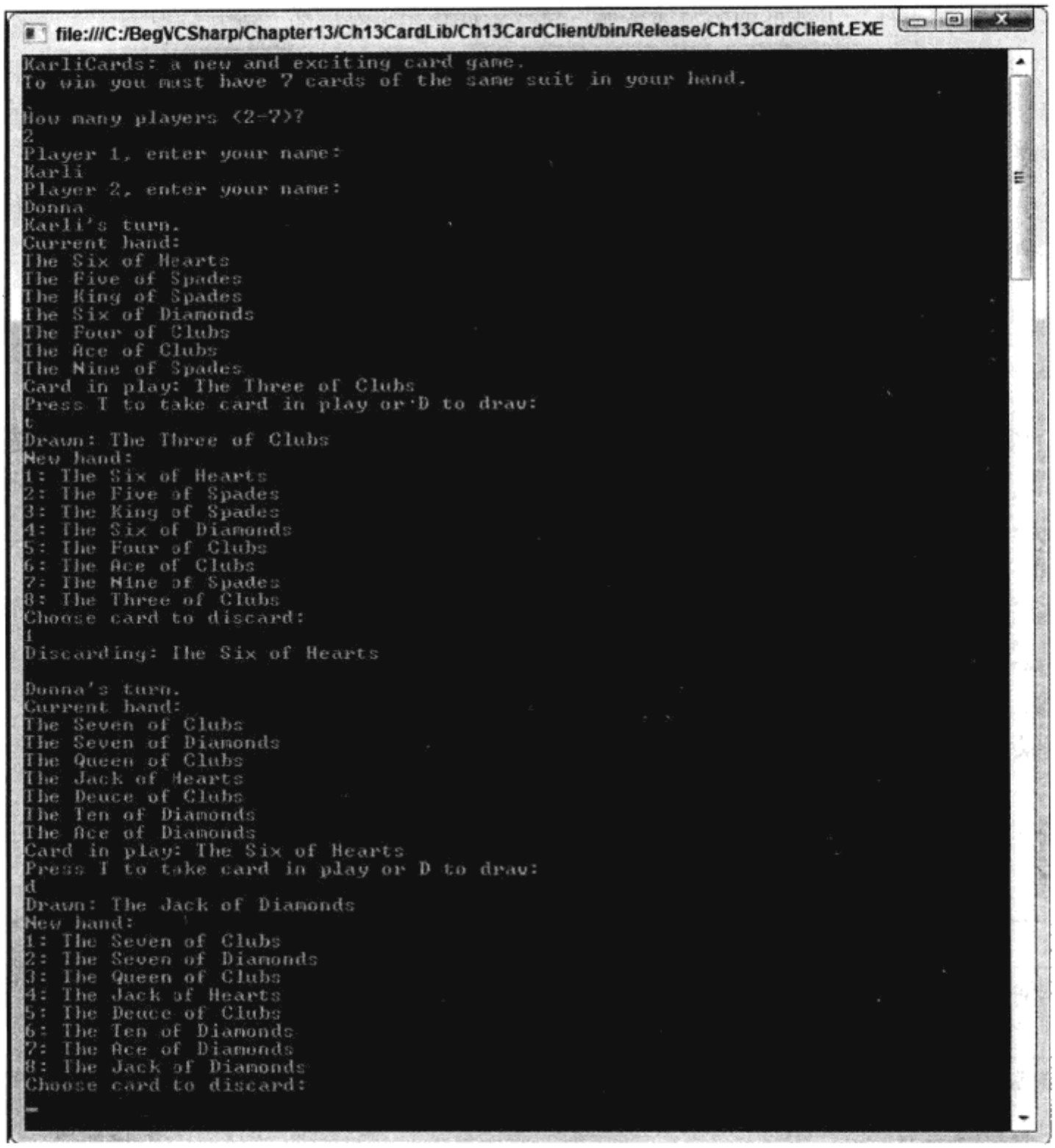


图　13-8

玩这个游戏，确保花一些时间去仔细研究它。应尝试在Reshuffle()方法中设置一个断点，由7个玩家来玩这个游戏。如果在拾取了扑克牌后，马上扔掉它，不需要太长的时间就要重新洗牌了，因为 7 个玩家玩这个游戏时，就只富余 3 张牌。这样就可以注意 3 张牌何时重新翻开，验证程序是否正常执行。

13.5　小结

本章介绍了一些高级技术，扩展讨论了 C#语言的基础知识。首先介绍了名称空间的限定、::运算符和 global 关键字，来确保对类型的引用是希望的类型引用。接着讨论了如何实现自己的异常对象，将更详细的信息传送给异常处理程序。然后在前几章开发的扑克牌游戏项目 CardLib 的代码中使用定制异常。

最后论述了事件和事件处理的重要论题。尽管事件是相当微妙的，刚开始很难理解，但它的代码是非常简单的，读者肯定会在本书的其他地方经常使用事件处理程序。我们还讨论了事件和事件处理的一些简单示例，修改了 CardLib 类库，使用这个类库创建了一个简单的客户扑克牌游戏程序。这个程序可以作为本书到现在为止介绍的所有技术的一个实用示例。

本章不仅完成了 OOP 作为 C#编程技术的讨论，还完成了C#语言的讨论。第 14 章将介绍 C# 3.0 和 4 中新增的特性。

13.6　练习

(1) 编写事件处理程序的代码，这些代码使用了通用语法(object sender, EventArgs e)，该语法将接受本章前面的 Timer.Elapsed 或 Connection.MessageArrived 事件。处理程序应输出一个表示接收了什么类型事件的字符串，并根据引发的事件，输出 MessageArrivedEventArgs 参数的 Message 属性或 ElapsedEventArgs 参数的 SignalTime 属性。

(2) 修改扑克牌游戏示例，设置流行拉米扑克牌的更有趣的取胜条件。即一个玩家要取胜，手中的牌必须包含两套牌，一套由 3 张牌组成，另一套由 4 张牌组成。一套牌应是连续的同花色的牌(例如，3H、4H、5H、6H)或者几张同点的牌(例如，2H、2D、2S)。

附录 A 给出了练习答案。

13.7　本章要点

主　题	重 要 概 念
名称空间限定符	为了避免名称空间限定的模糊性，可以使用::运算符强制编译器使用已创建好的别名。还可以使用 global 命名空间作为顶级名称空间的别名
定制异常	从根类 Exception 中派生，就可以创建自己的异常类。这是有益的，因为可以更多地控制特定异常的捕获，并允许定制包含在异常中的数据，以高效地处理它
事件处理	许多类都提供了事件，在代码中发生某个触发器时，就会引发这些事件。可以为这些事件编写处理程序，在引发事件时执行代码。这种双向的通信方式是响应代码的一种良好机制，无需编写可能导致改变对象的复杂、费解的代码
事件定义	可以定义自己的异常类型，这涉及给事件的处理程序创建指定的事件和委托类型。可以使用标准的、无返回类型的委托类型和派生于 System.EventArgs 的定制事件参数，使事件处理程序有多种用途。还可以使用 EventHandler 和 EventHandler<T>委托类型，以便通过更简单的代码定义事件
匿名方法	为了使代码更便于阅读，常常可以使用匿名方法来替代完整的事件处理方法。这表示，在添加事件处理程序的地方直接定义要在引发事件时执行的代码，为此需要使用 delegate 关键字

第14章

C#语言的改进

本章内容：

- 如何使用初始化器
- var 类型是什么，如何使用类型推理
- 如何使用匿名类型
- dynamic 类型是什么，如何使用它
- 如何使用命名和可选的方法参数
- 如何使用扩展方法
- Lambda 表达式是什么，如何使用它们

C#语言不是一成不变的，C#的发明者 Anders Hejlsberg 和微软公司的其他人一直在更新和改进该语言。在编写本书时，最新的改进都放在 C#语言的第 4 版本中，它作为 VS2010 系列产品的一部分发布。阅读了本书前面的内容后，读者可能会考虑还需要什么其他功能。实际上，C#以前的版本从功能的角度来看并不缺乏什么，但这并不意味着无法进一步简化 C#编程的某些方面，或者 C#和其他技术之间的关系不能更加流畅。

理解这一点的最佳方式是考虑该语言的 1.0 和 2.0 版本之间新增的内容——泛型。泛型虽然非常有用，但并没有真正提供以前不能实现的功能。的确，泛型大大简化了编程，但没有它们，就需要编写更多的代码。我们都不想回到以前没有泛型集合类的日子。无论如何，泛型并不是 C#的基础部分，只是该语言的改进。

C#语言后续的改进也是这样，它们为以前不借助冗长和/或高级编程技术时很难实现的功能提供了新的方式。本章将介绍其中的几处改进，一些改进(例如变体)已经在本章的对应章节做了介绍。

14.1 初始化器

前面的章节学习了如何用各种方式实例化和初始化对象。它们都需要在类定义中添加额外代码，以便使用独立的语句来初始化或实例化对象。我们还了解了如何创建各种类型的集合类，包括泛型

集合类。另外，把集合的创建和在集合中添加数据项合并起来并没有什么简便的方法。

对象初始化器提供了一种简化代码的方式，可以合并对象的实例化和初始化。集合初始化器提供了一种简洁的语法，您使用一个步骤就可以创建和填充集合。本节就介绍如何使用这两个新特性。

14.1.1　对象初始化器

考虑下面的简单类定义：

```
public class Curry
{
   public string MainIngredient {get; set;}
   public string Style {get; set;}
   public int Spiciness {get; set;}
}
```

这个类有 3 个属性，用第10章介绍的自动属性语法来定义。如果希望实例化和初始化这个类的一个对象实例，就必须执行如下几个语句：

```
Curry tastyCurry = new Curry();
tastyCurry.MainIngredient = "panir tikka";
tastyCurry.Style = "jalfrezi";
tastyCurry.Spiciness = 8;
```

如果类定义中未包含构造函数，这段代码就使用 C#编译器提供的默认无参数构造函数。为了简化这个初始化过程，可以提供一个合适的非默认构造函数：

```
public class Curry
{
   public Curry(string mainIngredient, string style,
                int spiciness)
   {
      MainIngredient = mainIngredient;
      Style = style;
      Spiciness = spiciness;
   }

   ...
}
```

这样就可以编写代码，把实例化和初始化合并起来：

```
Curry tastyCurry = new Curry("panir tikka","jalfrezi", 8);
```

这段代码工作得很好，但它会强制使用 Curry 类的代码使用这个构造函数，这将阻止前面使用无参构造函数的代码运行。常常需要提供无参构造函数，在必须序列化类时，情况尤其如此：

```
public class Curry
{
    public Curry()
    {
    }

    ...
}
```

现在可以用任意方式实例化和初始化 Curry 类，但已在最初的类定义中添加几行代码，与提供基本的执行代码相比，这种方法并没有做更多的工作。

进入对象初始化器(object initializer)，这是无需在类中添加额外的代码(如此处详细说明的构造函数)就可以实例化和初始化对象的方式。实例化对象时，要为每个需初始化的、可公开访问的属性或字段使用名称-值对，来提供其值。其语法如下：

```
<className> <variableName> = new <className>
{
   <propertyOrField1> = <value1>,
   <propertyOrField2> = <value2>,
   ...
   <propertyOrFieldN> = <valueN>
};
```

例如，重写前面的代码，实例化和初始化 Curry 类型的一个对象，如下所示：

```
Curry tastyCurry = new Curry
{
   MainIngredient = "panir tikka",
   Style = "jalfrezi",
   Spiciness = 8
};
```

我们常常可以把这样的代码放在一行上，而不会严重影响可读性。

使用对象初始化器时，不必显式调用类的构造函数。如果像上述代码那样省略构造函数的括号，就自动调用默认的无参构造函数。这是在初始化器设置参数值之前调用的，以便在需要时为默认构造函数中的参数提供默认值。另外，可以调用特定的构造函数。同样，先调用这个构造函数，所以在构造函数中对公共属性进行的初始化可能会被初始化器中提供的值覆盖。必须按顺序访问所使用的构造函数(如果没有显式指出，就执行默认的构造函数)，对象初始化器才能正常工作。

如果要用对象初始化器进行初始化的属性比本例中使用的简单类型还复杂，可以使用嵌套的对象初始化器，即使用与前面相同的语法：

```
Curry tastyCurry = new Curry
{
   MainIngredient = "panir tikka",
   Style = "jalfrezi",
   Spiciness = 8,
   Origin = new Restaurant
   {
      Name = "King's Balti",
      Location = "York Road",
      Rating = 5
   }
};
```

这里初始化了一个 Restaurant 类型(这里没有列出)的 Origin 属性。代码初始化了 Origin 属性的 3 个特性：Name、Location 和 Rating，其值的类型分别是 string、string 和 int 类型。这个初始化使用了嵌套的对象初始化器。

注意，对象初始化器没有替代非默认的构造函数。在初始化对象时，可以使用对象初始化器来

设置属性和字段值，但这并不意味着总是知道需要初始化什么状态。通过构造函数，可以准确地指定对象需要什么值才能起作用，再执行代码，以立即响应这些值。

14.1.2　集合初始化器

第 5 章描述了如何使用如下语法，用值来初始化数组：

```
int[] myIntArray = new int[5] {5, 9, 10, 2, 99};
```

这是一种合并实例化和初始化数组的简捷方式。集合初始化器只是把这个语法扩展到集合上：

```
List < int > myIntCollection = new List < int > {5, 9, 10, 2, 99};
```

通过合并对象和集合初始化器，就可以用简洁的代码配置集合了。下面的代码：

```
List < Curry > curries = new List < Curry > ();
curries.Add(new Curry("Chicken", "Pathia", 6));
curries.Add(new Curry("Vegetable", "Korma", 3));
curries.Add(new Curry("Prawn", "Vindaloo", 9));
```

可以用如下代码替换：

```
List < Curry > moreCurries = new List < Curry >
{
   new Curry
   {
      MainIngredient = "Chicken",
      Style = "Pathia",
      Spiciness = 6
   },
   new Curry
   {
      MainIngredient = "Vegetable",
      Style = "Korma",
      Spiciness = 3
   },
   new Curry
   {
      MainIngredient = "Prawn",
      Style = "Vindaloo",
      Spiciness = 9
   }
};
```

这非常适合于主要用于数据表示的类型，而且，集合初始化和本书后面介绍的 LINQ 技术一起使用时效果极佳。

下面的示例说明了如何使用对象和集合初始化器。

试一试：使用初始化器

(1) 创建一个新的控制台应用程序 Ch14Ex01，把它保存在 C:\BegVCSharp\Chapter14 目录下。

(2) 在 Solution Explorer 窗口中右击项目名称，选择 Add | Existing Item 选项。

(3) 在 C:\BegVCSharp\Chapter12\Ch12Ex04\Ch12Ex04 目录中选择 Animal.cs、Cow.cs、Chicken.cs、

SuperCow.cs 和 Farm.cs 文件，单击 Add 按钮。

(4) 修改文件中已添加的名称空间声明，如下所示：

```
namespace Ch14Ex01
```

(5) 删除 Cow、Chicken 和 SuperCow 类的构造函数。

(6) 修改 Program.cs 中的代码，如下所示：


可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
    Farm < Animal > farm = new Farm < Animal >
    {
        new Cow { Name="Norris" },
        new Chicken { Name=" Rita" },
        new Chicken(),
        new SuperCow { Name="Chesney" }
    };
    farm.MakeNoises();
    Console.ReadKey();
}
```

代码段 Ch14Ex01\Program.cs

(7) 生成应用程序，会得到如图 14-1 所示的生成错误。

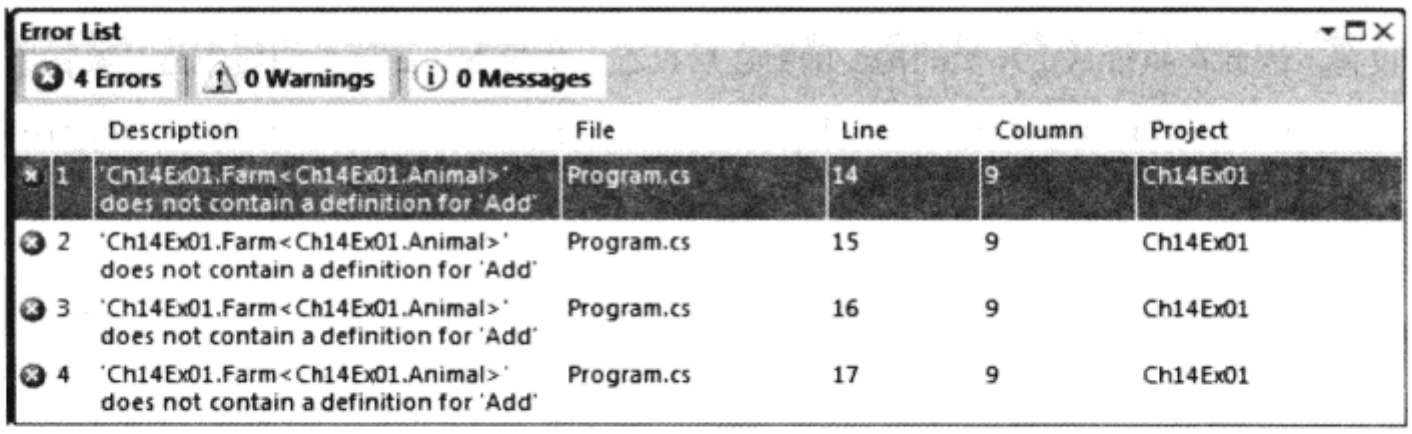


图　14-1

(8) 给 Farm.cs 添加如下代码：


可从 wrox.com 下载源代码

```
public class Farm<T> : IEnumerable<T>
    where T : Animal
{
    public void Add(T animal)
    {
        animals.Add(animal);
    }

    ...
```

代码段 Ch14Ex01\Farm.cs

(9) 运行应用程序，结果如图 14-2 所示。

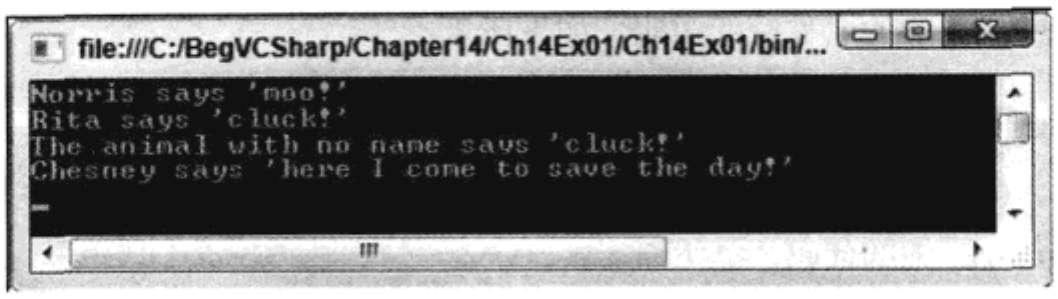


图　14-2

示例的说明

这个示例合并了对象和集合初始化器，用一个步骤创建并填充了一个对象集合。它使用了前面章节介绍的 farmyard 对象集合，但需要对用于这些类的初始化器进行两处修改。

首先，给派生于 Animal 基类的类删除构造函数。可以删除这些构造函数，是因为它们设置了动物的 Name 属性，而这里使用对象初始化器来完成。另外，还可以添加默认的构造函数。无论采用哪种方式，在使用默认的构造函数时，会根据基类中的默认构造函数来初始化 Name 属性，代码如下：

```
public Animal()
{
  name = "The animal with no name";
}
```

但是，对象初始化器与派生于 Animal 类的类一起使用时，初始化器设置的任何属性都在对象实例化后设置，因此在执行这个基类的构造函数之后设置。如果 Name 属性的值作为对象初始化器的一部分提供，这个值就会覆盖默认值。在示例代码中，为添加到集合中的所有项设置了 Name 属性，只有一项除外。

其次，必须给 Farm 类添加 Add()方法，否则会响应如下形式的一系列编译错误：

```
'Ch14Ex01.Farm < Ch14Ex01.Animal >' does not contain a definition for 'Add'
```

这个错误显示出了集合初始化器的部分底层功能。在后台，编译器为在集合初始化器中提供的每一项调用集合的 Add()方法。Farm 类通过 Animals 属性提供了一个 Animal 对象集合。编译器猜不出这就是要(通过 Animals.Add())填充的属性，所以代码会失败。为了更正这个问题，可以把 Add()方法添加到类中，类通过对象初始化器进行初始化。

还可以修改示例中的代码，为 Animals 属性提供一个嵌套的初始化器，如下所示：

```
static void Main(string[] args)
{
   Farm < Animal > farm = new Farm < Animal >
   {
        Animals =
        {
           new Cow { Name="Norris" },
           new Chicken { Name="Rita" },
           new Chicken(),
           new SuperCow { Name="Chesney" }
        }
```

```
        };
        farm.MakeNoises();
        Console.ReadKey();
    }
```

有了此代码，就不需要为 Farm 类提供 Add()方法了。这个备用技巧适用于包含多个集合的类。在这种情况下，用包含类的 Add()方法添加的集合没有其他方式。

14.2　类型推理

本书前面介绍过 C#是一种强类型化的语言，这表示每个变量都有固定的类型，只能用于接受该类型的代码中。在前面的所有代码示例中，都用如下形式的代码来声明变量：

```
<type> <varName> ;
```

或者

```
<type> <varName> = <value> ;
```

下面的代码显示了变量 varName 的类型：

```
int myInt = 5;
Console.WriteLine(myInt);
```

把鼠标指针停放在变量标识符上，IDE 就会显示该变量的类型，如图 14-3 所示。

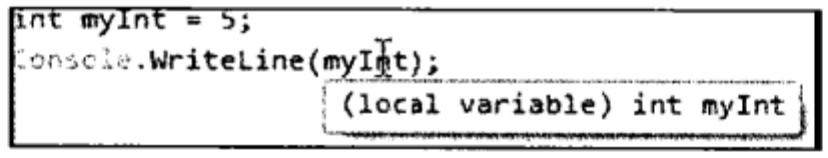


图　14-3

C# 3.0 引入了新关键字 var，它可以替代前面代码中的 type：

```
var <varName> = <value> ;
```

在这行代码中，变量<varName>隐式地类型化为 value 的类型。注意，类型的名称并不是 var。在下面的代码中：

```
var myVar = 5;
```

myVar 是 int 类型的变量，而不是 var 类型的变量，如图 14-4 所示，IDE 也显示了其类型。

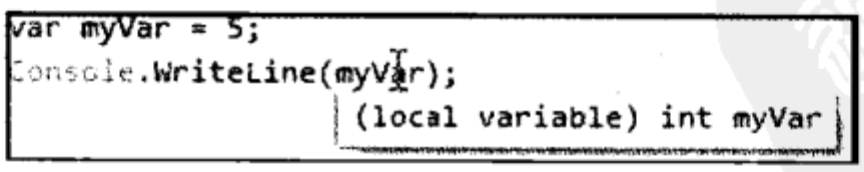


图　14-4

这是非常重要的一点。使用 var 时，并不是声明了一个没有类型的变量，也不是声明了一个类型可以变化的变量。否则，C#就不再是强类型化的语言了。我们只需利用编译器确定变量的类型即可。

.NET 4 引入的动态类型扩展了 C#是强类型化语言的定义，参见本章后面的“动态查找”一节。

如果编译器不能确定用 var 声明的变量类型，代码就不会编译。因此，在用 var 声明变量时，必须同时初始化该变量，因为如果没有初始值，编译器就不能确定变量的类型。下面的代码就不能编译：

```
var myVar;
```

var 关键字还可以通过数组初始化器来推断数组的类型：

```
var myArray = new[] {4, 5, 2};
```

在这行代码中，myArray 的类型被隐式地设置为 int[]。在用这种方式隐式指定数组的类型时，初始化器中使用的数组元素必须是以下情形中的一种：

- 相同的类型
- 相同的引用类型或空
- 所有元素的类型都可以隐式地转换为一个类型

如果应用最后一条规则，元素可以转换的类型就称为数组元素的最佳类型。如果这个最佳类型有任何含糊的地方，即所有元素的类型都可以隐式转换为两种或更多的类型，代码就不会编译。我们会接收到错误，错误中指出没有最佳类型：

```
var myArray = new[] {4, "not an int", 2};
```

还要注意数字值从来都不会解释为可空类型，所以下面的代码无法编译：

```
var myArray = new[] {4, null, 2};
```

但可以使用标准的数组初始化器，使如下代码编译：

```
var myArray = new int?[] {4, null, 2};
```

最后一点要说明的是，标识符 var 并非不能用于类名。这意味着，如果代码在其作用域中(在同一个名称空间或引用的名称空间中)有一个 var 类，就不能使用 var 关键字的隐式类型化功能。

类型推理功能本身并不是很有效，因为在本节前面的代码中，它只会使事情更复杂。使用 var 会加大判断给定变量的类型的难度。但是如本章后面所述，推断类型的概念非常重要，因为它是其他技术的基础。下一个主题是匿名类型，它就以推断类型为基础。

14.3 匿名类型

在编写程序一段时间后，会发现我们要花很多时间为数据表示创建简单、乏味的类，在数据库应用程序中尤其如此。常常有一系列类只提供属性。本章前面的 Curry 类就是一个很好的例子：

```
public class Curry
```

```
{
    public string MainIngredient {get; set;}
    public string Style {get; set;}
    public int Spiciness {get; set;}
}
```

这个类什么也没做，只是存储结构化数据。在数据库或电子表格中，可以把这个类看作表中的一行。可以保存这个类的实例的集合类应表示表或电子表格中的多个行。

这是类完全可以接受的一种用法，但编写这些类的代码比较单调，对底层数据模式的任何修改都需要添加、删除或修改定义类的代码。

匿名类型(anonymous type)是简化这个编程模型的一种方式。其理念是使用 C#编译器根据要存储的数据自动创建类型，而不是定义简单的数据存储类型。

可以按如下方式实例化前面的 Curry 类型：

```
Curry curry = new Curry
{
    MainIngredient = "Lamb",
    Style = "Dhansak",
    Spiciness = 5
};
```

也可以使用匿名类型，如下所示：

```
var curry = new
{
    MainIngredient = "Lamb",
    Style = "Dhansak",
    Spiciness = 5
};
```

这里有两个区别。第一，使用了 var 关键字。这是因为匿名类型没有可以使用的标识符。稍后可以看到，它们在内部有一个标识符，但不能在代码中使用。第二，在 new 关键字的后面没有指定类型名称，这是编译器确定我们要使用匿名类型的方式。

IDE 检测到匿名类型定义后，就会相应地更新 IntelliSense。通过前面的声明，可以看到如图 14-5 所示的匿名类型。

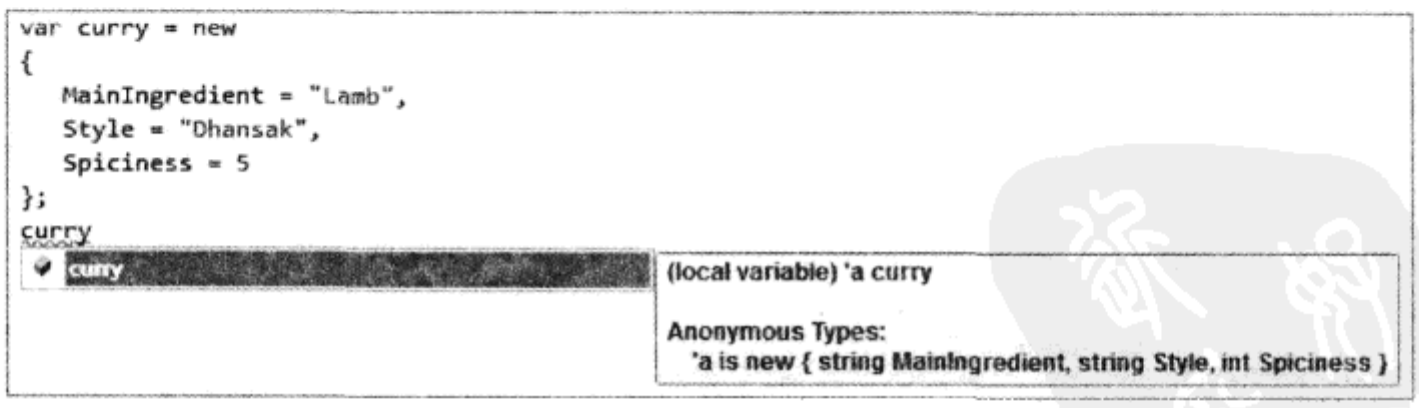


图　14-5

其中，变量 curry 的类型是'a。显然，不能在代码中使用这个类型——它甚至不是合法的标识符名称。'符号仅用于在 IntelliSense 中表示匿名类型。IntelliSense 也允许查看匿名类型的成员，如图 14-6 所示。

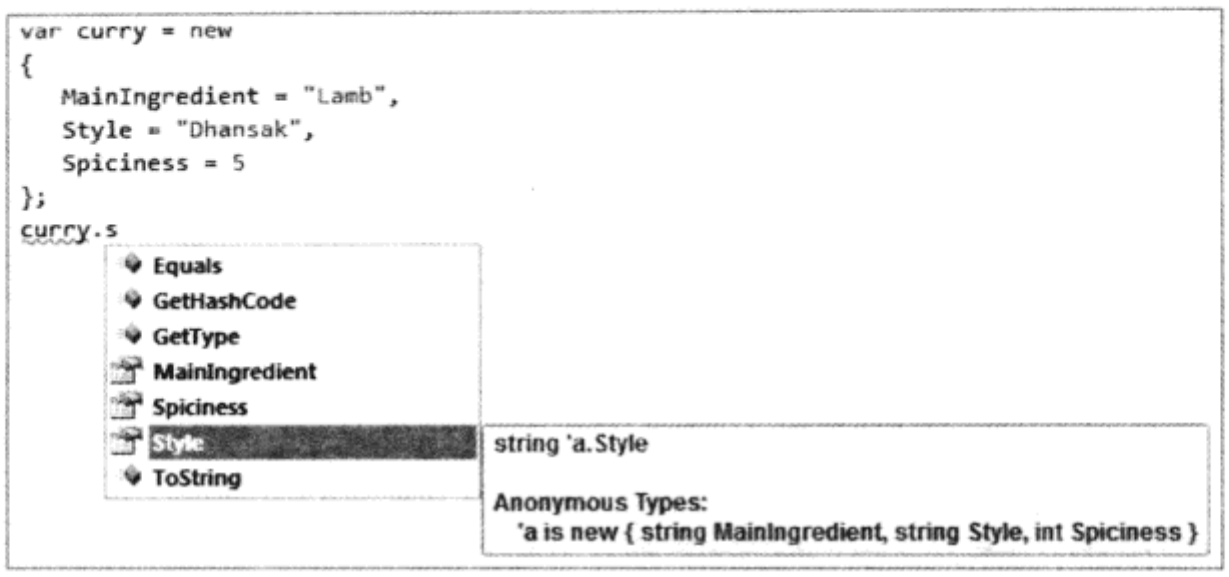


图　14-6

注意，这里显示的属性定义为只读属性。这表示，如果要在数据存储对象中修改属性的值，就不能使用匿名类型。

还实现了匿名类型的其他成员，如下面的示例所示。

试一试：使用匿名类型

(1) 创建一个新的控制台应用程序 Ch14Ex02，把它保存在 C:\BegVCSharp\Chapter14 目录中。

(2) 修改 Program.cs 中的代码，如下所示：

可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
    var curries = new[]
    {
        new
        {
            MainIngredient = "Lamb",
            Style = "Dhansak",
            Spiciness = 5
        },
        new
        {
            MainIngredient = "Lamb",
            Style = "Dhansak",
            Spiciness = 5
        },
        new
        {
            MainIngredient = "Chicken",
            Style = "Dhansak",
            Spiciness = 5
        }
    };
    Console.WriteLine(curries[0].ToString());
    Console.WriteLine(curries[0].GetHashCode());
    Console.WriteLine(curries[1].GetHashCode());
    Console.WriteLine(curries[2].GetHashCode());
    Console.WriteLine(curries[0].Equals(curries[1]));
```

```
        Console.WriteLine(curries[0].Equals(curries[2]));
        Console.WriteLine(curries[0] == curries[1]);
        Console.WriteLine(curries[0] == curries[2]);

        Console.ReadKey();
    }
```

代码段 Ch14Ex02\Program.cs

(3) 运行应用程序，结果如图 14-7 所示。

图　14-7

示例的说明

这个示例创建了一个匿名类型对象的数组，然后使用它测试匿名类型提供的成员。创建匿名类型对象的数组的代码如下：

```
var curries = new[]
{
    new
    {
        MainIngredient = "Lamb",
        Style = "Dhansak",
        Spiciness = 5
    },
    ...
};
```

这段代码通过本节和前面“类型推理”一节中介绍的语法，使用了隐式类型化为匿名类型的数组。结果是 curries 变量包含匿名类型的 3 个实例。

在创建了这个数组后，代码首先输出在匿名类型上调用 ToString()的结果：

```
Console.WriteLine(curries[0].ToString());
```

输出结果如下：

```
{ MainIngredient = Lamb, Style = Dhansak, Spiciness = 5 }
```

匿名类型上的 ToString()的实现输出了为该类型定义的每个属性的值。

接着，代码在数组的 3 个对象上分别调用 GetHashCode()：

```
Console.WriteLine(curries[0].GetHashCode());
Console.WriteLine(curries[1].GetHashCode());
Console.WriteLine(curries[2].GetHashCode());
```

GetHashCode()执行时，应根据对象的状态为对象返回一个唯一的整数。数组中的前两个对象有相同的属性值，所以其状态是相同的。这些调用的结果是前两个对象的整数相同，第三个对象的整数不同。结果如下：

```
294897435
294897435
621671265
```

接着，调用 Equals()方法比较第一个对象和第二个对象，再比较第一个对象和第三个对象：

```
Console.WriteLine(curries[0].Equals(curries[1]));
Console.WriteLine(curries[0].Equals(curries[2]));
```

结果如下：

```
True
False
```

匿名类型上的Equals()的实现比较对象的状态，如果一个对象的每个属性值都与另一个对象的对应属性值相同，结果就是 true。

但使用==运算符不会得到这样的结果。如前几章所述，==运算符比较对象引用。最后一部分代码进行与上一段代码相同的比较，但用==替代了Equals()方法：

```
Console.WriteLine(curries[0] == curries[1]);
Console.WriteLine(curries[0] == curries[2]);
```

Curries 数组中的每一项都引用匿名类型的不同实例，所以在两种情况下结果都是 false。输出结果与预期的相同：

```
False
False
```

有趣的是，在创建匿名类型的实例时，编译器会注意到，参数是相同的，所以创建同一个匿名类型的 3 个实例——而不是 3 个不同的匿名类型。但是，这并不意味着实例化匿名类型的对象时，编译器会查找匹配的类型。即使在其他地方定义了一个有匹配属性的类，如果使用了匿名类型语法，也只创建(或重用，如本例所示)一个匿名类型。

14.4　动态查找

如前所述，var 关键字本身并不是一个类型，所以并没有违反 C#的“强类型化”方法论。但在 C# 4 中做了少许改动。C# 4 引入了“动态变量”的概念，顾名思义，动态变量是类型不固定的变量。

引入动态变量的主要目的是在许多情况下，都希望使用 C#处理另一种语言创建的对象。这包括与旧技术的交互操作，例如 Component Object Model (COM)，以及处理动态语言，例如 JavaScript、Python 和 Ruby。在过去，没有非常详细的实现细节，使用 C#访问这些语言所创建的对象的方法和属性，需要用到笨拙的语法。例如，假定代码从 JavaScript 中获得了一个带 Add()方法的对象，该方法把两个数字加在一起。如果没有动态查找功能，调用这个方法的代码就如下所示：

```
ScriptObject jsObj = SomeMethodThatGetsTheObject();
int sum = Convert.ToInt32(jsObj.Invoke("Add", 2, 3));
```

ScriptObject 类型(这里不深入探讨)提供了一种访问 JavaScript 对象的方式，但不能执行如下操作：

```
int sum = jsObj.Add(2, 3);
```

动态查找功能改变了这一切，它允许编写上述代码，但如下面几节所述，这个功能是有代价的。

另一个可以使用动态查找功能的情形是处理未知类型的 C#对象。这听起来似乎很古怪，但这种情形出现的次数比我们想象得多。如果需要编写一些泛型代码，来处理它接收的输入，这也是一个重要的功能。处理这种情形的“旧”方法称为“反射(reflection)”，它涉及使用类型信息来访问类型和成员。实际上，反射的语法非常类似于上述代码中访问 JavaScript 对象的语法，也非常麻烦。

在后台，动态查找功能由 Dynamic Language Runtime (动态语言运行库，DLR)支持。与 CLR 一样，DLR 是.NET 4 的一部分。DLR 的精确描述及其如何简化交互操作超出了本书的范围，这里仅对如何在 C#中使用它感兴趣。

14.4.1　dynamic 类型

C# 4 引入了 dynamic 关键字，以用于定义变量。例如：

```
dynamic myDynamicVar;
```

与前面介绍的 var 关键字不同，的确存在 dynamic 类型，所以在声明 myDynamicVar 时，无需初始化它的值。

dynamic 类型不同寻常之处是，它仅在编译期间存在，在运行期间它会被 System.Object 类型替代。这是较细微的实现细节，但必须记住这一点，因为这可能澄清了后面的一些讨论。一旦有了动态变量，就可以继续访问其成员(这里没有列出实际获取变量值的代码)。

```
myDynamicVar.DoSomething("With this!");
```

无论 myDynamicVar 实际包含什么值，这行代码都会编译。但是，如果所请求的成员不存在，在执行这行代码时会生成一个 RuntimeBinderException 类型的异常。

实际上，像这样的代码提供了一个应在运行期间应用的“处方”。检查 myDynamicVar 的值，定位带一个字符串参数的 DoSomething()方法，并在需要时调用它。

这最好举例说明。

警告：下面的示例仅用于演示！一般情况下，应仅在动态类型是唯一的选项时使用它们，例如处理非.NET 对象。

试一试：使用动态类型

(1) 创建一个新的控制台应用程序 Ch14Ex03，把它保存在 C:\BegVCSharp\Chapter14 目录中。

(2) 修改 Program.cs 中的代码，如下所示：


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CSharp.RuntimeBinder;
namespace Ch14Ex03
{
    class MyClass1
    {
        public int Add(int var1, int var2)
        {
            return var1 + var2;
        }
    }

    class MyClass2
    {
    }

    class Program
    {
        static int callCount = 0;

        static dynamic GetValue()
        {
            if (callCount++ == 0)
            {
                return new MyClass1();
            }
            return new MyClass2();
        }

        static void Main(string[] args)
        {
            try
            {
                dynamic firstResult = GetValue();
                dynamic secondResult = GetValue();
                Console.WriteLine("firstResult is: {0}",
                    firstResult.ToString());
                Console.WriteLine("secondResult is: {0}",
                    secondResult.ToString());
                Console.WriteLine("firstResult call: {0}",
                    firstResult.Add(2, 3));
                Console.WriteLine("secondResult call: {0}",
                    secondResult.Add(2, 3));
            }
            catch (RuntimeBinderException ex)
            {
```

```
                Console.WriteLine(ex.Message);
            }
            Console.ReadKey();
        }
    }
}
```

代码段 Ch14Ex03\Program.cs

(3) 运行应用程序，结果如图 14-8 所示。

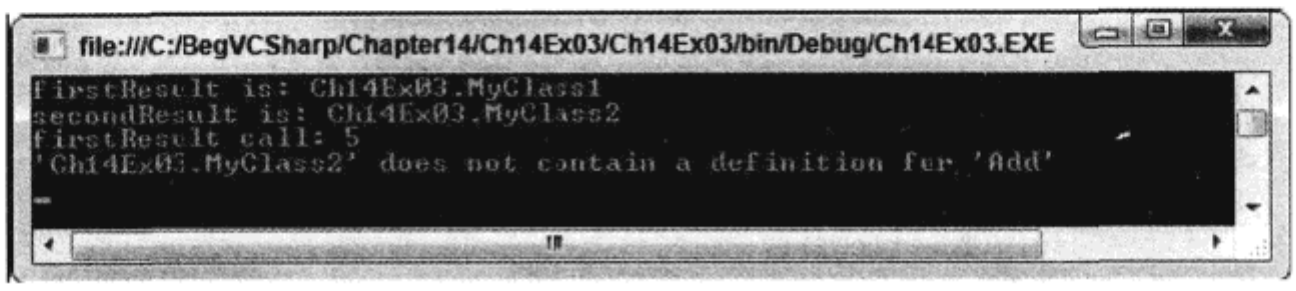


图　14-8

示例的说明

这个示例使用一个方法返回两个类型的对象中的一个，以获取动态值，再尝试使用所获取的对象。代码在编译时没有遇到任何问题，但尝试访问不存在的方法时，抛出(并处理)了一个异常。

首先，为包含 RuntimeBindingException 异常的名称空间添加一个 using 语句：

```
using Microsoft.CSharp.RuntimeBinder;
```

接着定义两个类 MyClass1 和 MyClass2，其中 MyClass1 包含 Add()方法，而 MyClass2 不含成员：

```
class MyClass1
{
    public int Add(int var1, int var2)
    {
        return var1 + var2;
    }
}

class MyClass2
{
}
```

还要给 Program 类添加一个字段(callCount)和一个方法(GetValue())，以获取其中一个类的实例：

```
static int callCount = 0;

static dynamic GetValue()
{
    if (callCount++ == 0)
    {
        return new MyClass1();
    }
```

```
    return new MyClass2();
}
```

使用一个简单的调用计数器，这样，第一次调用这个方法时，返回 MyClass1 的一个实例，之后返回 MyClass2 的实例。注意 dynamic 关键字可以用作方法的返回类型。

接着，Main()中的代码调用 GetValue()方法两次，再尝试在返回的两个值上依次调用 GetString()和 Add()。这些代码放在 try ... catch 块中，以捕获可能发生的 RuntimeBinderException 类型的异常。

```
static void Main(string[] args)
{
    try
    {
        dynamic firstResult = GetValue();
        dynamic secondResult = GetValue();
        Console.WriteLine("firstResult is: {0}",
            firstResult.ToString());
        Console.WriteLine("secondResult is: {0}",
            secondResult.ToString());
        Console.WriteLine("firstResult call: {0}",
            firstResult.Add(2, 3));
        Console.WriteLine("secondResult call: {0}",
            secondResult.Add(2, 3));
    }
    catch (RuntimeBinderException ex)
    {
        Console.WriteLine(ex.Message);
    }

    Console.ReadKey();
}
```

可以肯定，调用 secondResult.Add()时会抛出一个异常，因为在 MyClass2 上不存在这个方法。异常消息说明了这一点。

dynamic 关键字也可以用于其他需要类型名的地方，例如方法参数。Add()方法可以重写为：

```
public int Add(dynamic var1, dynamic var2)
{
    return var1 + var2;
}
```

这对结果没有任何影响。在这个例子中，传送给 var1 和 var2 的值在运行期间检查，以确定加号+是否存在一个兼容的运算符定义。如果传送了两个 int 值，就存在这样的运算符。如果使用了不兼容的值，就抛出 RuntimeBinderException 异常。例如，如果尝试：

```
Console.WriteLine("firstResult call: {0}", firstResult.Add("2", 3));
```

异常消息就如下所示：

```
Cannot implicitly convert type 'string' to 'int'
```

从这里获得的教训是动态类型是非常强大的，但有一个警告。如果用强类型代替动态类型，就完全可以避免抛出这些异常。对于大多数自己编写的 C#代码，应避免使用 dynamic 关键字。但是，

如果需要使用它，就应使用它，并会喜欢上它——而不像过去那些可怜的程序员那样没有这个强大的工具可用。

14.4.2　IDynamicMetaObjectProvider

在继续之前，应注意如何使用动态类型，或者更确切地讲，在运行期间对成员访问应用某种技术时会发生什么。实际上，有 3 种不同的方式访问成员：

- 如果动态值是 COM 对象，就使用 COM 技术访问成员(通过 IUnknown 接口，但这里不需要了解它)。
- 如果动态值支持 IDynamicMetaObjectProvider 接口，就使用该接口访问类型成员。
- 如果不能使用上述两种技术，就使用反射。

第二种情形比较有趣，它涉及到 IDynamicMetaObjectProvider 接口。这里不探讨具体的细节，只是注意可以实现这个接口，来准确地控制在运行期间访问成员时会发生什么。但这是高级编程图书的一个主题，因此这里不讨论。

14.5　高级方法参数

C# 4 扩展了定义和使用方法参数的方式。这主要是为了响应使用外部定义的接口时出现的一个特殊问题，例如 Microsoft Office 编程模型。其中，一些方法有大量的参数，许多参数并不是每次调用都需要的。过去，这意味着需要一种方式指定缺失的参数，否则在代码中会出现许多空值：

```
RemoteCall(var1, var2, null, null, null, null, null);
```

在这行代码中，null 值表示什么并不明显，或者它们为什么省略并不清楚。

也许，在理想情况下，这个 RemoteCall()方法有多个重载版本，其中一个重载版本仅需要两个参数：

```
RemoteCall(var1, var2);
```

但是，这需要更多带其他参数组合的方法，这本身就会带来更多问题(要维护更多的代码，增加了代码的复杂性等)。

Visual Basic 等语言以另一种方式处理这种情况，即允许使用命名参数和可选参数。在 C# 4 版本中也允许这样做，这是所有.NET 语言的演化趋于一致的一种方式。

下面几节介绍如何使用这些新的参数类型。

14.5.1　可选参数

调用方法时，常常给某个参数传送相同的值。例如，这可能是一个布尔值，以控制方法操作中的不重要部分。具体而言，考虑下面的方法定义：

```
public List<string> GetWords(
    string sentence,
    bool capitalizeWords)
{
    ...
}
```

无论给 capitalizeWords 参数传送什么值，这个方法都会返回一系列 string 值，每个 string 值都是输入句子中的一个单词。根据这个方法的使用方式，可能需要把返回的单词列表转换为大写(也许要格式化一个标题)。但在大多数情况下，并不需要这么做，所以大多数调用如下所示：

```
List<string> words = GetWords(sentence, false);
```

为了把这种方式变成“默认”方式，可以声明第二个方法，如下所示：

```
public List<string> GetWords(string sentence)
{
    return GetWords(sentence, false);
}
```

这个方法调用第二个方法，并给 capitalizeWords 传送值 false。

这么做没有任何错误，但可以想象在使用更多的参数时，这种方式会非常复杂。

另一种方式是把 capitalizeWords 参数变成可选参数。这需要在方法定义中将参数定义为可选参数，这需要提供一个默认值，如果没有提供值，就使用默认值，如下所示：

```
public List<string> GetWords(
    string sentence,
    bool capitalizeWords = false)
{
    ...
}
```

如果以这种方式定义方法，就可以提供一个或两个参数，只有希望 capitalizeWords 是 true 时，才需要第二个参数。

1. 可选参数的值

如上一节所述，方法定义了一个可选参数，其语法如下所示：

```
<parameterType> <parameterName> = <defaultValue>
```

给<defaultValue>用作默认值的内容有一些限制：默认值必须是字面值、常量值、新对象实例或者默认值类型值。因此不会编译下面的代码：

```
public bool CapitalizationDefault;

public List<string> GetWords(
    string sentence,
   bool capitalizeWords = CapitalizationDefault)
{
    ...
}
```

为了使上述代码可以工作，CapitalizationDefault 值必须定义为常量：

```
public const bool CapitalizationDefault = false;
```

这是否有意义取决于具体的情形，在大多数情况下，最好提供一个字面值，就像上一节那样。

2. 可选参数的顺序

使用可选值时，它们必须位于方法的参数列表末尾。没有默认值的参数不能放在有默认值的参数后面。

因此下面的代码是非法的：

```
public List<string> GetWords(
    bool capitalizeWords = false,
    string sentence)
{
    ...
}
```

其中，sentence 是必选参数，因此必须放在可选参数 capitalizedWords 的前面。

14.5.2　命名参数

使用可选参数时，可能会发现某个方法有几个可选参数，但您可能只想给第三个可选参数传送值。从上一节介绍的语法来看，如果不提供前两个可选参数的值，就无法给第三个可选参数传送值。

C# 4 引入了命名参数(named parameters)，它允许指定要使用哪个参数。这不需要在方法定义中进行任何特殊处理，它是一个在调用方法时使用的技术。其语法如下：

```
MyMethod(
    <param1Name>: <param1Value>,
    ...
    <paramNName>: <paramNValue>);
```

参数的名称是在方法定义中使用的变量名。

只要命名参数存在，就可以用这种方式指定需要的任意多个参数，而且参数的顺序是任意的。命名参数也是可选的。

可以仅给方法调用中的某些参数使用命名参数。当方法签名中有多个可选参数和一些必选参数时，这是非常有用的。可以先指定必选参数，再指定命名的可选参数。

例如：

```
MyMethod(
    requiredParameter1Value,
    optionalParameter5: optionalParameter5Value);
```

但注意，如果混合使用命名参数和位置参数，就必须先包含所有的位置参数，其后是命名参数。只要使用命名参数，参数的顺序也可以不同。例如：

```
MyMethod(
    optionalParameter5: optionalParameter5Value,
    requiredParameter1: requiredParameter1Value);
```

此时，必须包含所有必选参数的值。

下面的示例介绍了如何使用命名参数和可选参数。

试一试：使用命名参数和可选参数

(1) 创建一个新的控制台应用程序 Ch14Ex04，把它保存在 C:\BegVCSharp\Chapter14 目录中。

(2) 在项目中添加一个类 WordProcessor，修改其代码，如下所示：


```
public static class WordProcessor
{
    public static List<string> GetWords(
        string sentence,
        bool capitalizeWords = false,
        bool reverseOrder = false,
        bool reverseWords = false)
    {
        List<string> words = new List<string>(sentence.Split(' '));
        if (capitalizeWords)
            words = CapitalizeWords(words);
        if (reverseOrder)
            words = ReverseOrder(words);
        if (reverseWords)
            words = ReverseWords(words);
        return words;
    }

    private static List<string> CapitalizeWords(
        List<string> words)
    {
        List<string> capitalizedWords = new List<string>();
        foreach (string word in words)
        {
            if (word.Length == 0)
                continue;
            if (word.Length == 1)
                capitalizedWords.Add(
                    word[0].ToString().ToUpper());
            else
                capitalizedWords.Add(
                    word[0].ToString().ToUpper()
                    + word.Substring(1));
        }

        return capitalizedWords;
    }

    private static List<string> ReverseOrder(List<string> words)
    {
        List<string> reversedWords = new List<string>();
        for (int wordIndex = words.Count - 1;
             wordIndex >= 0; wordIndex--)
            reversedWords.Add(words[wordIndex]);

        return reversedWords;
    }

    private static List<string> ReverseWords(List<string> words)
    {
        List<string> reversedWords = new List<string>();
        foreach (string word in words)
            reversedWords.Add(ReverseWord(word));
```

```
        return reversedWords;
    }

    private static string ReverseWord(string word)
    {
        StringBuilder sb = new StringBuilder();
        for (int characterIndex = word.Length - 1;
             characterIndex >= 0; characterIndex--)
            sb.Append(word[characterIndex]);

        return sb.ToString();
    }
}
```

代码段 Ch14Ex04\WordProcessor.cs

(3) 修改 Program.cs 中的代码，如下所示：


```
static void Main(string[] args)
{
    string sentence = "'twas brillig, and the slithy toves did gyre "
        + "and gimble in the wabe:";
    List<string> words;

    words = WordProcessor.GetWords(sentence);
    Console.WriteLine("Original sentence:");
    foreach (string word in words)
    {
        Console.Write(word);
        Console.Write(' ');
    }

    Console.WriteLine('\n');

    words = WordProcessor.GetWords(
        sentence,
        reverseWords: true,
        capitalizeWords: true);
    Console.WriteLine("Capitalized sentence with reversed words:");
    foreach (string word in words)
    {
        Console.Write(word);
        Console.Write(' ');
    }

    Console.ReadKey();
}
```

代码段 Ch14Ex04\Program.cs

(4) 运行应用程序，结果如图 14-9 所示。

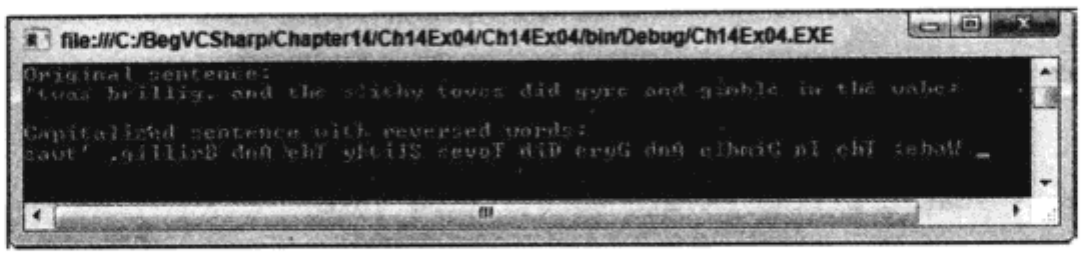


图　14-9

示例的说明

这个示例创建了一个实用类，它执行一些简单的字符串处理，再使用这个类修改一个字符串。类中的单个公共方法包含一个必选参数和 3 个可选参数：


可从 wrox.com 下载源代码

```
public static List<string> GetWords(
    string sentence,
    bool capitalizeWords = false,
    bool reverseOrder = false,
    bool reverseWords = false)
{
    ...
}
```

代码段 Ch14Ex04\WordProcessor.cs

这个方法返回一个 string 值的集合，每个 string 值都是初始输入的一个单词。根据指定的 3 个可选参数，可能会进行额外的转换：对字符串集合进行整体转换，或者仅转换某个单词。

这里没有深入探讨 WordProcessor 类的功能，读者可以自己研究它的代码，考虑一下如何改进这些代码，'twas 应改为'Twas 吗？如何进行这个修改？

调用这个方法时，只使用了两个可选参数，第三个参数(reverseOrder)使用其默认值 false：

```
words = WordProcessor.GetWords(
    sentence,
    reverseWords: true,
    capitalizeWords: true);
```

还要注意，所指定的两个参数的顺序与定义它们的顺序不同。最后要注意的是，处理带可选参数的方法时，使用 IntelliSense 会非常方便。输入这个示例的代码时，注意 GetWords()方法的工具提示，如图 14-10 所示(把鼠标指针停放在方法调用上，也会看到这个工具提示)。

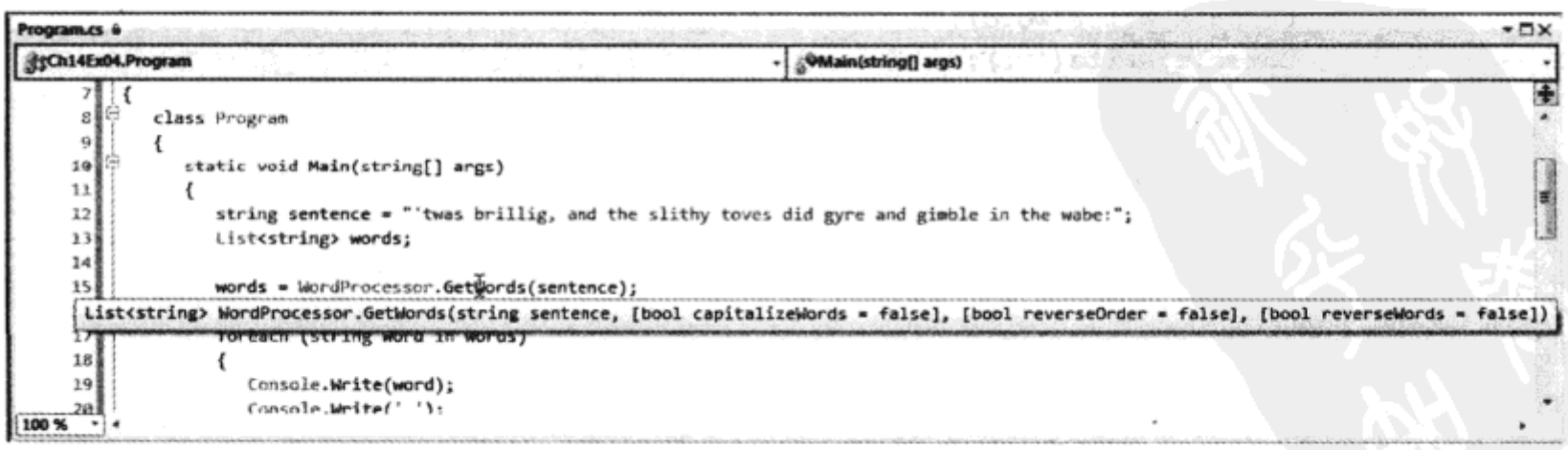


图　14-10

这是一个非常有用的工具提示，它不仅显示了可用参数的名称，还显示了可选参数的默认值，非常便于确定是否需要重写某个默认值。

14.5.3　命名参数和可选参数的规则

自从引入了命名参数和可选参数之后，人们对它们的反应不一。一些开发人员，尤其是使用 Microsoft Office 的开发人员，非常喜欢它们；但许多其他的开发人员认为这种修改对 C#语言而言是不必要的，觉得设计良好的用户界面应该不需要这种访问方式——至少在对 C#语言的这层改变上是不需要的。

我个人认为命名参数和可选参数有一些优点，但我担心对它们的过度使用会伤害代码。对于一些情形，例如上面提到的 Microsoft Office，它们肯定是有益的。另外，像上一个示例中的代码那样，定义了许多选项来控制方法的操作，这使代码更容易编写和使用。但在大多数情况下，没有合适的理由，最好不要使用命名参数和可选参数。也许应对方法调用进行充分的测试，看看在事先不知道方法应执行什么操作的情况下，是否能确定会得到什么结果。如果参数及其用法很明显(在编写好的代码中，应很明显)，就不需要使用命名参数或可选参数来修订代码。

14.6　扩展方法

扩展方法可以扩展类型的功能，但无需修改类型本身。甚至可以使用扩展方法扩展不能修改的类型，包括在.NET Framework 中定义的类型。例如，使用扩展方法甚至可以给 System.String 等基本类型添加功能。

为了扩展类型的功能，需要提供可以通过该类型的实例调用的方法。为此创建的方法称为扩展方法(extension method)，它可以带任意数量的参数，返回任意类型(包括 void)。要创建和使用扩展方法，必须：

(1) 创建一个非泛型静态类。

(2) 使用扩展方法的语法，给所创建的类添加扩展方法，作为静态方法(稍后介绍)。

(3) 确保使用扩展方法的代码用 using 语句导入了包含扩展方法类的名称空间。

(4) 通过扩展类型的一个实例调用扩展方法，与调用扩展类型的其他方法一样。

C#编译器在第(3)步和第(4)步之间完成了它的使命。IDE 会立即发现我们创建了一个扩展方法，并显示在 IntelliSense 中，如图 14-11 所示。

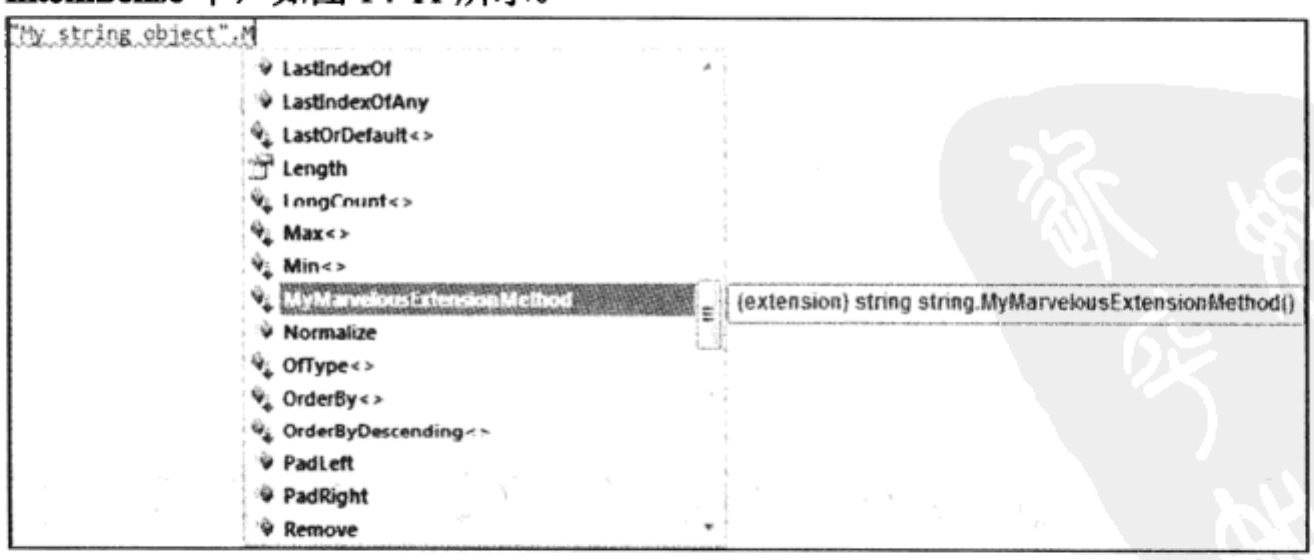


图　14-11

在图 14-11 中，可以通过 string 对象(这里仅是一个字面量字符串)使用扩展方法 MyMarvelousExtensionMethod()。这个方法用一个略微不同的方法图标来表示，该图标包含一个蓝色的向下箭头，这个方法不带其他参数，返回一个字符串。

为了定义扩展方法，应以与其他方法相同的方式定义一个方法，但该方法必须满足扩展方法的语法要求。这些要求如下：

- 方法必须是静态的。
- 方法必须包含一个参数，表示调用扩展方法的类型实例(这个参数在这里称为实例参数)。
- 实例参数必须是为方法定义的第一个参数。
- 除了 this 关键字之外，实例参数不能有其他修饰符。

扩展方法的语法如下：

```
public static class ExtensionClass
{
   public static <ReturnType> <ExtensionMethodName>
   (this <TypeToExtend> instance)
   {
      ...
   }
}
```

导入了包含静态类(其中包括此方法)的名称空间后(也就是使扩展方法变得可用)，就可以编写如下代码：

```
<TypeToExtend> myVar;
// myVar is initialized by code not shown here.
myVar. <ExtensionMethodName> ();
```

还可以在扩展方法中包含需要的其他参数，并使用其返回类型。

这个调用实际上与下面的调用相同，但语法更简单：

```
<TypeToExtend> myVar;
// myVar is initialized by code not shown here.
ExtensionClass. <ExtensionMethodName> (myVar);
```

另一个优点是，导入后，就可以通过 IntelliSense 查看匿名类型，这样能更容易地找到需要的功能。扩展方法可能分布在多个扩展类中，甚至分布在多个库中，但它们都会显示在扩展类型的成员列表中。

定义了可以用于某个类型的扩展方法后，还可以把它用于派生于这个类型的子类型。在本章前面的一个示例中，如果为 Animal 类定义了一个扩展方法，就可以在诸如 Cow 的对象上调用它。

还可以定义在特定接口上执行的扩展方法，接着就可以给实现了该接口的任意类型使用该扩展方法。

扩展方法为在应用程序中重用实用代码库提供了一种方式。它们还可以广泛用于本书后面介绍的 LINQ 中。为了更好地理解它，下面看一个示例。

试一试：定义和使用扩展方法

(1) 创建一个新的控制台应用程序 Ch14Ex05，把它保存在 C:\BegVCSharp\Chapter14 目录中。

(2) 在解决方案中添加一个新的类库项目 ExtensionLib。

(3) 从 ExtensionLib 中删除已有的 Class1.cs 类文件，在项目中添加 Ch14Ex04 中的类文件 WordProcessor.cs。

(4) 修改 WordProcessor.cs 中的代码，如下所示：


可从 wrox.com 下载源代码

```
namespace ExtensionLib
{
    public static class WordProcessor
    {
        public static List<string> GetWords(
            this string sentence,
            bool capitalizeWords = false,
            bool reverseOrder = false,
            bool reverseWords = false)
        {
            ...
        }
        ...
        public static string ToStringReversed(this object inputObject)
        {
            return ReverseWord(inputObject.ToString());
        }

        public static string AsSentence(this List<string> words)
        {
            StringBuilder sb = new StringBuilder();
            for (int wordIndex = 0; wordIndex < words.Count; wordIndex++)
            {
                sb.Append(words[wordIndex]);
                if (wordIndex != words.Count - 1)
                {
                    sb.Append(' ');
                }
            }
            return sb.ToString();
        }
    }
}
```

代码段 ExtensionLib\WordProcessor.cs

(5) 在 Ch14Ex05 项目中添加对 ExtensionLib 项目的引用。

(6) 修改 Program.cs 中的代码，如下所示：


可从 wrox.com 下载源代码

```
using ExtensionLib;

namespace Ch14Ex05
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter a string to convert:");
```

```
            string sourceString = Console.ReadLine();
            Console.WriteLine("String with title casing: {0}",
                sourceString.GetWords(capitalizeWords: true)
                    .AsSentence());
            Console.WriteLine("String backwards: {0}",
                sourceString.GetWords(reverseOrder: true,
                    reverseWords: true).AsSentence());
            Console.WriteLine("String length backwards: {0}",
                sourceString.Length.ToStringReversed());
            Console.ReadKey();
        }
    }
```

代码段 Ch14Ex05\Program.cs

(7) 运行应用程序。当出现提示时，键入一个字符串(至少 10 个字符长，最好包含多个单词)。结果如图 14-12 所示。

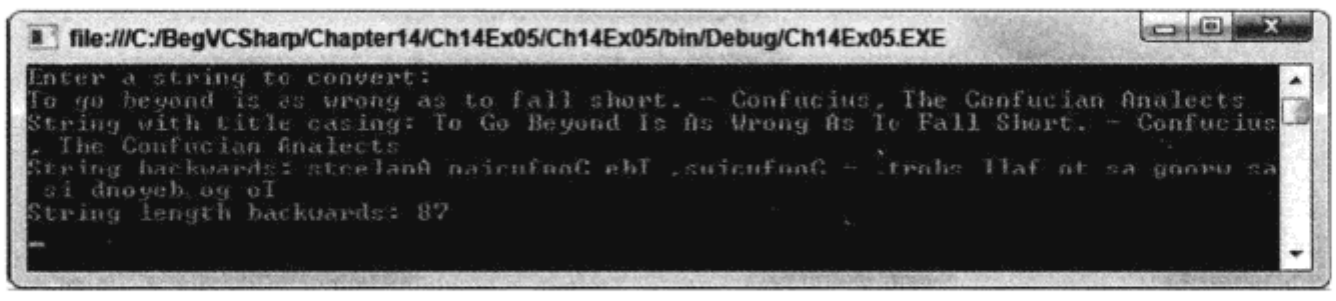


图 14-12

示例的说明

这个示例创建了一个类库，其中包含在一个简单的客户应用程序中使用的实用扩展方法。这个类库有一个静态类 WordProcessor 的扩展版本，WordProcessor 来自上一个包含扩展方法的示例，我们把包含这个类的名称空间 ExtensionLib 导入客户应用程序，以便使用这些扩展方法。

我们创建了 3 个扩展方法，如表 14-1 中所示。

表 14-1

方 法	用 法
GetWords()	操作字符串的灵活方法，如上一个示例所示。在这个示例中，这个方法改为扩展方法，返回一个 List<string>
ToStringReversed()	对于在对象上调用 ToString()所返回的字符串，使用 ReverseWord()使其中的字母逆序，并返回一个字符串
AsSentence()	“铺平”一个 List<string>对象，返回一个字符串，该字符串由它包含的单词组成

客户代码依次调用这些方法，以各种方式修改输入的字符串。前面定义的 GetWords()方法返回一个 List<string>，所以使用 AsSentence()把它的输出结果铺平为一个字符串，以便于使用。

扩展方法 ToStringReversed()是一个比较一般的扩展方法，它不需要 string 类型的实例参数，而是有一个 object 类型的实例参数。这表示这个扩展方法可以在任意对象上调用，显示在所使用的每个对象的 IntelliSense 中。在这个扩展方法中没有做太多的工作，因为不能对要使用的对象做太多的假定。可以使用 is 运算符或尝试类型转换，来确定实例参数的类型，并据此执行相应的操作，也可

以执行这个例子的操作，使用所有对象都支持的基本功能——ToString()方法：

```
public static string ToStringReversed(this object inputObject)
{
    return ReverseWord(inputObject.ToString());
}
```

这个方法只是在其实例参数上调用了 ToString()方法，用前面的 ReverseWord()方法使实例参数中的字母逆序。在示例客户程序中，从 int 变量上调用了 ToStringReversed()方法，得到该整数的一个字符串表示，其中的数字顺序被颠倒了。

可以在多个类型上使用的扩展方法非常有用。另外，还可以定义泛型扩展方法，它们可以把约束应用于类型，如第 12 章所述。

14.7　Lambda 表达式

Lambda 表达式是 C# 3.0 引入的一个结构，可用于简化 C#编程的某些方面，尤其是与 LINQ 合并的方面。Lambda 表达式一开始很难掌握，主要是因为其用法非常灵活。Lambda 表达式与其他 C#语言特性(如匿名方法)结合使用时尤其有用。由于本书后面才介绍 LINQ，因此匿名方法是介绍 Lambda 表达式的最佳切入点。下面首先概述一下匿名方法。

14.7.1　复习匿名方法

第 13 章学习了匿名方法，这是提供的内联(inline)方法，否则就需要使用委托类型的变量。给事件添加处理程序时，过程如下：

(1) 定义一个事件处理方法，其返回类型和参数匹配要订阅的事件需要的委托的返回类型和参数。

(2) 声明一个委托类型的变量，用于事件。

(3) 把委托变量初始化为委托类型的实例，该实例指向事件处理方法。

(4) 把委托变量添加到事件的订阅者列表中。

实际上，这个过程会比上述简单一些，因为一般不使用变量存储委托，只在订阅事件时使用委托的一个实例。

下面是第 13 章使用的代码：

```
Timer myTimer = new Timer(100);
myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
```

这段代码订阅了 Timer 对象的 Elapsed 事件。这个事件使用委托类型 ElapsedEventHandler，使用方法标识符 WriteChar 实例化该委托类型。结果是 Timer 对象引发 Elapsed 事件时，就调用方法 WriteChar()。传送给 WriteChar()的参数取决于由 ElapsedEventHandler 委托定义的参数类型和 Timer 中引发事件的代码传送的值。

实际上，如第 13 章所述，C#编译器可以通过方法组语法，用更少的代码获得相同的结果：

```
myTimer.Elapsed += WriteChar;
```

C#编译器知道 Elapsed 事件需要的委托类型，所以可以填充该类型。但是，在大多数情况下，最好不要这么做，因为这会使代码更难理解，也不清楚会发生什么。使用匿名方法时，该过程会减少为一步：

(1) 使用内联的匿名方法，该匿名方法的返回类型和参数匹配所订阅事件需要的委托的返回类型和参数。

用 delegate 关键字定义内联的匿名方法：

```
myTimer.Elapsed +=
   delegate(object source, ElapsedEventArgs e)
   {
      Console.WriteLine(
         "Event handler called after {0} milliseconds.",
         (source as Timer).Interval);
   };
```

这段代码可以正常工作，且使用了事件处理程序。主要区别是这里使用的匿名方法对于其余代码而言实际上是隐藏的。例如，不能在应用程序的其他地方重用这个事件处理程序。另外，为了更好地加以描述，这里使用的语法有点沉闷。delegate关键字总是会带来混淆，因为它有双重含义——匿名方法和定义委托类型都要使用它。

14.7.2　把 Lambda 表达式用于匿名方法

下面看看 Lambda 表达式。Lambda 表达式是简化匿名方法的语法的一种方式。实际上，Lambda 表达式还有其他用处，但为了简单起见，本节只介绍 Lambda 表达式的这个方面。使用 Lambda 表达式可以重写上一节最后的一段代码，如下所示：

```
myTimer.Elapsed += (source, e) => Console.WriteLine(
   "Event handler called after {0} milliseconds.",
   (source as Timer).Interval);
```

这段代码初看上去还好，只是有点让人摸不着头脑(除非很熟悉所谓的函数化编程语言，如 Lisp 或 Haskell)。但是，如果仔细观察，就会看出或至少推断出代码是如何工作的，它与所替代的匿名方法有什么关系。Lambda 表达式由 3 个部分组成：

- 放在括号中的参数列表(未类型化)
- =>运算符
- C#语句

使用本章前面“匿名类型”一节中介绍的逻辑，从上下文中推断出参数的类型。=>运算符只是把参数列表与表达式体分开。在调用 Lambda 表达式时，执行表达式体。

编译器会提取这个 Lambda 表达式，创建一个匿名方法，其工作方式与上一节中的匿名方法相同。其实，它会被编译为相同或相似的 CIL 代码。

为了说明 Lambda 表达式中的内容，下面举一个例子。

试一试：使用简单的 Lambda 表达式

(1) 创建一个新的控制台应用程序 Ch14Ex06，把它保存在 C:\BegVCSharp\Chapter14 目录中。

(2) 修改 Program.cs 中的代码，如下所示：


```
namespace Ch14Ex04
{
    delegate int TwoIntegerOperationDelegate(int paramA, int paramB);

    class Program
    {
        static void PerformOperations(TwoIntegerOperationDelegate del)
        {
            for (int paramAVal = 1; paramAVal < = 5; paramAVal++)
            {
                for (int paramBVal = 1; paramBVal < = 5; paramBVal++)
                {
                    int delegateCallResult = del(paramAVal, paramBVal);
                    Console.Write("f({0},{1})={2}",
                        paramAVal, paramBVal, delegateCallResult);
                    if (paramBVal != 5)
                    {
                        Console.Write(",");
                    }
                }
            Console.WriteLine();
            }
        }

        static void Main(string[] args)
        {
            Console.WriteLine("f(a, b) = a + b: ");
            PerformOperations((paramA, paramB) = > paramA + paramB);
            Console.WriteLine();
            Console.WriteLine("f(a, b) = a * b: ");
            PerformOperations((paramA, paramB) = > paramA * paramB);
            Console.WriteLine();
            Console.WriteLine("f(a, b) = (a - b) % b: ");
            PerformOperations((paramA, paramB) = > (paramA - paramB)
                % paramB);
            Console.ReadKey();
        }
    }
}
```

代码段 Ch14Ex06\Program.cs

(3) 运行应用程序，结果如图 14-13 所示。

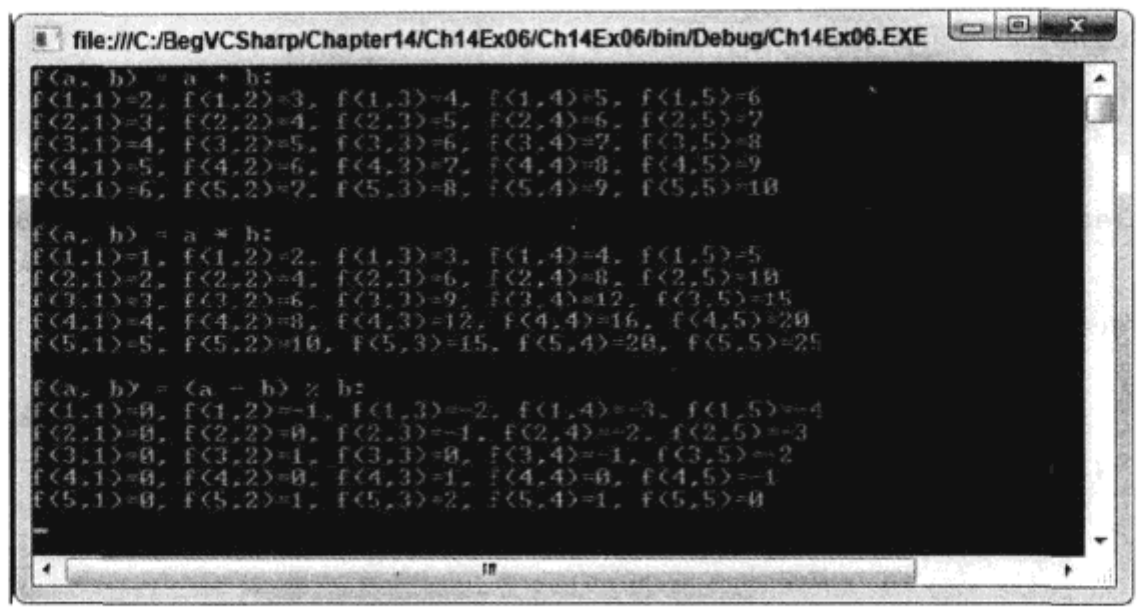


图　14-13

示例的说明

这个示例使用 Lambda 表达式生成函数，在两个输入参数上执行指定的处理，并返回结果。接着这些函数操作 25 对值，把结果输出到控制台上。

首先定义一个委托类型 TwoIntegerOperationDelegate，表示一个方法，该方法有两个 int 参数，返回一个 int 结果：

```
delegate int TwoIntegerOperationDelegate(int paramA, int paramB);
```

在以后定义 Lambda 表达式时使用这个委托类型。这些 Lambda 表达式编译为方法，其返回类型和参数匹配这个委托类型，如稍后所述。

接着添加方法 PerformOperations()，它带有一个 TwoIntegerOperationDelegate 类型的参数：

```
static void PerformOperations(TwoIntegerOperationDelegate del)
{
```

这个方法的含义是，可以给它传送一个委托实例(或者匿名方法，或者 Lambda 表达式，因为这些结构都会编译为委托实例)，该方法会用一组值调用委托实例所表示的方法：

```
for (int paramAVal = 1; paramAVal < = 5; paramAVal++)
{
   for (int paramBVal = 1; paramBVal < = 5; paramBVal++)
   {
      int delegateCallResult = del(paramAVal, paramBVal);
```

接着把参数和结果输出到控制台上：

```
            Console.Write("f({0},{1})={2}",
               paramAVal, paramBVal, delegateCallResult);
            if (paramBVal != 5)
            {
               Console.Write(",");
            }
         }
         Console.WriteLine();
      }
   }
```

在 Main()方法中，创建了 3 个 Lambda 表达式，使用它们依次调用 PerformOperations()，第一个调用如下所示：

```
Console.WriteLine("f(a, b) = a + b: ");
PerformOperations((paramA, paramB) = > paramA + paramB);
```

这里使用的 Lambda 表达式如下：

```
(paramA, paramB) = > paramA + paramB
```

这个 Lambda 表达式分为 3 部分：

(1) 参数定义部分。这里有两个参数 paramA 和 paramB。这些参数都是未类型化的，因此编译器可以根据上下文推断出它们的类型。在这个例子中，编译器可以确定，PerformOperations()方法调用需要一个 TwoIntegerOperationDelegate 类型的委托。这个委托类型有两个 int 参数，所以根据推断，paramA 和 paramB 都是 int 类型的变量。

(2) =>运算符。它把 Lambda 表达式的参数与表达式体分开。

(3) 表达式体。它指定了一个简单的操作：把 paramA 和 paramB 加起来。注意，不需要指定这是返回值。编译器知道要创建可以使用 TwoIntegerOperationDelegate 的方法，这个方法就必须有 int 返回类型。根据指定的操作，paramA + paramB 等于一个 int 类型的值，且没有提供额外的信息，所以编译器推断，这个表达式的结果就是方法的返回类型。

接着，就可以把使用这个 Lambda 表达式的代码扩展到下面使用匿名方法的代码中：

```
Console.WriteLine("f(a, b) = a + b:");
PerformOperations(delegate(int paramA, int paramB)
    {
        return paramA + paramB;
    });
```

其余代码以相同的方式使用两个不同的 Lambda 表达式来执行操作：

```
Console.WriteLine();
Console.WriteLine("f(a, b) = a * b: ");
PerformOperations((paramA, paramB) = > paramA * paramB);
Console.WriteLine();
Console.WriteLine("f(a, b) = (a - b) % b: ");
PerformOperations((paramA, paramB) = > (paramA - paramB)
    % paramB);
Console.ReadKey();
```

最后一个 Lambda 表达式涉及较多的计算，但并不比其他 Lambda 表达式更复杂。Lambda 表达式的语法允许执行更复杂的操作，如稍后所述。

14.7.3　Lambda 表达式的参数

在前面的代码中，Lambda 表达式使用类型推理功能确定所传送的参数类型。实际上这不是必须的，也可以定义类型。例如，可以使用下面的 Lambda 表达式：

```
(int paramA, int paramB) = > paramA + paramB
```

其优点是代码更容易理解，但不够简明灵活。在前面委托类型的示例中，可以通过隐式类型化

的 Lambda 表达式来使用其他数字类型，例如，long 变量。

注意，不能在同一个 Lambda 表达式中同时使用隐式和显式的参数类型。下面的 Lambda 表达式就不会编译，因为 paramA 是显式类型化的，而 paramB 是隐式类型化的：

```
(int paramA, paramB) = > paramA + paramB
```

Lambda 表达式的参数列表总是包含一个用逗号分隔的列表，其中的参数要么都是显式类型化的，要么都是隐式类型化的。如果只有一个参数，就可以省略括号；否则就需要在参数列表上加上括号，如前面所示。例如，下面的 Lambda 表达式只有一个参数，且是隐式类型化的：

```
param1 = > param1 * param1
```

还可以定义没有参数的 Lambda 表达式，这使用空括号来表示：

```
() = > Math.PI
```

当委托不需要参数，但需要返回一个 double 值时，就可以使用这个 Lambda 表达式。

14.7.4　Lambda 表达式的语句体

在前面的所有代码中，Lambda 表达式的语句体都只使用了一个表达式。并说明了这个表达式如何解释为 Lambda 表达式的返回值，例如，如何给返回类型为 int 的委托使用表达式 paramA+paramB 作为 Lambda 表达式的语句体(假定 paramA 和 paramB 隐式或显式类型化为 int 值，如示例代码所示)。

前面的一个示例说明了对于语句体中使用的代码而言，返回类型为 void 的委托的要求并不高：

```
myTimer.Elapsed += (source, e) = > Console.WriteLine(
  "Event handler called after {0} milliseconds.",
  (source as Timer).Interval);
```

上面的语句不返回任何值，所以它只是执行，其返回值不在任何地方使用。

Lambda 表达式可以看作匿名方法语法的扩展，所以还可以在 Lambda 表达式的语句体中包含多个语句。为此，只需把一个代码块放在花括号中，类似于 C#中提供多行代码的其他情况：

```
(param1, param2) = >
{
   // Multiple statements ahoy!
}
```

如果使用 Lambda 表达式和返回类型不是 void 的委托类型，就必须用 return 关键字返回一个值，这与其他方法一样：

```
(param1, param2) = >
{
   // Multiple statements ahoy!
   return returnValue ;
}
```

例如，可以把前面示例中的如下代码：

```
PerformOperations((paramA, paramB) = > paramA + paramB);
```

改写为：

```
PerformOperations(delegate(int paramA, int paramB)
    {
        return paramA + paramB;
    });
```

另外，也可以把代码改写为：

```
PerformOperations((paramA, paramB) = >
    {
        return paramA + paramB;
    });
```

这更像是原来的代码，因为它包含 paramA 和 paramB 参数的隐式类型化。

在大多数情况下，使用单一的表达式时，大都使用 Lambda 表达式，它们肯定是最简洁的。说实话，如果需要多个语句，则定义一个非匿名方法来替代 Lambda 表达式比较好，这也会使代码更便于重用。

14.7.5　Lambda 表达式用作委托和表达式树

前面提到了 Lambda 表达式和匿名方法的一些区别：匿名方法比较灵活，例如，隐式类型化的参数。目前，应注意另一个重要区别，但在学习本书后面的 LINQ 之前，这个区别并不是很明显。

可以用两种方式来解释 Lambda 表达式。第一，如本章所述，Lambda 表达式是一个委托。即可以把 Lambda 表达式赋予一个委托类型的变量，如前面的示例所示。

一般可以把拥有至多 8 个参数的 Lambda 表达式表示为如下泛型类型，它们都在 System 命名空间中定义：

- Action 表示的 Lambda 表达式不带参数，返回类型是 void
- Action<>表示的 Lambda 表达式有至多 8 个参数，返回类型是 void
- Func<>表示的 Lambda 表达式有至多 8 个参数，返回类型不是 void

Action<>有至多 8 个泛型类型的参数，分别用于 Lambda 表达式的 8 个参数，Func<>有至多9个泛型类型的参数，分别用于 Lambda 表达式的 8 个参数和返回类型。在 Func<>中，返回类型总是在列表的最后。

例如，下面的 Lambda 表达式：

```
(int paramA, int paramB) = > paramA + paramB
```

可以表示为 Func<int,int,int>类型的委托，因为它有两个 int 参数，返回类型是 int。

第二，可以把 Lambda 表达式解释为表达式树。表达式树是 Lambda 表达式的抽象表示，但不能直接执行。可以使用表达式树以编程方式分析 Lambda 表达式，执行操作，以响应 Lambda 表达式。

显然这是一个复杂的主题，但表达式树对本书后面介绍的 LINQ 功能至关重要。下面给出一个具体的例子。LINQ 架构包含一个泛型类 Expression<>，可用于封装 Lambda 表达式。使用这个类的一种方式是提取用 C#编写的 Lambda 表达式，把它转换为相应的 SQL 脚本，以便在数据库中直接执行。

目前并不需要了解太多的内容，在本书后面遇到这个功能时，能更好地理解其过程，因为现在

我们已经理解了 C#提供的重要概念。

14.7.6　Lambda 表达式和集合

学习了 Func<>泛型委托之后，就可以理解 System.Linq 名称空间为数组类型提供的一些扩展方法了(在编码的不同地方，可以在弹出 IntelliSense 时看到它们)。例如，有一个扩展方法 Aggregate() 定义了 3 个重载版本，如下所示：

```
public static TSource Aggregate < TSource > (
    this IEnumerable < TSource > source,
    Func < TSource, TSource, TSource > func);

public static TAccumulate Aggregate < TSource, TAccumulate > (
    this IEnumerable < TSource > source,
    TAccumulate seed,
    Func < TAccumulate, TSource, TAccumulate > func);

public static TResult Aggregate < TSource, TAccumulate,
                                  TResult > (
    this IEnumerable < TSource > source,
    TAccumulate seed,
    Func < TAccumulate, TSource, TAccumulate > func,
    Func < TAccumulate, TResult > resultSelector);
```

与前面的扩展方法一样，这段代码初看上去非常深奥，但如果分解它们，就很容易理解其工作过程。这个函数的 IntelliSense 告诉用户它会执行如下工作：

```
Applies an accumulator function over a sequence.
```

这表示要把一个累加器函数(可以以 Lambda 表达式的形式提供)应用于集合中从开始到结束的每对元素上，并把计算的结果作为下一个计算操作的输入。

在 3 个重载版本中，最简单的版本只有一个泛型类型，这可以从实例参数的类型推理出来。例如，在下面的代码中，泛型类型是 int(累加器函数现在是空白的)：

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate(...);
```

这等价于：

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate <int> (...);
```

这里需要的 Lambda 表达式可以从扩展方法中推断出来。在这段代码中，类型 TSource 是int，所以必须为委托 Func<int,int,int>提供一个 Lambda 表达式。例如，可以使用前面的 Lambda 表达式：

```
int[] myIntArray = {2, 6, 3};
int result = myIntArray.Aggregate((paramA, paramB) = > paramA + paramB);
```

这个调用会使 Lambda 表达式调用两次，一次使用的参数是 paramA=2，paramB=6，另一次使用的参数是 paramA=8(第一次计算的结果)，paramB=3。最后赋予变量 result 的结果是 int 值 11，即数组中所有元素的总和。

扩展方法 Aggregate()的其他两个重载版本是类似的，但可以执行略微复杂的计算，如下面的简短示例所示。

试一试：使用 Lambda 表达式和集合

(1) 创建一个新的控制台应用程序 Ch14Ex07，把它保存在 C:\BegVCSharp\Chapter14 目录中。

(2) 修改 Program.cs 中的代码，如下所示：


可从 wrox.com 下载源代码

```
static void Main(string[] args)
{
    string[] curries = { "pathia", "jalfreze","korma"};
    Console.WriteLine(curries.Aggregate(
        (a, b) = > a +" "+ b));
    Console.WriteLine(curries.Aggregate < string, int > (
        0,
        (a, b) = > a + b.Length));
    Console.WriteLine(curries.Aggregate < string, string, string > (
        "Some curries: ",
        (a, b) = > a + " "+ b,
        a = > a));
    Console.WriteLine(curries.Aggregate < string, string, int > (
        "Some curries: ",
        (a, b) = > a + " "+ b,
        a = > a.Length));
    Console.ReadKey();
}
```

代码段 Ch14Ex07\Program.cs

(3) 运行应用程序，结果如图 14-14 所示。

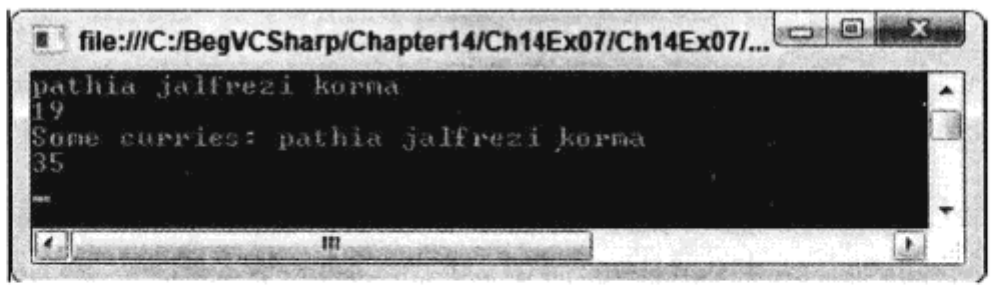


图　14-14

示例的说明

这个示例把包含 3 个元素的字符串数组作为源数据，试验了扩展方法 Aggregate()的每个重载版本。

首先执行一个简单的连接操作：

```
Console.WriteLine(curries.Aggregate(
    (a, b) = > a + " " + b));
```

第一对元素数用简单的语法串联成一个字符串。这不是连接字符串的最佳方式，最好使用 string.Concat()或 string.Format()优化性能，但这里使用它提供了一种非常简单的方式，来说明发生了什么。第一个串联操作之后，结果传送回 Lambda 表达式和数组中的第 3 个元素，其方式与前面要

计算总和的 int 值相同。结果是串联整个数组，并用空格分隔各个项。

接着，使用Aggregate()函数的第二个重载版本，它有两个泛型类型的参数TSource 和 TAccumulate。在这个示例中，Lambda 表达式的形式必须是 Func<TAccumulate, TSource, TAccumulate>。另外，必须指定 TAccumulate 类型的种子值，这个种子值在 Lambda 表达式的第一个调用和第一个数组元素中使用。后续的调用从前面的调用中把累加器的结果提取到表达式中。代码如下：

```
Console.WriteLine(curries.Aggregate < string, int > (
   0,
   (a, b) = > a + b.Length));
```

累加器(以及返回值)的类型是 int。累加器的值最初设置为种子值 0，在对 Lambda 表达式的每次调用中，都把该值累加到数组元素的长度上。最后的结果是数组中每个元素的总长度。

之后使用 Aggregate()函数的最后一个重载版本，它带有 3 个泛型类型的参数，与其他重载版本的唯一区别是，其返回类型可以与数组元素和累加值的类型都不同。首先，这个重载版本把字符串元素与种子字符串联在一起：

```
Console.WriteLine(curries.Aggregate < string, string, string > (
   "Some curries:",
   (a, b) = > a + " " + b,
   a = > a));
```

即使累加值只是复制到结果中，也必须指定这个方法的最后一个参数 resultSelector(如本例所示)。这个参数是一个 Func<TAccumulate, TResult >类型的 Lambda 表达式。

在最后一段代码中，再次使用了 Aggregate()的这个版本，但这次使用 int 类型的返回值。其中，给 resultSelector 提供一个 Lambda 表达式，返回累加字符串的长度：

```
Console.WriteLine(curries.Aggregate < string, string, int > (
   "Some curries:",
   (a, b) = > a + " " + b,
   a = > a.Length));
```

这个示例没有什么有趣的地方，但演示了如何使用更复杂的扩展方法，其中涉及泛型类型的参数、集合和看似复杂的语法。本书后面还要讨论它们。

14.8　小结

本章介绍了 VS2010 和 Visual C# Express 2010 中使用的 C# 4 语言的新特性，这些特性简化了实现常用或高级功能所需的某些编码。

本章的主要内容如下：

- 如何使用对象和集合初始化器，在一个步骤中完成对象和集合的实例化和初始化。
- IDE 和 C#编译器如何从上下文中推断类型，如何使用 var 关键字把类型推理功能用于任意变量类型。
- 如何创建和使用匿名类型，它们合并了前面介绍的初始化器和类型推理主题。
- 如何在变量上使用动态查找功能，该变量仅在运行期间访问，以用于成员。

- 如何使用命名参数和可选参数，以灵活的方式调用方法。
- 如何创建扩展方法(扩展方法可以在其他类型的实例上调用，且无需在这些类型的定义中添加代码)，如何使用这个技术提供实用方法库。
- 如何使用 Lambda 表达式给委托实例提供匿名方法，Lambda 方法的扩展语法如何实现其他功能。

本章介绍的大多数 C#特性都已添加，以满足.NET Framework 的 LINQ 新功能的需要。本章中代码的许多子主题以后会进一步阐明。如前所述，我们学习了一些非常强大的技术，可以用于快速提升 C#编程技巧。

现在，我们已经完成了 C# 4 语言的全部介绍，但这并不意味着，这些内容囊括了.NET Framework 编程的所有内容。C#语言提供了编写.NET 应用程序所需的所有工具，但.NET Framework 中的类为您提供了建立应用程序的原材料。本书从现在开始，就要深入探讨这些类，学习如何使用它们执行各种任务了。第 15 章不再使用前面大量使用的控制台应用程序，而开始使用 Windows Forms 提供的丰富功能创建图形化的用户界面。此时应注意尽管所创建的应用程序类型不同，但底层的规则是相同的，本书第 I 部分介绍的技巧也可以在后面的章节中使用。

14.9　练习

(1) 为什么不能把对象初始化器用于下面的类？修改这个类，使之能使用对象初始化器。之后给出一个示例代码，仅通过一个步骤实例化和初始化这个类：

```
public class Giraffe
{
   public Giraffe(double neckLength, string name)
   {
      NeckLength = neckLength;
      Name = name;
   }
   public double NeckLength {get; set;}
   public string Name {get; set;}
}
```

(2) 判断正误：如果声明一个 var 类型的变量，就可以使用它存储任意对象类型。

(3) 使用匿名类型时，如何比较两个实例，确定它们是否包含相同的数据？

(4) 更正下述扩展方法的代码，其中包含一个错误：

```
public string ToAcronym(this string inputString)
{
   inputString = inputString.Trim();
   if (inputString == "")
   {
        return "";
   }
   string[] inputStringAsArray = inputString.Split(' ');
   StringBuilder sb = new StringBuilder();
   for (int i = 0; i < inputStringAsArray.Length; i++)
   {
        if (inputStringAsArray[i].Length > 0)
        {
          sb.AppendFormat("{0}", inputStringAsArray[i].Substring(0, 1).ToUpper());
        }
```

```
        }
        return sb.ToString();
    }
```

(5) 如何确保题(4)中的扩展方法可用于客户代码?

(6) 把题(4)中的 ToAcronym 方法改为一行代码。该代码应确保单词之间包含多个空格的字符串不出错。提示：需要使用?:三元运算符、string.Aggregate<string, string>扩展方法和一个 Lambda 表达式。

附录 A 给出了练习答案。

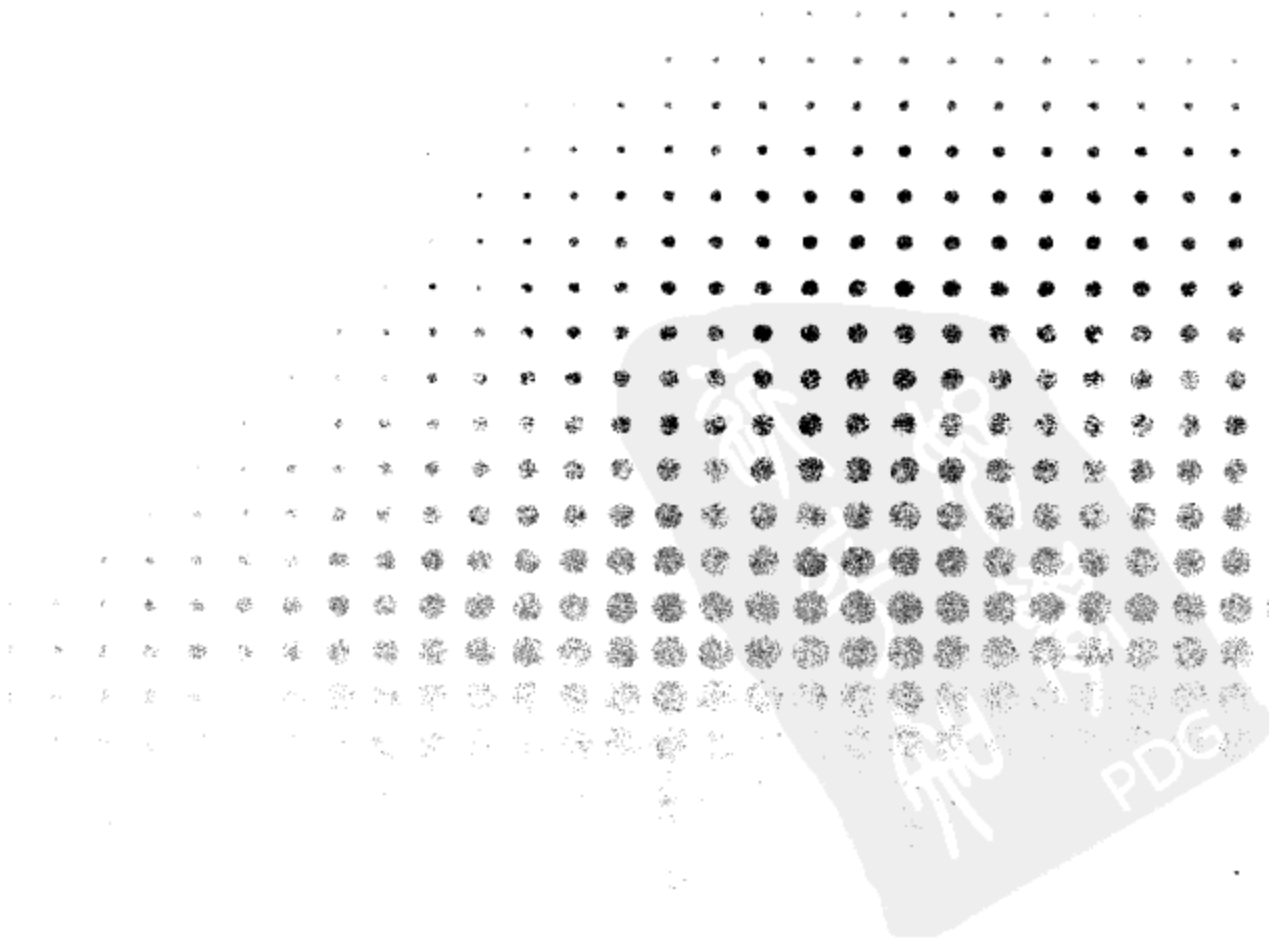
14.10　本章要点

主　题	重 要 概 念
初始化器	可以使用初始化器在创建对象或集合的同时初始化它们。这两种初始化器都包括一个放在花括号中的代码块。对象初始化器可以提供一个逗号分隔的属性名/值对列表，来设置属性值。集合初始化器仅需要逗号分隔的值列表。使用对象初始化器时，还可以使用非默认的构造函数
类型推理	声明变量时，使用 var 关键字允许忽略变量的类型。但只有类型可以在编译期间确定时才会如此。使用 var 没有违反 C#的强类型化规则，因为用 var 声明的变量只能有一种类型
匿名类型	对于用于结构数据存储的许多简单类型，定义类型是不必要的。而可以使用匿名类型，其成员根据用途来推断。使用对象初始化器语法来定义匿名类型，每个设置的属性都定义为只读属性
动态查找	使用 dynamic 关键字定义 dynamic 类型的变量，可以存储任意值。接着就可以使用一般的属性或方法语法来访问被包含的值的成员，这些成员仅在运行期间检查。如果在运行期间，尝试访问一个不存在的成员，就会抛出一个异常。这种动态的类型化显著简化了访问非.NET 类型或类型信息不能在编译期间获得的.NET 类型的语法。但是，在使用时动态类型要谨慎，因为无法在编译期间检查代码。实现 IDynamicMetaObjectProvider 接口，可以控制动态查找的行为
可选的方法参数	我们常常可以定义带许多参数的方法，但其中的许多参数都很少使用。可以提供多个方法重载，而无不是强制客户代码为很少使用的参数提供值。另外，也可以把这些参数定义为可选参数(并为未指定值的参数提供默认值)。调用方法的客户代码就可以仅指定需要的参数
命名的方法参数	客户代码可以根据位置或名称(或者根据位置和名称，其中位置参数放在前面)来指定方法的参数。命名的参数可以按任意顺序指定。这尤其适用于和可选参数一起使用的场合
扩展方法	可以为任意已有的类型定义扩展方法，而无需修改类型定义。例如，这包括扩展系统定义的类型，如 string。扩展方法定义为非泛型静态类的静态方法。实例方法的第一个参数使用 this 关键字定义，是针对其调用方法的实例值。定义完后，扩展方法就可以在任意代码中调用，但这些代码必须引用包含定义该方法的类的名称空间。扩展方法可以在方法定义或任意派生类型使用的类型实例上调用，所以可以为类型系列定义通用的扩展方法。创建通用的扩展方法的另一种方式是创建可以通过特定接口使用的扩展方法
Lambda 表达式	Lambda 表达式实际上是定义匿名方法的一种快捷方式，且有额外的功能，例如隐式的类型化。定义 Lambda 表达式时，需要使用逗号分隔的参数列表(如果没有参数，就使用空括号)、=>运算符和一个表达式。该表达式可以是放在花括号中的代码块。Lambda 表达式至多可以有 8 个参数和一个可选的返回类型，Lambda 表达式可以用 Action、Action<>和 Func<>委托类型来表示。许多可用于集合的 LINQ 扩展方法都使用 Lambda 表达式参数

第 II 部分

Windows 编 程

- 第 15 章　Windows 编程基础
- 第 16 章　Windows 窗体的高级功能
- 第 17 章　部署 Windows 应用程序



第15章

Windows 编程基础

本章内容：

- Windows 窗体设计器
- 向用户显示信息的控件，如 Label 和 LinkLabel 控件
- 触发事件的控件，如 Button 控件
- 允许应用程序的用户输入文本的控件，如 TextBox 控件
- 允许告诉用户应用程序的当前状态、让用户修改状态的控件，如 RadioButton 和 CheckButton 控件
- 允许显示信息列表的控件，如 ListBox 和 ListView 控件
- 允许把其他控件组合在一起的控件，如 TabControl 和 GroupBox 控件

近 10 年来，Visual Basic 允许程序员使用工具，通过直观的窗体设计器创建十分复杂的用户界面，其编程语言的易学易用，为快速开发应用程序(rapid application development，RAD)提供了尽可能好的环境，所以赢得了广泛的好评。Visual Basic 等 RAD 工具的一个优点是提供了许多预制控件，开发人员可以使用它们快速建立应用程序的用户界面。

开发大多数 Visual Basic Windows 应用程序的核心是窗体设计器(Forms Designer)。创建用户界面时，把控件从工具箱拖放到窗体上，把它们放在应用程序运行时需要的地方，再双击该控件，添加控件的处理程序。Microsoft 提供的控件和可以按合理价格购得的定制控件，为程序员提供了空前巨大的、已进行了全面测试的重用代码池，仅通过鼠标单击就可以使用它们。通过 Visual Studio，这种应用程序开发模式现在也可以用于 C#开发人员。

本章将使用 Windows 窗体，利用 Visual Studio 附带的许多控件。这些控件拥有各种功能，通过 Visual Studio 的设计功能，开发用户界面、处理用户的交互将非常简单、有趣。在本书全面介绍 Visual Studio 中的控件是不可能的，所以这里只介绍最常用的控件，包括标签、文本框、列表视图、选项卡控件等。

15.1 控件

在使用 Windows 窗体时，就是在使用 System.Windows.Forms 名称空间。这个名称空间使用 using

指令包含在存储 Form 类的一个文件中。.NET 中的大多数控件都派生于 System.Windows.Forms.Control 类。这个类定义了控件的基本功能，这就是控件中的许多属性和事件都相同的原因。许多类本身就是其他控件的基类，图 15-1 中的 Label 和 TextBoxBase 类就是这样。

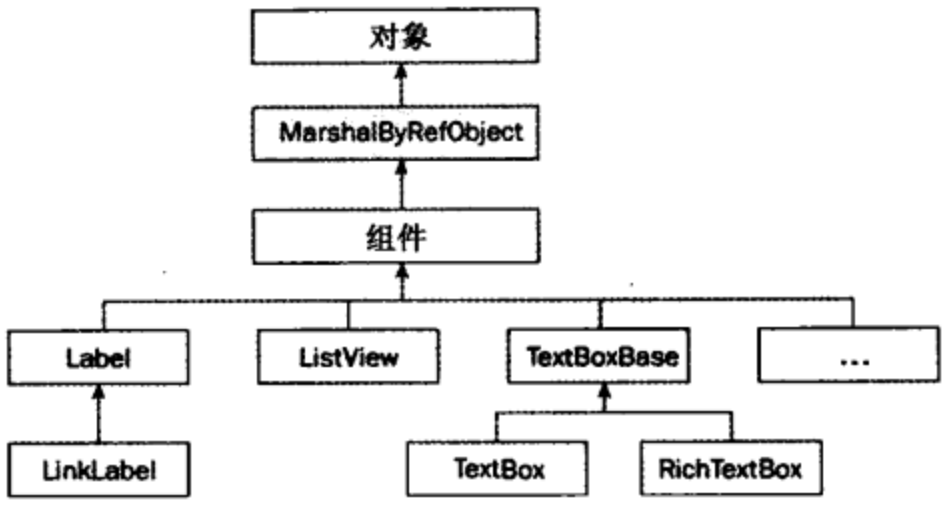


图 15-1

15.1.1 属性

所有控件都有许多属性，用于处理控件的操作。大多数控件的基类都是 System.Windows.Forms.Control，它有许多属性，其他控件要么直接继承了这些属性，要么重写它们以便提供某些定制的操作。

表 15-1 列出了 Control 类最常见的一些属性。这些属性在本章介绍的大多数控件中都有，所以后面将不再详细解释它们，除非属性的操作对于某个控件来说进行了改变。注意这个表并不完整，如果要查看该类的所有属性，请参阅.NET Framework SDK 文档说明。

表 15-1

属 性	说 明
Anchor	指定当控件的容器大小发生变化时，该控件如何响应。参见下一节来了解对这个属性的详细解释
BackColor	控件的背景色
Bottom	指定控件底部距窗口顶部的距离。这与指定控件的高度不同
Dock	使控件停靠在容器的边界上。参见下面对这个属性的详细解释
Enabled	把 Enabled 设置为 true 通常表示该控件可以接收用户的输入。把 Enabled 设置为 false 通常表示不能接收用户的输入
ForeColor	控件的前景色
Height	控件底部到顶部的距离
Left	控件的左边界距其容器左边界的距离
Name	控件的名称。这个名称可以在代码中用于引用该控件
Parent	控件的父控件
Right	控件的右边界距其容器左边界的距离
TabIndex	控件在容器中的标签顺序号
TabStop	指定是否可以用 Tab 键访问控件

(续表)

属　性	说　明
Tag	这个值通常不由控件本身使用，而是在控件中存储该控件的信息。当通过 Windows 窗体设计器给这个属性赋值时，就只能给它赋一个字符串值
Text	保存与该控件相关的文本
Top	控件顶部距其容器顶部的距离
Visible	指定控件是否在运行期间可见
Width	控件的宽度

15.1.2　控件的定位、停靠和对齐

在 Visual Studio 2005 中，窗体设计器默认情况下不再使用栅格状的界面供设置控件布局，而改用一个清洁的界面，并且使用捕捉线来定位控件，选择 Tools 菜单上的 Options 选项，就可以在两种设计样式之间切换。在树型视图左侧中选择 Windows Forms Designer 节点，设置 Layout Mode。这完全是一个个人喜好的问题，但在下面的示例中，将使用默认的设计样式。

试一试：使用捕捉线

按照下面的步骤使用 Windows 窗体设计器中的捕捉线：

(1) 创建一个 Windows 窗体应用程序，命名为 SnapLines。

(2) 把一个按钮控件从 Toolbox 拖放到窗体的中间位置。

(3) 把窗体向上拖动到窗体的左上角。注意在接近窗体的边缘时，会从窗体的左边缘和上边缘显示两条线，控件会被固定在该位置上。可以移动控件，使其超过捕捉线的范围，或者就把控件放在这个位置上。

(4) 把按钮移回到窗体的中心，把另一个按钮从 Toolbox 拖放到窗体上。把它移动到第一个按钮的下面，注意在这个过程中又会出现捕捉线。这些捕捉线可以把控件排列整齐，使控件位于相同的垂直或水平位置上。如果把新按钮向上移近已有的按钮，就会出现另一条捕捉线，允许用预设的距离放置按钮。

(5) 重新设置 button1 的大小，使它比另一个按钮宽，然后重新设置 button2 的大小，注意当 button2 的宽度与 button1 相同时，就会出现捕捉线，以便把控件的宽度设置为相同的值。

(6) 现在在按钮的下面给窗体添加一个 TextBox，把其 Text 属性改为 Hello World!。

(7) 在窗体上添加一个 Label，把它移动到 TextBox 的左边。注意在移动控件时，会出现两条捕捉线，捕捉 TextBox 的顶部和底部，在这两条捕捉线之间，还会出现第三条捕捉线，在把 Label 放在窗体上时，它可以使 TextBox 的文本和 Label 有相同的高度。

15.1.3　Anchor 和 Dock 属性

在设计窗体时，这两个属性特别有用。如果用户认为改变窗口的大小并非小事，应确保窗口看起来不显得很乱；在以前只有编写许多代码行才能达到这个目的。许多程序解决这个问题时，都是禁止给窗口重新设置大小，这显然是解决问题最简单的方法，但不是最好的方法。.NET 引入了 Anchor 和 Dock 属性，就是为了在不编写任何代码的情况下解决这个问题。

Anchor 属性指定在用户重新设置窗口的大小时控件该如何响应。可以指定如果控件重新设置了大小，就根据控件的边界合理地锁定它，或者其大小不变，但根据窗口的边界来锚定它的位置。

Dock 属性指定控件应停靠在容器的边框上。如果用户重新设置了窗口的大小，该控件将继续停放在窗口的边框上。例如，如果指定控件停靠在容器的底部边界上，则无论窗口的大小如何改变，该控件都将改变大小，或移动其位置，确保总是位于屏幕的底部。

本章后面将详细介绍 Anchor 属性。

15.1.4 事件

第 13 章介绍了事件的含义及其用法。本节介绍事件的特定类型，即 Windows 窗体控件生成的控件。这些事件通常与用户的操作相关。例如，在用户单击按钮时，该按钮就会生成一个事件，说明发生了什么。处理事件就是程序员为该按钮提供某些功能的方式。

Control 类定义了本章所用控件的一些比较常见的事件。表 15-2 描述了许多这类事件。这个表仅列出了最常见的事件；如果需要查看完整的列表，请参阅.NET Framework SDK 文档说明。

表 15-2

事 件	说 明
Click	在单击控件时引发。在某些情况下，这个事件也会在用户按下回车键时引发
DoubleClick	在双击控件时引发。处理某些控件上的 Click 事件，如 Button 控件，表示永远不会调用 DoubleClick 事件
DragDrop	在完成拖放操作时引发。换言之，当一个对象被拖到控件上，然后用户释放鼠标按钮后，引发该事件
DragEnter	在被拖动的对象进入控件的边界时引发
DragLeave	在被拖动的对象移出控件的边界时引发
DragOver	在被拖动的对象放在控件上时引发
KeyDown	当控件有焦点时，按下一个键时引发该事件，这个事件总是在 KeyPress 和 KeyUp 之前引发
KeyPress	当控件有焦点时，按下一个键时发生该事件，这个事件总是在 KeyDown 之后、KeyUp 之前引发。KeyDown 和 KeyPress 的区别是 KeyDown 传送被按下的键的键盘码，而 KeyPress 传送被按下的键的 char 值
KeyUp	当控件有焦点时，释放一个键时发生该事件，这个事件总是在 KeyDown 和 KeyPress 之后引发
GotFocus	在控件接收焦点时引发。不要用这个事件执行控件的有效性验证，而应使用 Validating 和 Validated
LostFocus	在控件失去焦点时引发。不要用这个事件执行控件的有效性验证，而应使用 Validating 和 Validated
MouseDown	在鼠标指针指向一个控件，且鼠标按钮被按下时引发。这与 Click 事件不同，因为在按钮被按下之后，且未被释放之前引发 MouseDown
MouseMove	在鼠标滑过控件时引发

(续表)

事　件	说　明
MouseUp	在鼠标指针位于控件上，且鼠标按钮被释放时引发
Paint	绘制控件时引发
Validated	当控件的 CausesValidation 属性设置为 true，且该控件获得焦点时，引发该事件。它在 Validating 事件之后发生，表示验证已经完成
Validating	当控件的 CausesValidation 属性设置为 true，且该控件获得焦点时，引发该事件。注意，被验证的控件是正在失去焦点的控件，而不是正在获得焦点的控件

本章后面的示例将介绍上表中的许多事件。所有的示例都使用相同的格式，即首先创建窗体的可视化外观，选择并定位控件，再添加事件处理程序，事件处理程序包含了示例的主要工作代码。

有 3 种处理事件的基本方式。第一种是双击控件，进入控件默认事件的处理程序，这个事件因控件而异。如果该事件就是我们需要的事件，就可以开始编写代码。如果需要的事件与默认事件不同，有两种方法来处理这种情况。

一种方法是使用 Properties 窗口中的 Events 列表，单击如图 15-2 所示的闪电图标按钮，就会显示 Events 列表。

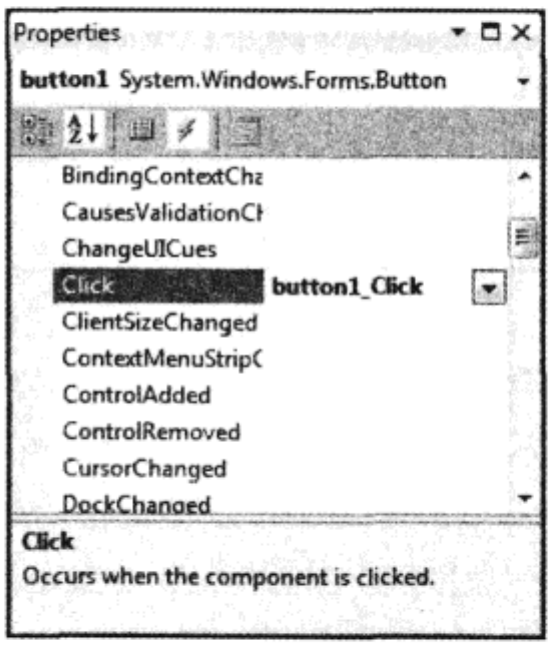


图　15-2

要给事件添加处理程序，只需在 Events 列表中双击该事件，就会生成给控件订阅该事件的代码，以及处理该事件的方法签名。另外，还可以在 Events 列表中该事件的旁边，为处理该事件的方法输入一个名称。按下回车键，就会用我们输入的名称生成一个事件处理程序。

另一个选项是自己添加订阅该事件的代码。在键入订阅该事件所需的代码时，VS 会检测到我们做的工作，并在代码中添加方法签名，就好像在窗体设计器中一样。

注意这两种方式都需要两步：订阅事件和处理方法的正确签名。如果双击控件，给要处理的事件编辑默认事件的方法签名，以处理另一个事件，就会失败，因为还需要修改 InitializeComponent() 中的事件订阅代码，所以这种方法并不是处理特定事件的快捷方式。

下面开始讨论控件本身，首先讨论 Windows 应用程序中最常用的一个控件，即 Button 控件。

15.2 Button 控件

在考虑按钮时，可能会把它想像为一个矩形按钮，单击该按钮，就可以执行某项任务。但.NET Framework 提供了一个派生于 Control 的类 System.Windows.Forms.ButtonBase，它实现了 Button 控件所需的基本功能，所以程序员可以从这个类中派生，创建定制的 Button 控件。

System.Windows.Forms 名称空间提供了 3 个派生于 ButtonBase 的控件，即 Button、CheckBox 和 RadioButton。本节主要讨论 Button 控件(这是标准的矩形按钮)，后面再介绍另外两个按钮。

Button 控件存在于几乎所有的 Windows 对话框中。按钮主要用于执行 3 类任务：

- 用某种状态关闭对话框(如 OK 和 Cancel 按钮)。
- 给对话框上输入的数据执行操作(例如，输入一些搜索条件后，单击 Search)。
- 打开另一个对话框或应用程序(如 Help 按钮)。

对 Button 控件的处理是非常简单的。通常是在窗体上添加控件，再双击它，给Click 事件添加代码，这对于大多数应用程序来说就足够了。

15.2.1 Button 控件的属性

下面介绍该控件的常用属性，了解该如何操作它。表 15-3 列出了 Button 类最常用的属性，但从技术上讲，它们都是在 ButtonBase 基类中定义的。这里只解释最常用的属性。完整的列表请参阅.NET Framework SDK 文档说明。

表 15-3

属 性	说 明
FlatStyle	可以用这个属性改变按钮的样式。如果把样式设置为 PopUp，则该按钮就显示为平面，直到用户再把鼠标指针移动到它上面为止。此时，按钮会弹出，显示为 3D 外观
Enabled	这个属性派生于 Control，但这里仍讨论它，因为这是一个非常重要的属性。把 Enabled 设置为 false，则该按钮就会灰显，单击它，不会起任何作用
Image	可以指定一个在按钮上显示的图像(位图，图标等)
ImageAlign	指定按钮上的图像在什么地方显示

15.2.2 Button 控件的事件

到目前为止，按钮最常用的事件是 Click。只要用户单击了按钮，即当鼠标指向该按钮时，按下鼠标左键，再释放它，就会引发该事件。如果在按钮上单击了鼠标左键，然后把鼠标移动到其他位置，再释放鼠标，将不会引发 Click 事件。同样，在按钮得到焦点，且用户按下了回车键时，也会引发 Click 事件。如果窗体上有一个按钮，就总是要处理这个事件。

在下面的示例中，创建一个带有 3 个按钮的对话框。其中两个按钮在英语和丹麦语之间来回切换(也可以使用其他语言)，最后一个按钮关闭对话框。

试一试：按钮的测试

按照下列步骤创建一个小型 Windows 应用程序，使用 3 个按钮来改变对话框标题的文本：

(1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 Windows 应用程序 ButtonDialog。

(2) 单击窗口右上角的 x 旁边的图钉图标，钉住工具箱，双击 Button 控件 3 次。然后移动按钮，重新设置窗体的大小，如图 15-3 所示。

(3) 右击一个按钮，选择 Properties，在 Properties 窗口上选择(Name)编辑字段，键入相关的文本，修改每个按钮的 Name 属性。

(4) 与 Name 字段一样，修改每个按钮的 Text 属性，但不修改 Text 属性值的 button 前缀。

(5) 我们要在文本的前面显示一个标志，清晰地表示出每个按钮的作用。选择 English 按钮，找到 Image 属性。单击右边的(...)，打开一个对话框，该对话框可以把图像添加到窗体的资源文件中。单击 Import 按钮，浏览图标。我们要显示的图标包含在 ButtonDialog 项目中。该项目可以从 Wrox 主页上下载。选择图标 UK.PNG 和 DK.PNG 文件。

(6) 选择 UK，单击 OK。然后选择 buttonDanish，单击 Image 属性上的(...)，选择 DK，再单击 OK。

(7) 注意按钮文本和图标彼此遮挡，所以需要改变图标的对齐方式。对于 English 和 Danish 按钮，把 ImageAlign 属性改为 MiddleLeft。

(8) 此时，可以调整按钮的宽度，使文本不会正好从图像的右边开头。为此，可以选择每个按钮，在文本和图像之间留出间距。

(9) 最后单击窗体，把 Text 属性改为“Do you speak English?”。

这就是对话框的用户界面，如图 15-4 所示。

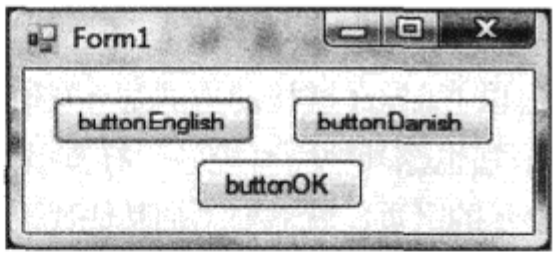


图　15-3

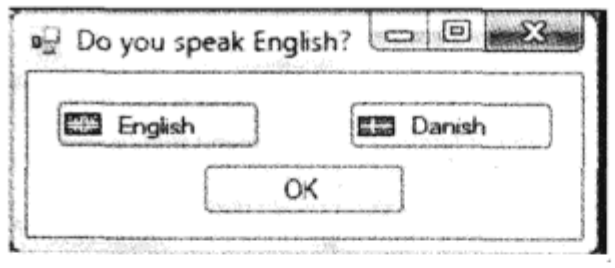


图　15-4

下面准备给对话框添加事件处理程序。双击 English 按钮，进入该控件默认事件的处理程序。Click 事件是按钮的默认事件，所以创建了它的处理程序。

15.2.3　添加事件处理程序

双击 English 按钮，在事件处理程序中添加如下代码：

```
private void buttonEnglish_Click (object sender, EventArgs e)
{
   Text = "Do you speak English?";
}
```

在 Visual Studio 创建处理事件的方法时，方法名是控件名、下划线和要处理的事件名的组合。

对于Click 事件，第一个参数 object sender 包含被单击的控件。在这个示例中，控件总是由方法名来标识，但在其他情况下，许多控件可能使用同一个方法来处理事件，此时就要通过查看这个值，

来确定是哪个控件调用了该方法。本章后面的“文本框控件”一节说明了多个控件如何使用同一个方法。另一个参数 System.EventArgs e 包含实际发生的事情的信息。在本例中，不需要这些信息。

返回设计视图，双击 Danish 按钮，进入这个按钮的事件处理程序，下面是代码：

```
private void buttonDanish_Click(object sender, EventArgs e)
{
   Text = "Taler du dansk?";
}
```

这个方法与 btnEnglish_Click 相同，但文本是丹麦文字。最后，以相同的方式添加 OK 按钮的事件处理程序。其代码有一些不同之处：

```
private void buttonOK_Click(object sender, EventArgs e)
{
   Application.Exit();
}
```

使用这段代码，就可以退出应用程序。这就是第一个示例。编译这个示例，运行它，单击其中的几个按钮，会发现对话框标题栏上的文本改变了。

15.3 Label 和 LinkLabel 控件

Label 控件也许是最常用的控件。在任何 Windows 应用程序中，都可以在对话框中见到它们。标签是一个简单的控件，其用途只有一个：在窗体上显示文本。

.NET Framework 包含两个标签控件，它们可以用两种截然不同的方式来显示：

- Label 是标准的 Windows 标签。
- LinkLabel 类似于标准标签(派生于标准标签)，但以 Internet 链接的方式显示(超链接)。

标准的 Label 通常不需要添加任何事件处理代码。但它也像其他所有控件一样支持事件。对于 LinkLabel 控件，如果希望用户可以单击它，进入文本中显示的网页，就需要添加其他代码。

Label 控件有非常多的属性。大多数属性都派生于 Control，但有一些属性是新增的。表 15-4 列出了最常见的属性。如果没有特别说明，Label 和 LinkLabel 控件中都存在这些属性。

表 15-4

属 性	说 明
BorderStyle	可以指定标签边框的样式。默认为无边框
FlatStyle	控制显示控件的方式。把这个属性设置为 PopUp，表示控件一直显示为平面样式，直到用户把鼠标指针移动到该控件上面，此时，控件显示为弹起样式
Image	指定要在标签上显示的图像(位图，图标等)
ImageAlign	指定图像显示在标签的什么地方
LinkArea	(只用于 LinkLabel)文本中显示为链接的部分
LinkColor	(只用于 LinkLabel)链接的颜色

(续表)

属　性	说　明
Links	(只用于 LinkLabel)LinkLabel 可以包含多个链接。利用这个属性可以查找需要的链接。控件会跟踪显示文本中的链接。不能在设计期间使用
LinkVisited	(只用于 LinkLabel)把它设置为 true，单击控件，链接就会显示为另一种颜色
TextAlign	指定文本显示在控件的什么地方
VisitedLinkColor	(只用于 LinkLabel)用户单击 LinkLabel 后控件的颜色

15.4　TextBox 控件

在希望用户输入程序员在设计阶段不知道的文本(如用户的姓名)时，应使用文本框。文本框的主要用途是让用户输入文本，用户可以输入任何字符，也可以只允许用户输入数值。

.NET Framework 内置了两个基本控件来提取用户输入的文本：TextBox 和 RichTextBox 。这两个控件都派生于基类 TextBoxBase，而 TextBoxBase 派生于 Control。

TextBoxBase 提供了在文本框中处理文本的基本功能，例如选择文本、剪切和从剪切板上粘贴，以及许多事件。这里不讨论派生关系，而是先介绍两个控件中比较简单的一个：TextBox。下面创建一个示例，说明 TextBox 的属性，后面在此基础上说明 RichTextBox 控件。

15.4.1　TextBox 控件的属性

如本章前面所述，列出控件的所有属性是不可能的，所以表 15-5 仅列出最常见的属性。

表　15-5

属　性	说　明
CausesValidation	当控件的这个属性设置为 true，且该控件要获得焦点时，会引发两个事件：Validating 和 Validated。可以处理这些事件，以便验证正在失去焦点的控件中数据的有效性。这可能使控件永远都不能获得焦点。下一节会讨论相关的事件
CharacterCasing	这个值表示 TextBox 是否会改变输入的文本的大小写。可能的值有： • Lower: 输入的所有文本都转换为小写 • Normal: 不对文本进行任何转换 • Upper:　输入的所有文本都转换为大写
MaxLength	这个值指定输入到 TextBox 中的文本的最大字符长度。把这个值设置为 0，表示最大字符长度仅受限于可用的内存
Multiline	表示该控件是否是一个多行控件。多行控件可以显示多行文本。如果将 Multiline 属性设置为 true，通常也把 WordWrap 也设置为 true
PasswordChar	指定是否用密码字符替换在单行文本框中输入的字符。如果 Multiline 属性为 true，这个属性就不起作用
ReadOnly	这个 Boolean 值表示文本是否为只读

(续表)

属 性	说 明
ScrollBars	指定多行文本框是否显示滚动条
SelectedText	在文本框中选择的文本
SelectionLength	在文本中选择的字符数。如果这个值设置得比文本中的总字符数大，则控件会把它重新设置为字符总数减去 SelectionStart 的值
SelectionStart	文本框中被选中文本的开头
WordWrap	指定在多行文本框中，如果一行的宽度超出了控件的宽度，其文本是否应自动换行

15.4.2 TextBox 控件的事件

在窗体上，对 TextBox 控件中文本进行精细的有效性验证会使一些用户很高兴，使另一些用户则会感到很生气。当用户单击了 OK 按钮后，对话框只验证其内容，此时用户可能非常生气。这种验证数据有效性的方式通常会显示一个信息框，告诉用户“第三个 TextBox”中的数据不正确。接着继续单击 OK 按钮，直到所有的数据都正确为止。显然这不是验证数据有效性的好方法，那么我们还能怎么做呢？

答案取决于 TextBox 控件提供的有效性验证事件。如果要确保文本框中不输入无效的字符，或者只输入某个范围内的数值，就需要告诉控件的用户：输入的值是否有效。

TextBox 控件提供了表 15-6 所示的事件(所有的事件都派生于 Control)。

表 15-6

名 称	说 明
Enter Leave Validating Validated	这 4 个事件按照列出的顺序引发。它们统称为“焦点事件”，当控件的焦点发生改变时引发，但有两个例外。Validating 和 Validated 仅在控件接收了焦点，且其 CausesValidation 属性设置为 true 时引发。接收焦点的控件引发事件的原因是有时即使焦点改变了，我们也不希望验证控件的有效性。例如用户单击了 Help 按钮
KeyDown KeyPress KeyUp	这 3 个事件称为“键事件”。它们可以监视和改变输入到控件中的内容。KeyDown 和 KeyUp 接收与所按下键对应的键码，这样就可以确定是否按下了特殊的键 Shift 或 Ctrl 和 F1。另一方面，KeyPress 接收与键对应的字符。这表示字母 a 的值与字母 A 的值不同。如果要排除某个范围内的字符，例如只允许输入数值，这是很有用的
TextChanged	只要文本框中的文本发生了改变，无论发生什么改变，都会引发该事件

下面的示例将创建一个对话框，在该对话框中可以输入姓名、地址、职业和年龄。这个示例的目的是为处理属性和使用事件打下基础，而不是创建什么特别有用的东西。

试一试：TextBoxTest

先建立用户界面：

(1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 Windows 应用程序 TextBoxControls。

(2) 创建如图 15-5 所示的窗体，把标签、文本框和按钮拖放到设计界面上。在重新设置两个文本框 textBoxAddress 和 textBoxOutput 的大小时，必须把它们的 Multiline 属性设置为 true。为此，右击控件，选择 Properties。

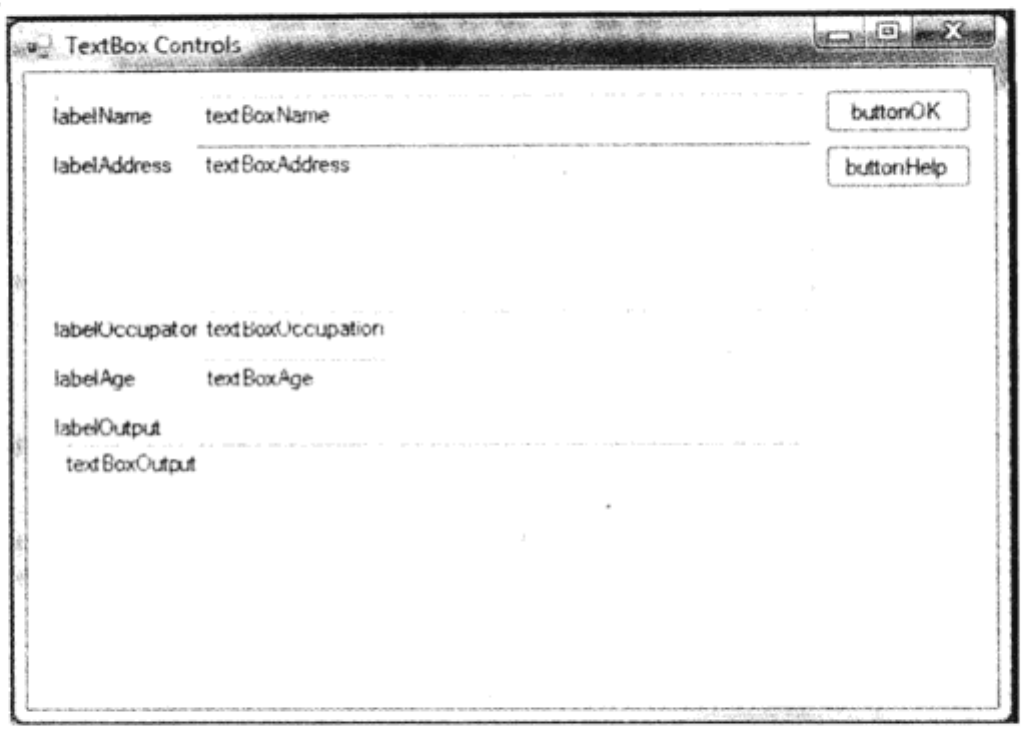


图　15-5

(3) 给控件命名，如图 15-5 所示。

(4) 把其他控件的 Text 属性设置为控件的名称，但不改变表示控件类型的前缀(即 Button 和 TextBox 和 Label)。把窗体的 Text 属性设置为 TextBoxControls。

(5) 把两个控件 textBoxOutput 和textBoxAddress 的 Scrollbars 属性设置为 Vertical。

(6) 把 textBoxOutput 控件的 ReadOnly 属性设置为 true。

(7) 把按钮 btnHelpButton 的 CausesValidation 属性设置为 false。在前面讨论 Validating 和 Validated 事件时，把这个属性设置为 false，就可以让用户单击这个按钮，而不必考虑输入的无效数据。

(8) 改变窗体的大小，使之适合于控件的大小，然后锚定控件，这样它们就可以在重新设置窗体的大小时正确地响应。下面一次设置每类控件的Anchor 属性。首先，按住 Ctrl 键，依次选择除了 textBoxOutput 之外的所有文本框控件。然后在 Properties 窗口中，把 Anchor 属性设置为Top, Left, Right，这就为每个选中的文本框控件设置了 Anchor 属性。再选择 textBoxOutput 控件，把 Anchor 属性设置为 Top, Bottom, Left, Right。现在把两个按钮控件的 Anchor 属性设置为 Top, Right。

锚定 textBoxOutput 而不是让它停靠在窗体底部的原因是，在拖动窗体时，要重新设置输出文本区域的大小。如果把该控件停靠在窗体的底部，它就会随窗体一起移动，但不会重新设置大小。

(9) 最后要设置的是，在窗体上，找到 Size 和MinimumSize 属性。如果窗体设置得比现在的小，就没有什么意义了，因此应把 MinimumSize 属性值设置得与 Size 属性值一样大。

示例的说明

设置窗体的可见部分现在已经完成了。如果运行它，则单击按钮或输入文本，将不会发生什么情况。但如果最大化或拖动对话框，控件就会按照希望的那样在用户界面上位于正确的位置，并重新设置其大小，以填充整个对话框。

15.4.3 添加事件处理程序

在设计视图中，双击 buttonOK 按钮，对另一个按钮重复这个过程。与本章前面的按钮示例一样，这会创建按钮的Click 事件处理程序。单击 OK 按钮，把输入文本框中的文本传送到只读的输出框中。

下面是两个 Click 事件处理程序的代码：


可从 wrox.com 下载源代码

```
private void buttonOK_Click(object sender, EventArgs e)
{
  // No testing for invalid values are made, as that should
  // not be necessary

  string output;

  // Concatenate the text values of the four TextBoxes.
  output = "Name: " + this.textBoxName.Text + "\r\n";
  output += "Address: " + this.textBoxAddress.Text + "\r\n";
  output += "Occupation: " + this.textBoxOccupation.Text + "\r\n";
  output += "Age: " + this.textBoxAge.Text;

  // Insert the new text.
  this.textBoxOutput.Text = output;
}

private void buttonHelp_Click(object sender, EventArgs e)
{
  // Write a short description of each TextBox in the Output TextBox.
  string output;
  output = "Name = Your name\r\n";
  output += "Address = Your address\r\n";
  output += "Occupation = Only allowed value is 'Programmer'\r\n";
  output += "Age = Your age";

  // Insert the new text.
  this.textBoxOutput.Text = output;
}
```

代码段 Chapter15\TextBoxControls\Form1.cs

在这两个函数中，使用了文本框的 Text 属性。textBoxAge 控件的 Text 属性获取输入的值，作为人的年龄，textBoxOutput 控件的 text 属性用于显示串在一起的文本。

我们插入用户输入的信息，无需检查该信息是否正确。这就是说，必须在其他地方进行检查。在本例中，必须满足许多条件，其值才是正确的。

- 用户名不能为空。
- 用户的年龄必须是一个大于或等于 0 的数字。
- 用户的职业必须是“程序员”或为空。
- 用户的地址不能为空。

从中可以看出，对两个文本框(textBoxName 和 textBoxAddress)进行的检查是相同的。并应禁止用户在 Age 框中输入无效的数值，最后必须检查用户是不是程序员。

为了防止用户在输入完信息之前单击 OK 按钮，要先把 OK 按钮的 Enabled 属性设置为 false，这应在窗体的构造函数中设置，而不是在 Properties 窗口中设置。如果在构造函数中设置属性，应确保在调用 InitializeComponent()中生成的代码之后，再设置这些属性：

```
public Form1()
{
   InitializeComponent();
   buttonOK.Enabled = false;
}
```

下面创建这两个文本框的处理程序，检查一下它们是否为空。为此，订阅文本框的 Validating 事件。因为需要在这两个控件上执行相同的操作，所以把同一个事件处理程序赋予它们。在窗体上选择这两个控件，再在 Events 列表中选择 Validating 事件，键入 textBoxEmpty_Validating 作为事件名。单击闪电按钮，就可以在 Properties 窗口中打开 Events 列表。

与前面的按钮事件处理程序不同，Validating 事件的处理程序是标准处理程序 System.EventHandler 的一个专用版本。这个事件需要专用处理程序的原因是，如果有效性验证失败，就必须有一种方式防止进行任何进一步的处理。如果要取消进一步的处理，就表示在输入有效的数据前，不能退出该文本框。

在以前的 Visual Studio 版本中，使用 GotFocus 和 LostFocus 事件执行控件的有效性验证时，Validating、Validated 事件和 CausesValidation 属性一起更正了一个错误。当 GotFocus 和 LostFocus 事件被连续引发时，就产生了这个错误；因为有效性验证代码试图在控件之间移动焦点，这将产生一个无限循环。

用下面的代码替换 VS 在事件处理程序中生成的 throw 语句：

```
private void textBoxEmpty_Validating (object sender,
                                      System.ComponentModel.CancelEventArgs e)
{
     TextBox tb = (TextBox)sender;

     if (tb.Text.Length == 0)
       tb.BackColor = Color.Red;
     else
       tb.BackColor = System.Drawing.SystemColors.Window;
     ValidateOK();
}
```

因为有多个文本框使用这个方法来处理事件，所以我们不知道哪个控件调用了函数，但无论是哪个控件调用了方法，其结果是一样的，所以可以对传送给文本框的 sender 参数进行类型转换，对它执行操作。

```
TextBox tb = (TextBox)sender;
```

如果文本框中的文本长度是 0，就把背景色设置为红色，把 Tag 设置为 false。如果不是，就把背景色设置为窗口的标准 Windows 颜色。

在设置控件的标准颜色时，应总是使用 System.Drawing.SystemColors 枚举中的颜色。如果把颜色设置为白色，而用户修改了默认的颜色设置，应用程序看起来就会很古怪。

ValidateOK()函数将在本例的最后介绍。与 Validating 事件相对照，下一个添加的处理程序是 Occupation 文本框的处理程序。这个过程与前面两个处理程序完全相同，但有效性验证代码有所不同，因为职业必须是 Programmer 或一个空字符串。要添加事件处理程序，应双击 textBoxOccupation 控件的 Validating 事件。

然后添加处理程序本身：

```
private void textBoxOccupation_Validating(object sender,
                                          System.ComponentModel.CancelEventArgs e)
{
      TextBox tb = (TextBox)sender;

      if (tb.Text == "Programmer" || tb.Text.Length == 0)
        tb.BackColor = System.Drawing.SystemColors.Window;
      else
        tb.BackColor = Color.Red;
      ValidateOK();
}
```

倒数第二个要处理的事项是处理 Age 文本框。我们希望用户只键入正数(包括 0，这样可以使检测简单一些)。为此，使用KeyPress 事件，在不想要的字符在文本框中显示出来之前就删除它们。还要把输入到控件中的字符数限制为 3 个。

首先，把 textBoxAge 控件的 MaxLength 属性设置为 3，然后在 Properties 窗口的 Events 列表中双击 KeyPress 事件，以订阅它。KeyPress 事件处理程序也是专用的，其中提供了 System.Windows.Forms.KeyPressEventHandler，因为事件需要被按下键的信息。

然后给事件处理程序添加如下代码：

```
private void textBoxAge_KeyPress(object sender, KeyPressEventArgs e)
{
   if ((e.KeyChar < 48 || e.KeyChar > 57) && e.KeyChar != 8)
      e.Handled = true;
}
```

0~9 之间数字的 ASCII 值是 48~57，所以应保证字符在这个范围内。但有一个例外。ASCII 值 8 表示退格键，为了编辑方便，允许跳过它。把 KeyPressEventArgs 的 Handled 属性设置为 true，告诉控件不应对字符进行其他任何操作，所以如果按下的键不是数字或退格，就不显示该字符。

现在控件没有标记为有效或无效，这是因为需要进行另一个检查，看看是否输入了信息。这是很简单的，因为前面已经编写了执行该检查的方法，在 Events 列表中给 textBoxAge 控件选择 Validating 事件下拉列表中的 textBoxEmpty_Validating 事件处理程序。

最后一件事：启用或禁用 OK 按钮的 ValidateOK 方法：

```
private void ValidateOK()
```

```
{
    buttonOK.Enabled = (textBoxName.BackColor != Color.Red &&
    textBoxAddress.BackColor != Color.Red &&
    textBoxOccupation.BackColor != Color.Red &&
    textBoxAge.BackColor != Color.Red);
}
```

如果所有的文本框都不使用红色作为背景色，这个方法就把 OK 按钮的 Enabled 属性设置为 true。

如果现在测试该程序，就会得到如图 15-6 所示的结果。注意，在文本框中输入了无效的数据，但背景色没有改为红色时，可以单击 Help 按钮。

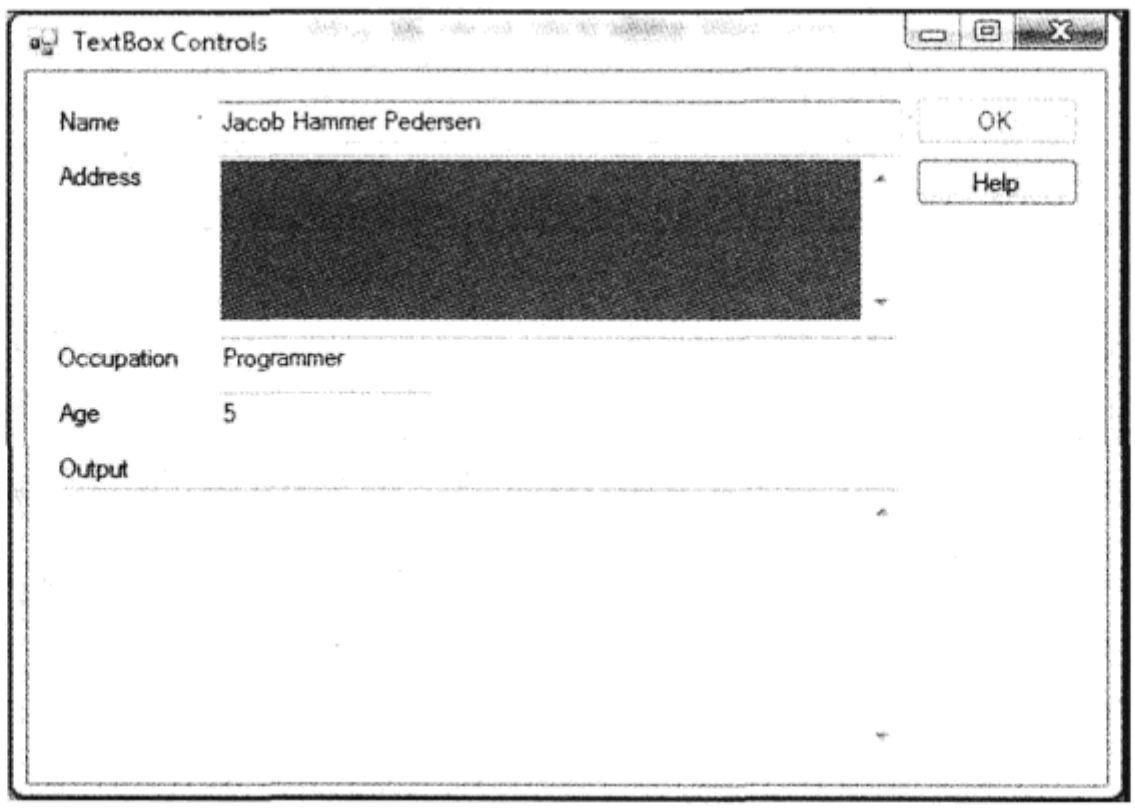


图　15-6

15.5　RadioButton 和 CheckBox 控件

如前所述，RadioButton 和 CheckBox 控件与 Button 控件有相同的基类，但它们的外观和用法大不相同。

传统上，单选按钮显示为一个标签，左边是一个圆点，该点可以是选中或未选中。在要给用户提供两个或多个互斥选项时，就可以使用单选按钮。例如，询问用户的性别。

把单选按钮组合在一起，给它们创建一个逻辑单元，此时必须使用 GroupBox 控件或其他一些容器。首先在窗体上拖放一个组框，再把需要的RadioButton 按钮放在组框的边界之内，RadioButton 按钮会自动改变自己的状态，以使组框中只选中一个选项。如果不把它们放在组框中，则在任意时刻，窗体上只有一个RadioButton 被选中。

传统上，CheckBox 显示为一个标签，左边是一个小方框。在希望用户可以选择一个或多个选项时，就应使用复选框。例如询问用户要使用的操作系统(如Windows Vista、Windows XP 和 Linux 等)。

下面介绍这两个控件的重要属性和事件，从 RadioButton 开始，然后用一个简短小示例说明它们的用法。

15.5.1 RadioButton 控件的属性

这个控件派生于 ButtonBase，前面已经有一个使用按钮的示例了，所以需要描述的属性仅有几个，如表 15-7 所示。完整的列表请参阅.NET Framework SDK 文档说明。

表 15-7

属 性	说 明
Appearance	RadioButton 可以显示为一个标签，相应的圆点放在左边、中间或右边，或者显示为标准按钮。当它显示为按钮时，控件被选中时显示为按下状态，否则显示为弹起状态
AutoCheck	如果这个属性为 true，用户单击单选按钮时，会显示一个选中标记。如果该属性为 false，就必须在 Click 事件处理程序的代码中手工选中单选按钮
CheckAlign	使用这个属性，可以改变单选按钮的复选框的对齐形式，默认是 ContentAlignment. MiddleLeft
Checked	表示控件的状态。如果控件有一个选中标记，它就是 true，否则为 false

15.5.2 RadioButton 控件的事件

在处理 RadioButton 控件时，通常只使用一个事件，但还可以订阅许多其他事件。本章只介绍两个事件，介绍第二个事件的原因是它们之间有微妙的区别，如表 15-8 所示。

表 15-8

属 性	说 明
CheckedChanged	当 RadioButton 的选中选项发生改变时，引发这个事件
Click	每次单击 RadioButton 时，都会引发该事件。这与 CheckedChange 事件是不同的，因为连续单击 RadioButton 两次或多次只改变 Checked 属性一次(而且只有尚未选中时才如此)。而且，如果被单击按钮的 AutoCheck 属性是 false，则该按钮根本不会被选中，只引发 Click 事件

15.5.3 CheckBox 控件的属性

可以想像，这个控件的属性和事件非常类似于 RadioButton 控件，但有两个新属性，如表 15-9 所示。

表 15-9

属 性	说 明
CheckState	与 RadioButton 不同，CheckBox 有 3 种状态： Checked、Indeterminate 和 Unchecked。复选框的状态是 Indeterminate 时，控件旁边的复选框通常是灰色的，表示复选框的当前值是无效的。或者无法确定(例如，如果选中标记表示文件的只读状态，且选中了两个文件，则其中一个文件是只读的，另一个文件不是)，或者在当前环境下没有意义
ThreeState	这个属性为 false 时，用户就不能把 CheckState 属性改为 Indeterminate。但仍可以在代码中把 CheckState 属性改为 Indeterminate

15.5.4　CheckBox 控件的事件

一般只使用这个控件的一两个事件。注意，RadioButton 和 CheckBox 控件都有 CheckChanged 事件，但效果不同，如表 15-10 所示。

表　15-10

事　件	说　明
CheckedChanged	当复选框的 Checked 属性发生改变时，就引发该事件。注意在复选框中，当 ThreeState 属性为 true 时，单击复选框可能不会改变 Checked 属性。在复选框从 Checked 变为 Indeterminate 状态时，就会出现这种情况
CheckedStateChanged	当 CheckedState 属性改变时，引发该事件。CheckedState 属性的值可以是 Checked 和 Unchecked。只要 Checked 属性改变了，就引发该事件。另外，当状态从 Checked 变为 Indeterminate 时，也会引发该事件

前面总结了 RadioButton 和 CheckBox 控件的事件和属性。在使用它们之前，先介绍一下前面提及的 GroupBox 控件。

15.5.5　GroupBox 控件

GroupBox 控件常常用于合理地组合一组控件，如 RadioButton 及 CheckBox 控件，显示一个框架，其上有一个标题。

组框的用法非常简单，把它拖放到窗体上，再把所需的控件拖放到组框中即可(但其顺序不能颠倒——不能把组框放在已有的控件上面)。其结果是父控件是组框，而不是窗体，所以在任意时刻，可以选择多个 RadioButton。但在组框中，一次只能选择一个 RadioButton。

这里需要解释一下父控件和子控件的关系。把一个控件放在窗体上时，窗体就是该控件的父控件，所以该控件是窗体的一个子控件。而把一个GroupBox 放在窗体上时，它就成为窗体的一个子控件。而组框本身可以包含控件，所以它就是这些控件的父控件，其结果是移动 GroupBox 时，其中的所有控件也会随之移动。

把控件放在组框上的另一个结果是可以改变其中所有控件的某些属性，方法是在组框上设置这些属性。例如，如果要禁用组框中的所有控件，只需把组框的 Enabled 属性设置为 false 即可。

下面用一个示例说明 GroupBox 控件的用法。

试一试：RadioButton 和 CheckBox 示例

下面修改本章全面创建的 TextBoxControls 示例。在该示例中，唯一可能的职业是程序员。下面不强迫用户填写程序员，而是把这个文本框改成复选框。为了说明 RadioButton 的用法，我们将要求用户再提供一条信息：性别。

把文本框示例改为：

(1) 删除 labelOccupation 标签和文本框 textBoxOccupation。

(2) 添加一个CheckBox、一个 GroupBox 和两个 RadioButton 控件，并命名这些新控件，如图 15-7 所示。注意与前面使用的其他控件不同，GroupBox 控件位于Toolbox 面板的 Containers 选项卡上。

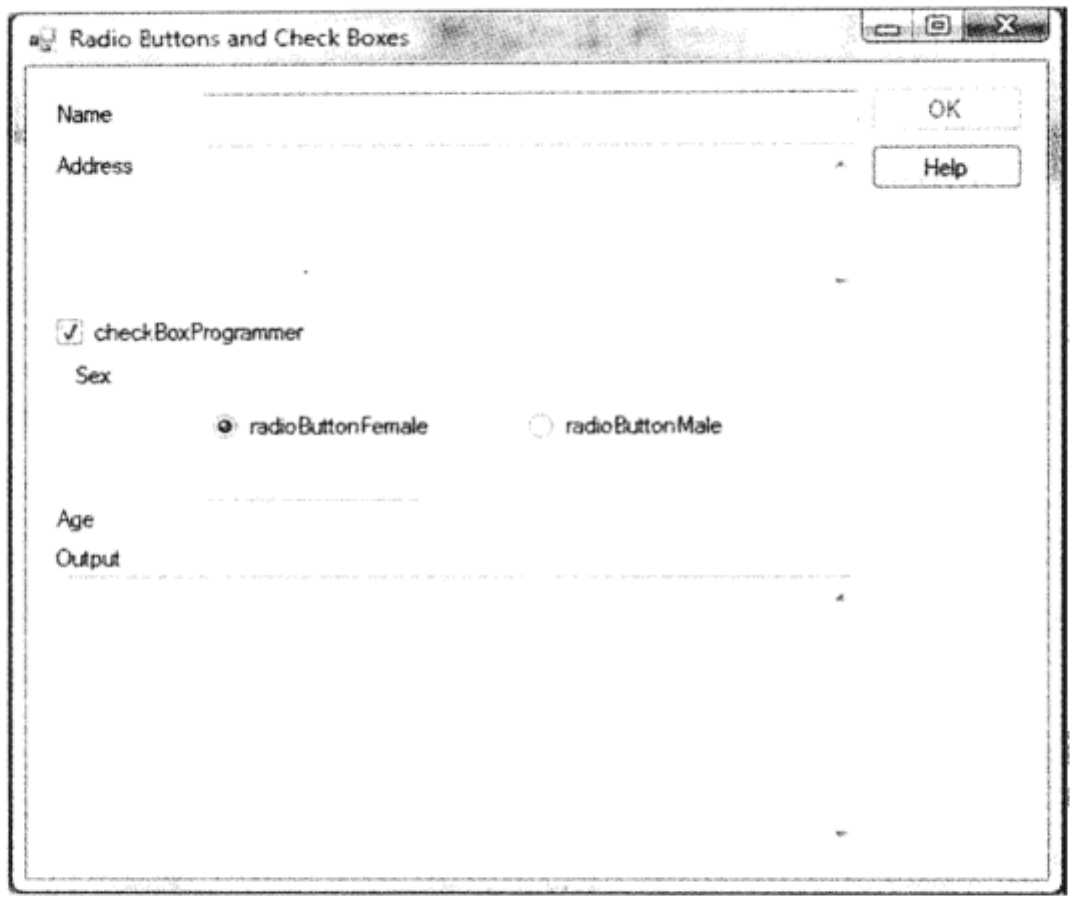


图 15-7

(3) RadioButton 和 CheckBox 控件的 Text 属性应与该控件名相同(前 3 个字符不算)。GroupBox 的 Text 属性应是 Sex。

(4) 把 checkBoxProgrammer 复选框的 Checked 属性设置为 true。注意 CheckState 属性自动改为 Checked。

(5) 把 radioButtonMale 或 radioButtonFemale 的 Checked 属性设置为 true。注意不能把它们两个同时设置为 true。否则，另一个 RadioButton 的值会自动变为 false。

对这个示例的可见部分不再需要更多的修改，但代码要进行许多修改。首先，需要删除所有对已删除文本框的引用。进入代码，完成下述步骤。

(1) 在 ValidateOK 方法中，删除对 textBoxOccupation 背景的测试：


可从 wrox.com 下载源代码

```
private void ValidateOK()
{
  // Set the OK button to enabled if all the Tags are true.
  buttonOK.Enabled = (textBoxName.BackColor != Color.Red &&
                      textBoxAddress.BackColor != Color.Red &&
                      textBoxAge.BackColor != Color.Red);
}
```

代码段 Chapter15\Radio and Check Buttons\Form1.cs

(2) 彻底删除 textBoxOccupation_Validating 方法。

(3) 删除 buttonOK_Click 中的引用。

示例的说明

这里使用的是复选框，而不是文本框，所以用户不会输入无效的信息，因为用户要么是一个程序员，要么不是。

我们也知道用户要么是男性，要么是女性，因为前面把一个 RadioButton 的属性设置为 true，这样用户就不会选择无效的值。因此，下面只需要修改帮助文本和输出。在按钮事件处理程序中完成它：

```
private void buttonHelp_Click(object sender, System.EventArgs e)
{
   // Write a short descrption of each TextBox in the Output TextBox.
   string output;

   output = "Name = Your name\r\n";
   output += "Address = Your address\r\n";
   output += "Programmer = Check 'Programmer' if you are a programmer\r\n";
   output += "Sex = Choose your sex\r\n";
   output += "Age = Your age";

   // Insert the new text.
   this.textBoxOutput.Text = output;
}
```

只是修改了帮助文本，所以不要对 help 方法感到惊讶。OK 方法显得更有趣一点：

```
private void buttonOK_Click(object sender, System.EventArgs e)
{
   // No testing for invalid values are made, as that should
   // not be neccessary

   string output;

   // Concatenate the text values of the four TextBoxes.
   output = "Name: " + this.textBoxName.Text + "\r\n";
   output += "Address: " + this.textBoxAddress.Text + "\r\n";
   output += "Occupation: " + (string)(this.chkProgrammer.Checked ?
          "Programmer" : "Not a programmer") + "\r\n";
   output += "Sex: " + (string)(this.radioButtonFemale.Checked ? "Female" :
                                                        "Male") + "\r\n";
   output += "Age: " + this.textBoxAge.Text;

   // Insert the new text.
   this.textBoxOutput.Text = output;
}
```

在突出显示的代码中，第一行打印出了用户的职业。考察一下复选框的 Checked 属性，如果它是 true，就写入字符串“Programmer”，如果它是 false，就填写“Not a programmer”。

第二行代码检查单选按钮 radioButtonFemale。如果该控件的 Checked 属性是 true，则该用户是一位女性。如果它是 false，则该用户是一位男性。在启动程序时，可以不选中任何单选按钮，但因为是在设计期间选择了其中一个单选按钮，所以可以肯定总是会选中两个单选按钮中的一个。

现在运行示例，得到如图 15-8 所示的结果。

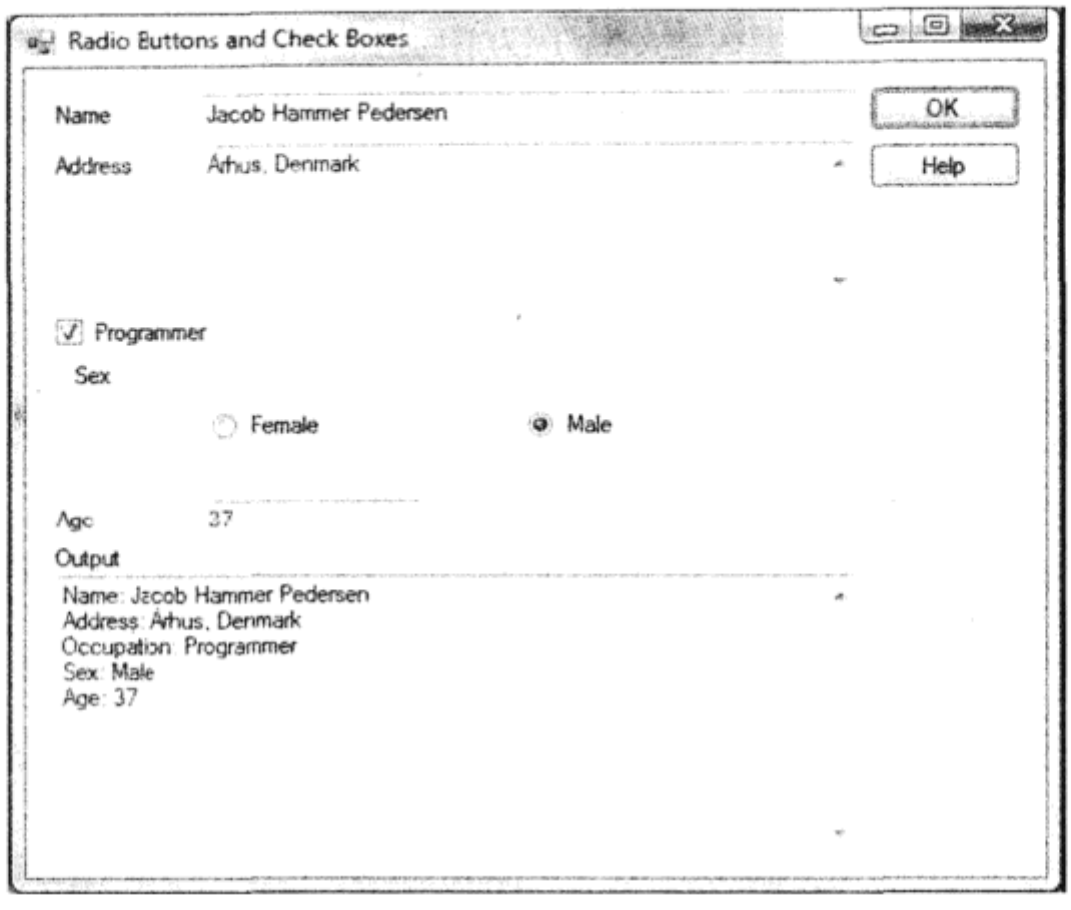


图 15-8

15.6 RichTextBox 控件

与常用的 TextBox 一样，RichTextBox 控件派生于 TextBoxBase。所以，它与 TextBox 共享许多功能，但许多功能是不同的。TextBox 常用于从用户处获取简短的文本字符串，而 RichTextBox 用于显示和输入格式化的文本(例如，黑体、下划线和斜体)。它使用标准的格式化文本，称为 Rich Text Format (富文本格式)或 RTF。

在上面的示例中，我们使用了标准的 TextBox。也可以使用 RichTextBox 来完成该任务。实际上，如后面的示例所示，可以删除 textBoxOutput 文本框，在它的位置上插入一个同名的 RichTextBox，这个示例还会像以前那样运行。

15.6.1 RichTextBox 控件的属性

如果这种文本框比上一节介绍的文本框更高级，我们就会期望它有一些新属性。表15-11 中列出了 RichTextBox 的一些最常用属性。

表 15-11

属 性	说 明
CanRedo	如果上一个被撤消的操作可以使用 Redo 重复，这个属性就是 true
CanUndo	如果可以在 RichTextBox 上撤消上一个操作，这个属性就是 true，注意，CanUndo 在 TextBoxBase 中定义，所以也可用于 TextBox 控件
RedoActionName	这个属性包含通过 Redo 方法执行的操作名称

(续表)

属　性	说　明
DetectUrls	把这个属性设置为 true，可以使控件检测 URL，并格式化它们(像在浏览器中那样有下划线)
Rtf	它对应于 Text 属性，但包含 RTF 格式的文本
SelectedRtf	使用这个属性可以获取或设置控件中被选中的 RTF 格式文本。如果把相应文本复制到另一个应用程序中，例如 Word，该文本会保留所有的格式化信息
SelectedText	与 SelectedRtf 一样，可以使用这个属性获取或设置被选中的文本。但与该属性的 RTF 版本不同，所有的格式化信息都会丢失
SelectionAlignment	它表示选中文本的对齐方式，可以是 Center、Left 或 Right
SelectionBullet	使用这个属性可以确定选中的文本是否格式化为项目符号的格式，或使用它插入或删除项目符号
BulletIndent	使用这个属性可以指定项目符号的缩进像素值
SelectionColor	这个属性可以修改选中文本的颜色
SelectionFont	这个属性可以修改选中文本的字体
SelectionLength	使用这个属性可以设置或获取选中文本的长度
SelectionType	这个属性包含了选中文本的信息。它可以确定是选择了一个或多个 OLE 对象，还是仅选择了文本
ShowSelectionMargin	如果把这个属性设置为 true，在 RichTextBox 的左边就会出现页边距，这将使用户更易于选择文本
UndoActionName	如果用户选择撤消某个动作，该属性将获取该操作的名称
SelectionProtected	把这个属性设置为 true，可以指定不修改文本的某些部分

从上表可以看出，大多数新属性都与选中的文本有关。这是因为在用户处理其文本时，对它们应用的任何格式化操作都是对用户选择出来的文本进行的。万一没有选择出文本，格式化操作就从文本中光标所在的位置开始应用，该位置称为插入点。

15.6.2　RichTextBox 控件的事件

RichTextBox 使用的大多数事件与 TextBox 使用的事件相同，表 15-12 列出几个有趣的新事件。

表　15-12

名　称	说　明
LinkClicked	在用户单击文本中的链接时，引发该事件
Protected	在用户尝试修改已经标记为受保护的文本时，引发该事件
SelectionChanged	在选中文本发生变化时，引发该事件。如果因某些原因不希望用户修改选中的文本，就可以在这里禁止修改

在下面的示例中，将创建一个非常基本的文本编辑器。它说明了如何修改文本的基本格式，如

何加载和保存 RichTextBox 中的文本。为了简单起见，这个示例从固定文件中加载并保存在固定的文件中。

试一试：RichTextBox 示例

与往常一样，首先设计窗体：

(1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 C# Windows 应用程序，命名为 Simple Text Editor。

(2) 创建窗体，如图 15-9 所示。文本框 textBoxSize 应是一个 TextBox 控件。RichTextBoxText 文本框应是一个 RichTextBox 控件。

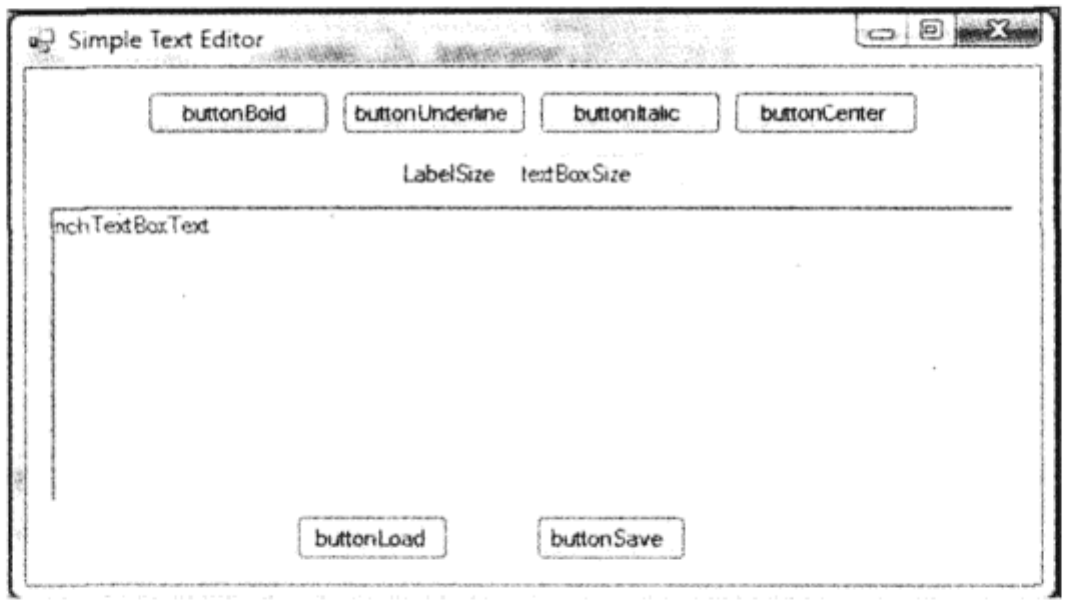


图 15-9

(3) 如图 15-9 所示命名控件。

(4) 除了文本框以外，把其他控件的 Text 属性设置为其控件名称(但表示该控件类型的名称的第一部分不算)。

(5) 把 textBoxSize 文本框的 Text 属性改为 10。

(6) 锚定控件，如表 15-13 所示。

表 15-13

控 件 名 称	Anchor 值
buttonLoad 和 buttonSave	Bottom
richTextBoxText	Top, Left, Bottom, Right
其他控件	Top

(7) 把窗体的 MinimumSize 属性值设置为 Size 属性的值。

示例的说明

前面是该示例的可见部分，下面将分析代码。双击 Bold 按钮，在代码中添加 Click 事件处理程序。下面是该事件的代码：

可从 wrox.com 下载源代码

```
private void buttonBold_Click(object sender, EventArgs e)
{
   Font oldFont;
```

```
    Font newFont;

    oldFont = this.richTextBoxText.SelectionFont;

    if (oldFont.Bold)
      newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
    else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

    this.richTextBoxText.SelectionFont = newFont;
    this.richTextBoxText.Focus();
  }
```

代码段 Chapter15\Simple Text Editor\Form1.cs

首先获取当前选中文本使用的字体，并把它赋给一个局部变量 oldFont。然后检查一下选中文本是否为粗体。如果是，就去除粗体设置；否则就设置粗体。使用 oldFont 作为原型，创建一个新字体，但根据需要添加或删除粗体格式。

最后，把新字体赋给选中的文本，把焦点返回给 RichTextBox。

buttonItalic 和 buttonUnderline 的事件处理程序的代码与上面的代码相同，但检查对应样式的代码不同。双击 Italic 和 Underline 两个按钮，添加下列代码：

```
private void buttonItalic_Click(object sender, EventArgs e)
{
  Font oldFont;
  Font newFont;

  // Get the font that is being used in the selected text.
  oldFont = this.richTextBoxText.SelectionFont;

  // If the font is using Italic style now, we should remove it.
  if (oldFont.Italic)
    newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
  else
    newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

  // Insert the new font.
  this.richTextBoxText.SelectionFont = newFont;
  this.richTextBoxText.Focus();
}

private void buttonUnderline_Click(object sender, System.EventArgs e)
{
  Font oldFont;
  Font newFont;

  // Get the font that is being used in the selected text.
  oldFont = this.richTextBoxText.SelectionFont;

  // If the font is using Underline style now, we should remove it.
  if (oldFont.Underline)
    newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
```

```
    else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);

    // Insert the new font.
    this.richTextBoxText.SelectionFont = newFont;
    this.richTextBoxText.Focus();
}
```

双击最后一个格式化按钮 Center，添加下列代码：

```
private void buttonCenter_Click(object sender, System.EventArgs e)
{
    if (this.richTextBoxText.SelectionAlignment == HorizontalAlignment.Center)
      this.richTextBoxText.SelectionAlignment = HorizontalAlignment.Left;
    else
      this.richTextBoxText.SelectionAlignment = HorizontalAlignment.Center;
    this.richTextBoxText.Focus();
}
```

这里必须检查另一个属性 SelectionAlignment，看看选中的文本是否已经居中对齐，因为我们希望按钮像一个切换按钮那样运作。如果文本已居中，就使它左对齐，否则就使它居中。HorizontalAlignment 是一个枚举，其值可以是 Left、Right、Center、Justify 和 NotSet。在本例中，只检查一下是否设置了 Center，如果已经设置了，就把对齐方式设置为 Left。如果不是，就设置为 Center。

文本编辑器能进行的最后一个格式化操作是设置文本的大小。为文本框 Size 添加两个事件处理程序，一个处理程序控制输入，另一个处理程序检测用户输入完一个值的时间。

在 Properties 窗口的 Events 列表中找到并双击 textBoxSize 控件的 KeyPress 和 Validated 事件，给处理程序添加代码。

与前面示例使用的 Validating 事件不同，Validated 事件在进行完验证后引发。这两个事件处理程序都使用一个帮助方法 ApplyTextSize，该方法带有一个字符串参数，表示文本的大小：

```
private void textBoxSize_KeyPress(object sender, KeyPressEventArgs e)
{
    if ((e.KeyChar < 48 || e.KeyChar > 57) &&
                                    e.KeyChar != 8 && e.KeyChar != 13)
      e.Handled = true;
    else if (e.KeyChar == 13)
    {
      TextBox txt = (TextBox)sender;

      if (txt.Text.Length > 0)
        ApplyTextSize(txt.Text);
      e.Handled = true;
      this.richTextBoxText.Focus();
    }
}

private void textBoxSize_Validated(object sender, CancelEventArgs e)
{
    ApplyTextSize(txt.Text);
    this.richTextBoxText.Focus();
}
```

```
private void ApplyTextSize(string textSize)
{
   float newSize = Convert.ToSingle(textSize);
   FontFamily currentFontFamily;
   Font newFont;

   currentFontFamily = this.richTextBoxText.SelectionFont.FontFamily;
   newFont = new Font(currentFontFamily, newSize);

   this.richTextBoxText.SelectionFont = newFont;
```

KeyPress 事件只允许用户输入一个整数，并在用户按下回车键时，调用 ApplyTextSize。我们感兴趣的是帮助方法 ApplyTextSize()。它首先把文本的大小从字符串转换为浮点数。如前所述，我们只允许用户输入整数，但在创建新字体时，需要使用浮点数，所以把它转换为正确的类型。

之后，获取字体所属的字体系列，从该系列中创建一个带有新字号的新字体。最后，把选中文本的字体设置为新字体。

这就是我们所能进行的所有格式化操作，有一些操作可以由 RichTextBox 本身处理。如果现在尝试运行这个示例，就可以把文本设置为黑体、斜体和下划线，还可以使文本居中对齐。这就是我们期望的操作，但还有一些比较有趣的操作。试着在文本中键入一个网址，例如 http://www.wrox.com，该文本就被控件识别为一个 Internet 地址，加上下划线，当把鼠标指针移到该文本上时，鼠标指针就会变成手的形状。再添加一些代码，就可以在单击该文本时打开一个网页，为此，需要处理用户单击链接时引发的事件：LinkClicked。

在 Properties 窗口的 Events 列表中找到 LinkClicked 事件，双击它，在事件处理程序中添加代码。我们以前没有见过这个事件处理程序。它用于提供被单击的链接的文本，处理程序非常简单，如下所示：

```
private void richTextBoxText_LinkedClick(object sender,
                              System.Windows.Forms.LinkClickedEventArgs e)
{
    System.Diagnostics.Process.Start(e.LinkText);
}
```

这段代码打开了默认的浏览器(如果浏览器没有打开)，并导航到该链接指向的站点。

应用程序的编辑部分就完成了。剩下的是加载和保存控件的内容。这里使用一个固定的文件。双击 Load 按钮，添加下面的代码：

```
private void buttonLoad_Click(object sender, EventArgs e)
{
   try
   {
      richTextBoxText.LoadFile("Test.rtf");
   }
   catch (System.IO.FileNotFoundException)
   {
      MessageBox.Show("No file to load yet");
   }
}
```

这就完成了，不需要做其他工作。因为我们处理的是文件，所以总是有可能遇到异常，必须处

理这些异常。在Load 方法中，处理了因文件不存在而抛出的异常。保存文件也很简单，双击 Save 按钮，添加下面的代码：

```
private void buttonSave_Click(object sender, EventArgs e)
{
   try
   {
      richTextBoxText.SaveFile("Test.rtf");
   }
   catch (System.Exception err)
   {
      MessageBox.Show(err.Message);
   }
}
```

现在运行示例，格式化一些文本，再单击 Save 按钮。清空文本框，单击 Load 按钮，刚才保存的文本就会再次显示出来。

运行它时，应得到如图 15-10 所示的结果。

图 15-10

15.7 ListBox 和 CheckedListBox 控件

列表框用于显示一组字符串，可以一次从中选择一个或多个选项。与复选框和单选按钮一样，列表框也提供了要求用户选择一个或多个选项的方式。在设计期间，如果不知道用户要选择的数值个数，就应使用列表框(例如同事列表)。即使在设计期间知道所有可能的值，但列表中的值非常多，也应考虑使用列表框。

ListBox 类派生于 ListControl 类。后者提供了.NET Framework 内置列表类型控件的基本功能。

另一类列表框称为 CheckedListBox，派生于ListBox 类。它提供的列表类似于 ListBox，但除了文本字符串以外，每个列表选项还附带一个复选标记。

15.7.1 ListBox 控件的属性

除非明确声明，表 15-14 中列出的所有属性都可用于 ListBox 类和 CheckedListBox 类。

表　15-14

属　性	说　明
SelectedIndex	这个值表示列表框中选中项的基于 0 的索引。如果列表框可以一次选择多个选项，这个属性就包含选中列表中第一个选项的索引
ColumnWidth	在包含多个列的列表框中，这个属性指定列宽
Items	Items 集合包含列表框中的所有选项，使用这个集合的属性可以增加和删除选项
MultiColumn	列表框可以有多个列。使用这个属性可以获取是否采用多列形式的信息，也可以设置是否采用多列形式
SelectedIndices	这个属性是一个集合，包含列表框中选中项的所有基于 0 的索引
SelectedItem	在只能选择一个选项的列表框中，这个属性包含选中的选项。在可以选择多个选项的列表框中，这个属性包含选中项中的第一项
SelectedItems	这个属性是一个集合，包含当前选中的所有选项
SelectionMode	在列表框中，可以使用 ListSelectionMode 枚举中的 4 种选择模式： • None: 不能选择任何选项 • One: 一次只能选择一个选项 • MultiSimple: 可以选择多个选项。使用这个模式，在单击列表中的一项时，该项就会被选中，即使单击另一项，该项也仍保持选中状态，除非再次单击它 • MultiExtended: 可以选择多个选项，用户还可以使用 Ctrl、Shift 和箭头键进行选择。它与 MultiSimple 不同，如果先单击一项，然后单击另一项，则只选中第二个单击的项
Sorted	把这个属性设置为 true，会使列表框对它包含的选项按字母顺序排序
Text	许多控件都有 Text 属性。但这个 Text 属性与其他控件的 Text 属性大不相同。如果设置列表框控件的 Text 属性，它将搜索匹配该文本的选项，并选择该选项。如果获取 Text 属性，返回的值是列表中第一个选中的选项。如果 SelectionMode 是 None，就不能使用这个属性
CheckedIndices	(只用于 CheckedListBox)这个属性是一个集合，包含 CheckedListBox 中状态是 checked 或 indeterminate 的所有选项的索引
CheckedItems	(只用于 CheckedListBox)这是一个集合，包含 CheckedListBox 中状态是 Checked 或 Indeterminate 的所有选项
CheckOnClick	(只用于 CheckedListBox)如果这个属性是 true，则选项就会在用户单击它时改变它的状态
ThreeDCheckBoxes	(只用于 CheckedListBox)设置这个属性，就可以选择平面或正常的 CheckBoxes

15.7.2　ListBox 控件的方法

为了高效地操作列表框，读者应了解它可以调用的一些方法。表 15-15 列出了最常用的方法。除非特别声明，否则这些方法均属于 ListBox 和 CheckedListBox 类。

表 15-15

方 法	说 明
ClearSelected()	清除列表框中的所有选中项
FindString()	查找列表框中第一个以指定字符串开头的字符串，例如 FindString("a")就是查找列表框中第一个以 a 开头的字符串
FindStringExact()	与 FindString 类似，但必须匹配整个字符串
GetSelected()	返回一个表示是否选择一个选项的值
SetSelected()	设置或清除选项的选中状态
ToString()	返回当前选中的选项
GetItemChecked()	(只用于 CheckedListBox)返回一个表示选项是否被选中的值
GetItemCheckState()	(只用于 CheckedListBox)返回一个表示选项的选中状态的值
SetItemChecked()	(只用于 CheckedListBox)设置指定为选中状态的选项
SetItemCheckState()	(只用于 CheckedListBox)设置选项的选中状态

15.7.3 ListBox 控件的事件

通常，在处理 ListBox 和 CheckedListBox 时，使用的事件都与用户选中的选项有关，如表 15-16 所示。

表 15-16

事 件	说 明
ItemCheck	(只用于 CheckedListBox)在列表框中一个选项的选中状态改变时引发该事件
SelectedIndexChanged	在选中选项的索引改变时引发该事件

下面用 ListBox 和 CheckedListBox 创建一个小示例。用户可以查看 CheckedListBox 中的选项，然后单击一个按钮，把选中的选项移动到一般的 ListBox 中。

试一试：使用 ListBox 控件

创建如下所示的对话框：

(1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 Windows 应用程序 Lists。

(2) 在窗体上添加一个 ListBox、一个 CheckedListBox 和一个按钮，改变其名称，如图 15-11 所示。

(3) 把按钮的 Text 属性改成 Move。

(4) 把 CheckedListBox 的属性 CheckOnClick 改成 true。

(5) 单击省略号(...)，打开 CheckedListBox 控件的 Items 编辑器。再输入 One、Two、Three、Four、Five、Six、Seven、Eight 和 Nine，一项占一行，然后单击 OK。

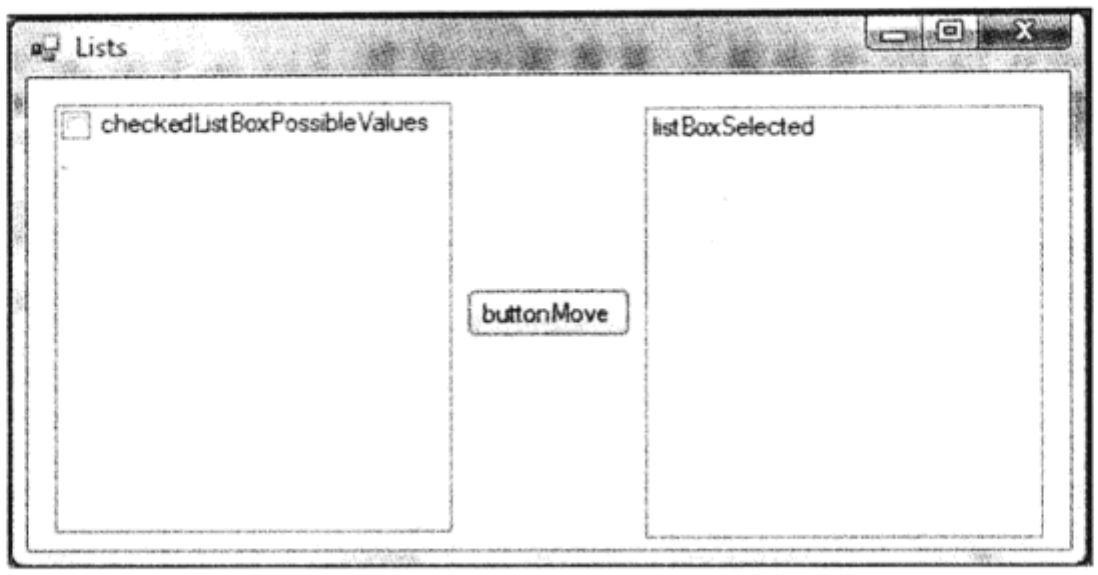


图　15-11

(6) 在 CheckedListBox 中再插入一项，但在代码中插入：


可从 wrox.com 下载源代码

```
public Form1()
{
  InitializeComponent();

  checkedListBoxPossibleValues.Items.Add("Ten");
}
```

代码段 Chapter15\Lists\Form1.cs

现在准备添加事件处理程序。可以添加一些代码，把 checkedListBox 中的项移动到正常的列表框中。当用户单击 Move 按钮时，要查找被选中的选项，再把它们复制到右边的列表框中。

(7) 双击该按钮，输入下面的代码：

```
private void buttonMove_Click(object sender, EventArgs e)
  {
      if (checkedListBoxPossibleValues.CheckedItems.Count > 0)
      {
        listBoxSelected.Items.Clear();
        foreach (string item in checkedListBoxPossibleValues.CheckedItems)
        {
          listBoxSelected.Items.Add(item.ToString());
        }
        for (int i = 0; i < checkedListBoxPossibleValues.Items.Count; i++)
          checkedListBoxPossibleValues.SetItemChecked(i, false);
      }
  }
```

示例的说明

首先查看一下 CheckedItems 集合的 Count 属性。如果集合中有选中的选项，该属性就会大于 0。接着清除 listBoxSelected 列表框中的所有选项，循环 CheckedItems 集合，把每个选项添加到 listBoxSelected 列表框中。最后，删除 CheckedListBox 中的所有选中标记。

如果现在运行应用程序，就可以在左边选择一些项，再单击 Move 按钮，把它们移动到右边。这就结束了列表框的演示，下面看一个比较新的控件 ListView。

15.8 ListView 控件

图 15-12 显示了 Windows 中最常用的 ListView 控件。Windows 为显示文件和文件夹提供了许多其他方式，ListView 控件就包含其中一些选项，例如，显示大图标和详细视图等。

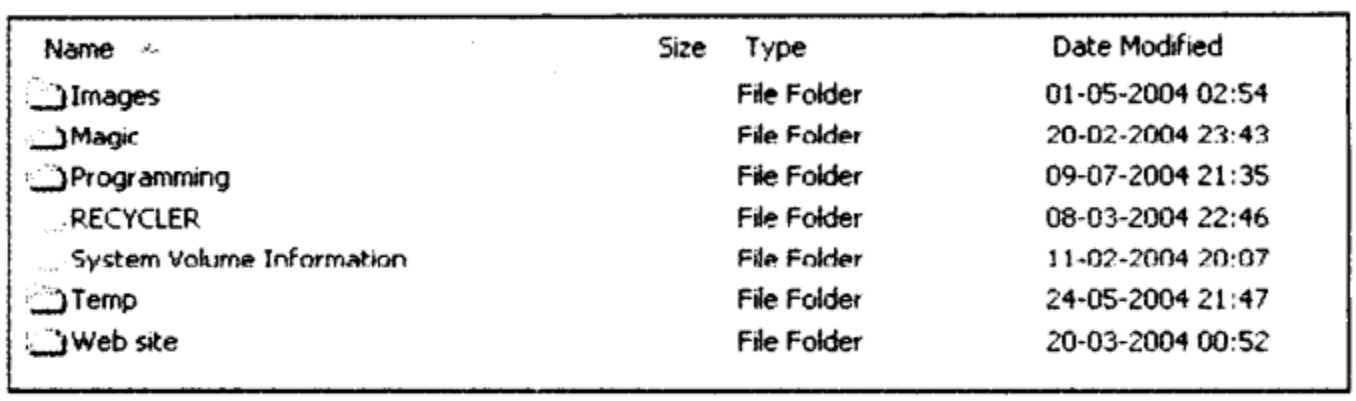


图 15-12

列表视图通常用于显示数据，用户可以对这些数据和显示方式进行某些控制。还可以把包含在控件中的数据显示为列和行(像网格那样)，或者显示为一列，或者显示为图标表示。最常用的列表视图就是图 15-12 中用于导航计算机中文件夹的视图。

ListView 控件是本章中最复杂的一个控件，它包括了超出本书范围的内容，这里仅编写一个示例，使用 ListView 控件中最重要的功能，为用户打下坚实的基础，将全面介绍可以使用的许多属性、事件和方法。本章还将讨论 ImageList 控件，它用于存储在 ListView 控件中使用的图像。

15.8.1 ListView 控件的属性

ListView 的属性如表 15-17 所示。

表 15-17

属 性	说 明
Activation	使用这个属性，可以控制用户在列表视图中激活选项的方式。可能的值如下： • Standard: 这个设置是用户为自己的计算机选择的值 • OneClick: 单击一个选项，激活它 • TwoClick: 双击一个选项，激活它
Alignment	这个属性可以控制列表视图中选项的对齐方式。有 4 个可能的值： • Default: 如果用户拖放一个选项，它将仍位于拖动前的位置 • Left: 选项与 ListView 控件的左边界对齐 • Top: 选项与 ListView 控件的顶边界对齐 • SnapToGrid: ListView 控件包含一个不可见的网格，选项都放在该网格中
AllowColumn Reorder	如果把这个属性设置为 true，就允许用户改变列表视图中列的顺序。如果这么做，就应确保即使改变了列的属性顺序，填充列表视图的例程也能正确插入选项

(续表)

属　性	说　明
AutoArrange	如果把这个属性设置为 true，选项会自动根据 Alignment 属性排序。如果用户把一个选项拖放到列表视图的中央，且 Alignment 是 Left，则选项会自动左对齐。只有在 View 属性是 LargeIcon 或 SmallIcon 时，这个属性才有意义
CheckBoxes	如果把这个属性设置为 true，列表视图中的每个选项会在其左边显示一个复选框。只有在 View 属性是 Details 或 List 时，这个属性才有意义
CheckedIndices CheckedItems	利用这两个属性分别可以访问索引和选项的集合，该集合包含列表中被选中的选项
Columns	列表视图可以包含列。通过这个属性可以访问列集合，通过该集合，可以增加或删除列
FocusedItem	这个属性包含列表视图中有焦点的选项。如果没有选择任何选项，该属性就为 null
FullRowSelect	这个属性为 true 时，单击一个选项，该选项所在的整行文本都会突出显示。如果该属性为 false，则只有选项本身会突出显示
GridLines	把这个属性设置为 true，则列表视图会在行和列之间绘制网格线。只有 View 属性为 Details 时，这个属性才有意义
HeaderStyle	可以控制列标题的显示方式，有 3 种样式： • Clickable: 列标题显示为一个按钮 • NonClickable: 列标题不响应鼠标单击 • None: 不显示列标题
HoverSelection	这个属性设置为 true 时，用户可以把鼠标指针放在列表视图的一个选项上，以选择它
Items	列表视图中的选项集合
LabelEdit	这个属性设置为 true 时，用户可以在 Details 视图下编辑第一列的内容
LabelWrap	如果这个属性是 true 时，标签就会自动换行，以便显示所有文本
LargeImageList	这个属性包含 ImageList，而 ImageList 包含大图像。这些图像可以在 View 属性为 LargeIcon 时使用
MultiSelect	这个属性设置为 true 时，用户可以选择多个选项
Scrollable	这个属性设置为 true 时，就显示滚动条
SelectedIndices SelectedItems	这两个属性分别包含选中索引和选项的集合
SmallImageList	当 View 属性为 SmallIcon 时，这个属性包含了 ImageList，其中 ImageList 包含了要使用的图像

(续表)

属 性	说 明
Sorting	可以让列表视图对它包含的选项排序，有 3 种可能的模式： • Ascending • Descending • None
StateImageList	ImageList 包含图像的蒙板，这些图像蒙板可用作 LargeImageList 和 SmallImageList 图像的覆盖图，表示定制的状态
TopItem	返回列表视图顶部的选项
View	列表视图可以用 4 种不同的基本模式显示其选项： • LargeIcon: 所有选项都在其旁边显示一个大图标(32x32)和一个标签 • SmallIcon: 所有选项都在其旁边显示一个小图标(16x16)和一个标签 • List: 只显示一列。该列可以包含一个图标和一个标签 • Details: 可以显示任意数量的列。只有第一列可以包含图标 • Tile：(只用于 Windows XP 和较新的 Windows 平台)显示一个大图标和一个标签，在图标的右边显示子项信息

15.8.2 ListView 控件的方法

对于像列表视图这样复杂的控件来说，专用的方法非常少。表 15-18 列出了这些方法。

表 15-18

方 法	说 明
BeginUpdate()	调用这个方法，将告诉列表视图停止更新，直到调用 EndUpdate()为止。当一次插入多个选项时使用这个方法很有用，因为它会禁止视图闪烁，大大提高速度
Clear()	彻底清除列表视图，删除所有的选项和列
EndUpdate()	在调用 BeginUpdate 之后调用这个方法。在调用这个方法时，列表视图会显示其所有选项
EnsureVisible()	在调用这个方法时，列表视图会滚动，以显示指定索引的选项
GetItemAt()	返回列表视图中位于 x,y 位置的选项

15.8.3 ListView 控件的事件

表 15-19 列出了要处理的 ListView 控件的事件。

表　15-19

事　　件	说　　明
AfterLabelEdit	在编辑了标签后，引发该事件
BeforeLabelEdit	在用户开始编辑标签前，引发该事件
ColumnClick	在单击一个列时，引发该事件
ItemActivate	在激活一个选项时，引发该事件

15.8.4　ListViewItem

列表视图中的选项总是 ListViewItem 类的一个实例。ListViewItem 包含要显示的信息，如文本和图标的索引。ListViewItems 对象有一个 SubItems 属性，其中包含另一个类 ListViewSubItem 的实例。如果 ListView 控件处于 Details 或 Tile 模式下，这些子选项就会显示出来。每个子选项表示列表视图中的一列。子选项和主选项之间的区别是，子选项不能显示图标。

通过 Items 集合把 ListViewItems 添加到 ListView 中，通过 ListViewItem 上的 SubItems 集合把 ListViewSubItems 添加到 ListViewItem 中。

15.8.5　ColumnHeader

要使列表视图显示列标题，需要把类 ColumnHeader 的实例添加到 ListView 的 Columns 集合中。当 ListView 控件处于 Details 模式下时，ColumnHeaders 为要显示的列提供一个标题。

15.8.6　ImageList 控件

ImageList 控件提供了一个集合，可以用于存储在窗体的其他控件中使用的图像。可以在图像列表中存储任意大小的图像，但在每个控件中，每个图像的大小必须相同。对于 ListView，则需要两个 ImageList 控件，才能显示大图像和小图像。

ImageList 是本章介绍的第一个不在运行期间显示自身的控件。在把它拖放到正在开发的窗体上时，它并不是放在窗体上，而是放在它的下面，其中包含所有的这类组件。这个功能可以防止不是用户界面一部分的控件把窗体设计器弄乱。这个控件的处理方式与其他控件相同，但不能把它移动到窗体上。

可以在设计和运行期间给 ImageList 添加图像。如果知道在设计期间需要显示哪些图像，就可以单击 Images 属性右边的按钮，添加这些图像。这会打开一个对话框，在该对话框中，可以浏览要插入的图像。如果选择在运行期间添加图像，就可以通过 Images 集合添加它们。

学习使用 ListView 控件及其相关的图像列表的最好方式是利用一个示例。下面的示例将创建一个对话框，其中有一个 ListView 和两个 ImageList。ListView 显示硬盘上的文件和文件夹。为简单起见，我们不提取文件和文件夹中的正确图标，而使用文件夹的标准文件夹图标和文件的文本图标。

双击文件夹，就可以浏览文件夹树，后退按钮可以在文件夹树中向上移动。5 个单选按钮用于在运行期间改变列表视图的模式。如果双击了一个文件，就可以执行它。

试一试：使用 ListView 控件

与往常一样，首先创建用户界面：

(1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新 Windows 应用程序 ListView。

(2) 在窗体上添加一个列表视图、一个按钮、一个标签和一个组框。然后在组框中添加 5 个单选按钮，此时窗体应如图 15-13 所示。要设置标签控件的宽度，应把其 AutoSize 属性设置为 False。把标签控件设置得与列表视图一样宽。

(3) 如图 15-13 所示命名控件。ListView 不在图中显示其名称，这里添加了一个选项，显示其名称。读者不需要这么做。

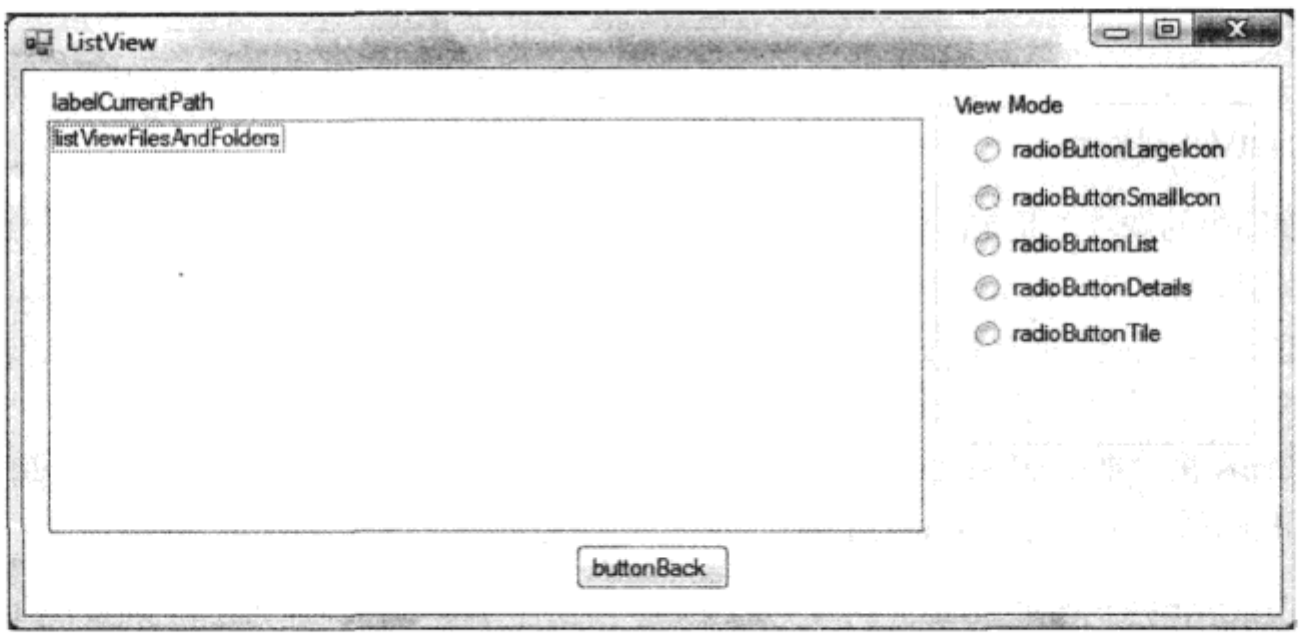


图 15-13

(4) 把单选按钮的 Text 属性值改为其名称(但控件名称除外)，把窗体的 Text 属性设置为 ListView。

(5) 清空标签的 Text 属性。

(6) 在工具箱中双击 ImageList 控件的图标，在窗体中添加两个图像列表。ImageList 控件在工具箱的 Components 区域中。把它们重新命名为 imageListSmall 和 imageListLarge。

(7) 把图像列表 imageListLarge 的 Size 属性值改为 32，32。

(8) 单击 imageListLarge 图像列表的 Images 属性右边的按钮，打开一个对话框，从中可以浏览要插入的图像。

(9) 单击 Add，浏览本章代码的 ListView 文件夹。这些文件是 Folder 32x32.ico 和 Text 32x32.ico。

(10) 确保文件夹图标位于列表顶部。

(11) 对另一个图像列表 imageListSmall 重复第(8)、(9)步，选择 16×16 版本的图标。

(12) 把单选按钮 radioButtonDetails 的 Checked 属性设置为 true。

(13) 设置列表视图的属性，如表 15-20 所示。

表 15-20

属 性	值
LargeImageList	imageListLarge
SmallImageList	imageListSmall
View	Details

添加事件处理程序

这是我们的用户界面，下面可以添加代码了。首先，需要一个字段，以包含前面浏览的文件夹，在单击后退按钮时，就可以返回这些文件夹。我们将存储文件夹的绝对路径，所以选择使用 StringCollection：

```
partial class Form1 : Form
{
   private System.Collections.Specialized.StringCollection folderCol;
```

在窗体设计器中没有创建任何列标题，现在要使用 CreateHeadersAndFillListView()方法在代码中创建：


```
private void CreateHeadersAndFillListView()
{
   ColumnHeader colHead;

   colHead = new ColumnHeader();
   colHead.Text = "Filename";
   listViewFilesAndFolders.Columns.Add(colHead); // Insert the header

   colHead = new ColumnHeader();
   colHead.Text = "Size";
   listViewFilesAndFolders.Columns.Add(colHead); // Insert the header

   colHead = new ColumnHeader();
   colHead.Text = "Last accessed";
   listViewFilesAndFolders.Columns.Add(colHead); // Insert the header
}
```

代码段 Chapter15\ListView\Form1.cs.

先声明一个变量 colHead，用于创建 3 个列标题。这 3 个标题都是用 new 关键字创建的，在把它添加到 ListView 的 Columns 集合中之前，将文本赋给它。

在第一次显示窗体时，进行的最后一个初始化工作是用硬盘上的文件和文件夹填充列表视图。这通过另一个方法来完成：


```
private void PaintListView(string root)
   {
     try
     {
        ListViewItem lvi;
        ListViewItem.ListViewSubItem lvsi;

        if (string.IsNullOrEmpty(root))
          return;

        DirectoryInfo dir = new DirectoryInfo(root);
        DirectoryInfo[] dirs = dir.GetDirectories();
        FileInfo[] files = dir.GetFiles();
```

```
            listViewFilesAndFolders.Items.Clear();
            labelCurrentPath.Text = root;
            listViewFilesAndFolders.BeginUpdate();

            foreach (DirectoryInfo di in dirs)
            {
              lvi = new ListViewItem();
              lvi.Text = di.Name;
              lvi.ImageIndex = 0;
              lvi.Tag = di.FullName;

              lvsi = new ListViewItem.ListViewSubItem();
              lvsi.Text = "";
              lvi.SubItems.Add(lvsi);

              lvsi = new ListViewItem.ListViewSubItem();
              lvsi.Text = di.LastAccessTime.ToString();
              lvi.SubItems.Add(lvsi);
              listViewFilesAndFolders.Items.Add(lvi);
            }
            foreach (FileInfo fi in files)
            {
              lvi = new ListViewItem();
              lvi.Text = fi.Name;
              lvi.ImageIndex = 1;
              lvi.Tag = fi.FullName;

              lvsi = new ListViewItem.ListViewSubItem();
              lvsi.Text = fi.Length.ToString();
              lvi.SubItems.Add(lvsi);

              lvsi = new ListViewItem.ListViewSubItem();
              lvsi.Text = fi.LastAccessTime.ToString();
              lvi.SubItems.Add(lvsi);
              listViewFilesAndFolders.Items.Add(lvi);
            }

            listViewFilesAndFolders.EndUpdate();
          }
          catch (System.Exception err)
          {
            MessageBox.Show("Error: " + err.Message);
          }
        }
```

代码段 Chapter15\ListView\Form1.cs.

示例的说明

在第一个 foreach 块中，对 ListView 控件调用了 BeginUpdate()。ListView 控件上的 BeginUpdate() 方法告诉 ListView 控件，停止更新其可见区域，直到调用了 EndUpdate()为止。如果没有调用这个方法，列表视图的填充就会进行得更加缓慢，列表可能在填充选项时闪烁。在第二个 foreach 块的后面调用了 EndUpdate()，就可以使 ListView 控件显示出填充到它里面的内容。

这两个 foreach 块包含了我们感兴趣的代码。首先创建 ListViewItem 的一个新实例，再把 Text 属性设置为要插入的文件名或文件夹名。ListViewItem 的 ImageIndex 表示其中一个 ImageList 中的选项索引。所以两个 ImageList 中的图标有相同的索引是非常重要的。使用 Tag 属性保存文件夹和文件的完全限定路径，在用户双击选项时，将使用该路径。

然后创建两个子选项，将要显示的文本赋给这两个子选项，再把它们添加到 ListViewItem 的 SubItems 集合中。

最后，把 ListViewItem 添加到 ListView 的 Items 集合中。ListView 非常聪明，知道如果视图模式不是 Details，就应忽略子选项。所以，现在无论视图模式是什么，都可以增加子选项。

注意代码的某些方面没有讨论，即实际获取文件信息的代码行：

```
// Get information about the root folder.
DirectoryInfo dir = new DirectoryInfo(root);
// Retrieve the files and folders from the root folder.
DirectoryInfo[] dirs = dir.GetDirectories();
FileInfo[] files = dir.GetFiles();
```

这些代码使用 System.IO 名称空间中的类访问文件，所以需要在代码顶部的 using 区域添加如下代码：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using System.IO;
```

第 24 章将详细介绍文件访问和 System.IO 名称空间，但现在应明白，DirectoryInfo 对象上的 GetDirectories()方法返回一个对象集合，它们表示所查看的目录下的文件夹，GetFiles()方法返回一个对象集合，它们表示当前目录下的文件。可以迭代这些集合，如上面的代码所示，使用对象的 Name 属性返回相关目录或文件的名称，创建一个 ListViewItem 来保存这个字符串。

剩下的就是列表视图应显示根文件夹，为此，在窗体的构造函数中调用两个函数。同时用根文件夹实例化 folderCol StringCollection 字符串集合：

```
InitializeComponent();

folderCol = new System.Collections.Specialized.StringCollection();
CreateHeadersAndFillListView();
PaintListView(@"C:\");
folderCol.Add(@"C:\");
```

为了允许用户通过双击 ListView 中的选项来浏览文件夹，需要订阅 ItemActivate 事件。在设计器中选择 ListView，在 Properties 窗口的 Events 列表中双击 ItemActivate 事件。

对应的事件处理程序如下所示：

```
private void listViewFilesAndFolders_ItemActivate(object sender, EventArgs e)
    {
       System.Windows.Forms.ListView lw = (System.Windows.Forms.ListView)sender;
       string filename = lw.SelectedItems[0].Tag.ToString();
```

```
if (lw.SelectedItems[0].ImageIndex != 0)
{
   try
   {
      System.Diagnostics.Process.Start(filename);
   }
   catch {return;}
}
else
{
   PaintListView(filename);
   folderCol.Add(filename);
}}
```

选中项的 Tag 包含被双击的文件或文件夹的完全限定路径。索引为 0 的图像是一个文件夹，所以查看索引就可以确定哪个选项是文件，哪个选项是文件夹。如果选项是一个文件，就试着加载它。如果选项是一个文件夹，就通过新文件夹调用 PaintListView()，再把新文件夹添加到 folderCol 集合中。

在讨论单选按钮前，先给 Back 按钮添加 Click 事件，提供完整的浏览功能。双击该按钮，为事件处理程序添加如下代码：

```
private void buttonBack_Click(object sender, EventArgs e)
{
   if (folderCol.Count > 1)
   {
      PaintListView(folderCol[folderCol.Count-2].ToString());
      folderCol.RemoveAt(folderCol.Count-1);
   }
   else
      PaintListView(folderCol[0].ToString());
}
```

如果 folderCol 集合中有多个选项，我们就不在浏览器的根文件夹下，对该路径调用 PaintListView()，进入上面的文件夹。folderCol 集合中的最后一个选项是当前文件夹，这就是需要第二次提取最后一个选项的原因。然后删除集合中的最后一个选项，使前面一个选项成为当前文件夹。如果该集合中只有一个选项，就只需对该选项调用 PaintListView()。

剩下的是修改列表视图的查看类型。双击每个单选按钮，添加如下代码：

可从 wrox.com 下载源代码

```
private void radioButtonLargeIcon_CheckedChanged(object sender, EventArgs e)
{
   RadioButton rdb = (RadioButton)sender;
   if (rdb.Checked)
      this.listViewFilesAndFolders.View = View.LargeIcon;
}

private void radioButtonList_CheckedChanged(object sender, EventArgs e)
  {
     RadioButton rdb = (RadioButton)sender;
     if (rdb.Checked)
        this.listViewFilesAndFolders.View = View.SmallIcon;
```

```
}

private void radioButtonSmallIcon_CheckedChanged(object sender, EventArgs e)
{
   RadioButton rdb = (RadioButton)sender;
   if (rdb.Checked)
      this.listViewFilesAndFolders.View = View.List;
}

private void radioButtonDetails_CheckedChanged(object sender, EventArgs e)
{
   RadioButton rdb = (RadioButton)sender;
   if (rdb.Checked)
      this.listViewFilesAndFolders.View = View.Details;
}

private void radioButtonTile_CheckedChanged(object sender, EventArgs e)
{
   RadioButton rdb = (radioButton)sender;
   if (rdb.Checked)
      this.listViewFilesAndFolders.View = View.Tile;
}
```

代码段 Chapter15\ListView\Form1.cs.

检查单选按钮，看看是否已将其改为 Checked。如果是，就设置 ListView 的 View 属性。这就是 ListView 示例。运行它，将得到如图 15-14 所示的结果。

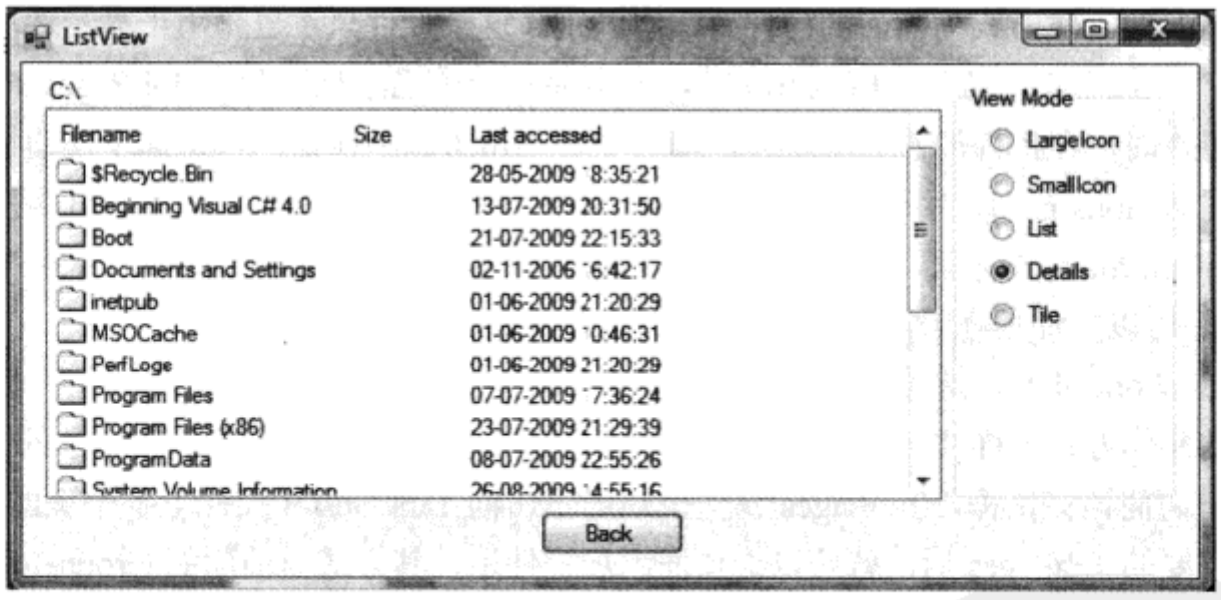


图　15-14

15.9　TabControl 控件

TabControl 提供了一种简单的方式，可以把对话框组织为合乎逻辑的部分，以便根据控件顶部的选项卡来访问。TabControl 包含 TabPages，TabPages 的工作方式与 GroupBox 控件非常类似，也是把控件组合在一起，但它们复杂。

TabControl 控件的使用是非常简单的。可以在控件的 TabPage 对象集合中添加任意数量的选项

卡，再把要显示的控件拖放到各个页面上。

15.9.1 TabControl 控件的属性

TabControl 的属性(如表 15-21 所示)一般用于控制 TabPage 对象容器的外观，特别是显示的选项卡的外观。

表 15-21

属 性	说 明
Alignment	控制选项卡在选项卡控件的什么位置显示。默认位置为控件的顶部
Appearance	控制选项卡的显示方式。选项卡可以显示为一般的按钮或带有平面样式
HotTrack	如果这个属性设置为 true，则当鼠标指针滑过控件上的选项卡时，其外观就会改变
Multiline	如果这个属性设置为 true，就可以有几行选项卡
RowCount	返回当前显示的选项卡行数
SelectedIndex	返回或设置选中选项卡的索引
SelectedTab	返回或设置选中的选项卡。注意这个属性在 TabPages 的实例上使用
TabCount	返回选项卡的总数
TabPages	这是控件中的 TabPage 对象集合。使用这个集合可以添加和删除 TabPage 对象

15.9.2 使用 TabControl 控件

TabControl 的工作方式与前面的控件有一些区别。这个控件只不过是用于显示选项卡的选项卡页面的容器。在工具箱中双击 TabControl 时，就会显示一个已添加了两个 TabPages 的控件。

选择该控件时，在控件的右上角就会出现一个带三角形的小按钮。单击这个按钮，就会打开一个小窗口，即 Actions 窗口，用于访问控件的所选属性和方法。Visual Studio 中的许多控件都有这个特性，但 TabControl 是本章第一个允许在 Actions 窗口中执行某些操作的控件。使用 TabControl 的 Actions 窗口，可以方便地在设计期间添加和删除 TabPages。

上面给 TabControl 添加选项卡页的过程可以让用户很快使用和运行该控件。另一方面，如果要改变选项卡的操作方式或样式，就应使用 TabPages 对话框；在选择 Properties 窗口中的 TabPages 时，可以通过按钮访问该对话框。TabPages 属性也是用于访问 TabControl 控件上各个页面的集合。

添加了需要的 TabPages 后，就可以给页面添加控件了，其方式与前面的 GroupBox 相同。下面创建一个示例，说明该控件的基本内容。

试一试：使用标签页

按照下面的步骤创建一个 Windows 应用程序，说明如何把控件放在选项卡控件的不同页面上：

(1) 在 C:\BegVCSharp\Chapter15 目录中创建一个新的 Windows 应用程序 TabControl。

(2) 把一个 TabControl 控件从工具箱拖放到窗体上。与 GroupBox 一样，TabControl 在工具箱的 Containers 选项卡中。

(3) 找到 TabPages 属性，选择它后，单击它右边的按钮，打开 TabPage Collection Editor。

(4) 把选项卡页的 Text 属性分别改成 Tab One 和 Tab Two。单击 OK，关闭该对话框。

(5) 单击控件顶部的选项卡，选择要处理的选项卡。选择标有 Tab One 的选项卡。在控件上拖放按钮。确保把该按钮放在 TabControl 的框架中。如果把它放在框架的外部，则该按钮就会放在窗体上，而不是选项卡控件上。

(6) 将按钮的名称改为 buttonShowMessage，将其 Text 属性改为 Show Message。

(7) 单击 Text 属性为 Tab Two 的选项卡，把一个文本框控件拖放到 TabControl 上。为这个控件命名。

(8) 这两个选项卡应如图 15-15 和 15-16 所示。

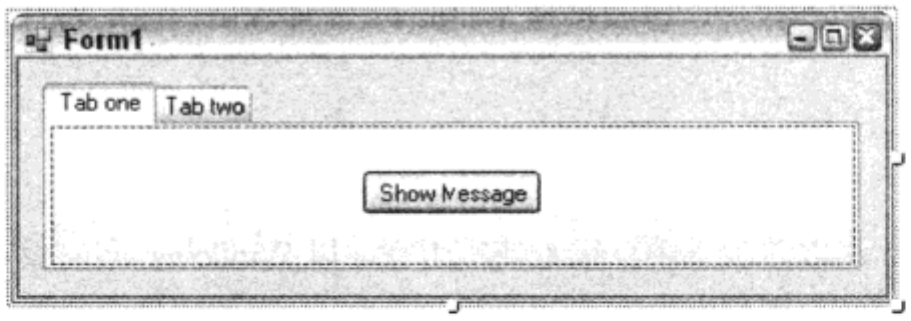


图　15-15

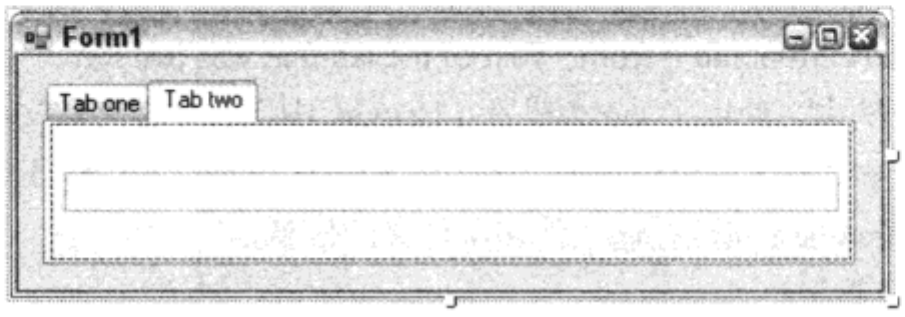


图　15-16

下面准备访问控件。如果运行代码，就会看到选项卡正确显示出来了。为了说明选项卡控件的用法，剩下要做的工作是添加一些代码，以便在用户单击一个选项卡上的 Show Message 按钮时，在另一个选项卡页中输入的文本将显示在消息框中。首先，双击第一个选项卡上的按钮，为 Click 事件添加一个处理程序，再添加下述代码：

```
private void buttonShowMessage_Click(object sender, EventArgs e)
{
   MessageBox.Show(textBoxMessage.Text);
}
```

示例的说明

在选项卡页上访问一个控件，与访问窗体上的其他控件是一样的。获取文本框的 Text 属性，在消息框中显示它。

本章前面介绍过，在窗体中一次只能选择一个单选按钮(除非把它们放在组框中)。TabPages 与组框的工作方式完全相同，所以可以在不同的选项卡上放置多组单选按钮，而不需要使用组框。如 buttonShowMessage_Click 方法所示，还可以访问位于其他选项卡上的控件。

要能处理选项卡控件，最后要注意的是如何确定当前显示的是哪个选项卡。这可以使用两个属性：SelectedTab 和 SelectedIndex，顾名思义，SelectedTab 返回 TabPage 对象，如果没有选择选项卡，

就返回 null。而 SelectedIndex 返回选项卡的索引，如果没有选择选项卡，就返回 -1。练习题(2)将使用这些属性。

15.10 小结

本章介绍了创建 Windows 应用程序时最常用的一些控件，并讨论了如何使用它们创建简单而强大的用户界面。还论述了这些控件的属性和事件，列出了使用它们的示例，解释了如何为控件的特定事件添加处理程序。

第 16 章将讨论更复杂的控件，以及创建 Windows 窗体应用程序的特性。

15.11 练习

(1) 在 Visual Studio 的以前版本中，很难使应用程序以 Windows 当前版本的样式显示其控件。本练习要在 Windows 窗体应用程序中，找到新的 Windows 窗体项目中启用各种可视样式的位置。试着启用和禁用该样式，看看这些操作对窗体上的控件有什么影响。

(2) 修改 TabControl 示例，方式是添加几个选项卡页，在消息框中显示文本 You changed the current tab to <Text of the current tab > from <Text of the tab that was just left>。

(3) 在 ListView 示例中，使用了 tag 属性在 ListView 中保存文件夹和文件的完全限定路径。修改这个操作，创建一个派生于 ListViewItem 的新类，使用这个新类的实例作为 ListView 中的项。在新类中使用 FullyQualifiedPath 属性存储文件和文件夹的信息。

附录 A 给出了练习答案。

15.12 本章要点

主 题	重 要 概 念
Label 控件	使用 Label 和 LinkedLabel 控件给用户显示信息
Button 控件	使用 Button 控件和相应的 Click 事件，让用户告诉应用程序他们要进行什么操作
TextBox 控件	使用 TextBox 和 RichTextBox 控件，让用户输入纯文本或格式化文本
选项控件	区分 CheckBox 和 RadioButton 及其用法。还学习了如何把这两个控件组合到 GroupBox 控件中，以及这么做对控件有什么影响
ListBox 控件	使用 CheckedListBox 提供列表，用户可以单击复选框，从该列表中选择选项。还学习了如何使用更常见的 ListView 控件，提供与 CheckedListBox 控件类似的列表，但没有复选框
ListView 控件	使用 ListView 和 ImageList 控件提供一个列表，让用户以不同的方式查看
TabControl 控件	最后学习了如何使用 TabControl 把控件组合到同一个窗体的不同页面上，用户可以随意从这些页面上选择控件

第16章

Windows 窗体的高级功能

本章内容：

- 使用 3 个常见控件创建外观多样的菜单、工具栏和状态栏
- 创建 MDI 应用程序
- 创建自己的控件

第 15 章介绍了 Windows 应用程序开发中一些最常用的控件。使用第 15 章介绍的控件，可以创建界面十分友好的对话框，但 Windows 应用程序的用户界面很少只包含一个对话框。这些应用程序使用单一文档界面(Single Document Interface，SDI)或者多文档界面(Multiple Document Interface，MDI)。这两种类型的应用程序通常会大量使用菜单和工具栏，本章就讨论它们。

.NET Framework 添加了 WPF(Windows Presentation Foundation)后，引入了几个新类型的 Windows 应用程序，详见第 25 章。

像第 15 章那样，本章在介绍控件时，首先介绍菜单控件，再介绍工具栏，说明如何把工具栏上的按钮与特定的菜单项链接起来，或把特定的菜单项与工具栏上的按钮链接起来。接着创建 SDI 和 MDI 应用程序，主要讨论 MDI 应用程序，因为 SDI 应用程序基本上是 MDI 应用程序的子集。

到目前为止，仅使用了.NET Framework 提供的控件。这些控件非常强大，提供了许多功能，但有时候使用它们还是不够。所以需要创建定制控件，本章的最后介绍如何创建定制控件。

16.1 菜单和工具栏

有几个Windows 应用程序不包含菜单或工具栏？这个数字可能接近于 0。在为 Windows 操作系统编写的应用程序中，菜单和工具栏可能是不可或缺的重要部分。为了帮助用户创建应用程序的菜单，Visual Studio 2010 提供了两个控件，使用它不必做太多的工作，就可以快速创建外观类似于 Visual

Studio 的菜单和工具栏。

16.1.1 两个实质一样的控件

下面要介绍的两个控件是 Visual Studio 2005 的新增控件，它们为开发人员和专业人士提供了许多强大的功能。仍旧在编写定制绘图处理程序和购买第三方组件的用户，使用这两个控件可以创建出工具栏和菜单具有专业化外观的应用程序。以前需要几个星期才能创建出的应用程序，现在变成在区区数秒内即可完成的简单任务。

我们要使用的控件包含在后缀为Strip 的控件系列中，分别是ToolStrip、MenuStrip和StatusStrip。StatusStrip 在本章后面介绍。从它们最纯粹的形式来看，ToolStrip 和 MenuStrip 实际上是相同的控件，因为 MenuStrip 直接派生于 ToolStrip。也就是说，ToolStrip 可以做的工作，MenuStrip 也能完成。显然，它们两个一起完成会更好。

16.1.2 使用 MenuStrip 控件

除了 MenuStrip 控件之外，还有许多控件可用于填充菜单。3 个最常见的控件是 ToolStrip MenuItem、ToolStripDropDown 和ToolStripSeparator。这些控件表示查看菜单或工具栏中某一项的特定方式。ToolStripMenuItem 表示菜单中的一项，ToolStripDropDown 表示单击一项，就会显示包含其他项目的一个列表，ToolStripSeparator 表示菜单或工具栏中的水平或垂直分隔线。

在讨论完 MenuStrip 之后，还要讨论另一种菜单 ContextMenuStrip。当用户右击一项时，关联菜单就会显示出来，它通常显示与该项相关的信息。

在下面的示例中，将创建本章的第一个示例。

试一试：在 5 秒内创建具有专业外观的菜单

第一个示例只是尝试一下，如果要创建外观非常标准的菜单，就应了解新控件的其他精彩方面。

(1) 在 C:\BegVCSharp\Chapter16 目录中创建一个新 Windows 应用程序，命名为 Professional Menus。

(2) 从工具箱中，把一个 MenuStrip 控件的实例拖放到设计界面上。

(3) 在对话框的顶部，单击 MenuStrip 右边的三角形，显示 Actions 窗口。

(4) 单击菜单右上角的三角形，再单击 Insert Standard Items 链接。

这就完成了。如果向下拖动 File 菜单，就会看到许多熟悉的选项，包括快捷键和图标。现在该菜单还没有什么功能——必须在其中填充内容。还可以编辑菜单，请继续往下看。

16.1.3 手工创建菜单

从工具箱中把 MenuStrip 控件拖放到设计界面上时，该控件会位于窗体和控件盘上，而且可以直接在窗体上编辑。要创建新的菜单项，只需把指针放在 Type Here 框上。

在突出显示的框中输入菜单的标题，在要用作该菜单项快捷键字符的字母前面加上一个宏字符(&)，在菜单项中，该字母显示为下划线形式，可以按下 Alt 键和该字母键来选择该菜单项。

注意，可以在一个菜单中用同一快捷键字符创建好几个菜单项，其规则是每个弹出菜单只能将该字符使用一次(例如，在 Files 弹出菜单中使用一次，在 View 菜单中使用一次等)。如果不小心把同一个快捷键字符赋予了同一个弹出菜单中的多个菜单项，则只有最靠近控件顶部的那个菜单项会

响应该字符。

选择一项，控件就会自动在当前项的下面和右边显示一些项。给这两个控件输入标题，就创建了与开始时选中的项相关的一个新项，这就是创建下拉菜单的方式。

要创建水平线，把菜单分成组，必须使用 ToolStripSeparator 控件，而不是 ToolStripMenuItem 控件，但不需要插入另一个控件。还可以键入一个短横线(-)，作为该项的标题。Visual Studio 会自动假定该项是一个分隔符，改变控件的类型。

下面的示例要创建一个菜单，但不使用 Visual Studio 生成其上的各项。

试一试：从头创建菜单

在这个示例中，要从头创建 File 和 Help 菜单，Edit 和 Tools 菜单留给读者创建。

(1) 创建一个新的 Windows 应用程序项目，命名为 Manual Menus，保存在 C:\BegVCSharp\Chapter16 文件夹中。

(2) 把 MenuStrip 控件从工具栏拖放到设计界面上。

(3) 单击 MenuStrip 控件的 Type Here 文本区域，键入&File，按下回车键。

(4) 在 File 项下面的文本区域键入如下内容：

- &New
- &Open
- -
- &Save
- Save &As
- -
- &Print
- Print Preview
- -
- E&xit

注意 Visual Studio 会自动把短线变成分隔各元素的线条。

(5) 单击 Files 右边的文本区域，键入&Help。

(6) 在 Help 项下面的文本区域中键入以下内容：

- Contents
- Index
- Search
- -
- About

(7) 下面返回到 File 菜单，为菜单项设置快捷键。为此，选择要设置的菜单项，在 Properties 面板中找到 ShortcutKeys 属性，单击向下箭头，打开一个小窗口，在该窗口中可以设置与菜单项相关的键组合。由于这个菜单是一个标准菜单，因此应使用标准的键组合。如果要创建其他键组合，可以自由选择其他键组合。File 菜单中的 ShortcutKeys 属性设置如表 16-1 所示。

表 16-1

菜单项名称	属 性 和 值
&New	Ctrl + N
&Open	Ctrl + O
&Save	Ctrl + S
&Print	Ctrl + P

(8) 现在完成图像的处理。在 File 菜单中选择 New 菜单项，在属性面板的 Image 属性左边单击省略号(...)，打开 Select Resource 对话框。

在创建这些菜单时，最困难的地方是获取要显示的图像。在本例中，可以从 www.wrox.com 上下载本书的源代码来获得这些图像。但一般情况下应自己绘制图像，或者以其他方式获得。

(9) 由于目前项目中没有资源，所以 Entry 列表框是空的。单击 Import，这个示例的图像在本书源代码的 Chapter16\Manual Menus\Images 文件夹下。选择该文件夹下的所有文件，单击 Open。现在编辑的是 New 菜单项，所以在 Entry 列表中选择 New 图像，单击 OK 按钮。

(10) 对 Open、Save、Save As、Print 和 Print Preview 按钮的按钮重复第(9)步。

(11) 运行项目，注意可以单击 File 菜单，或者按下 Alt+F 组合键，来选择该菜单。Help 菜单可通过 Alt+H 组合键来选择。

16.1.4 ToolStripMenuItem 控件的其他属性

ToolStripMenuItem 有另外几个属性，在创建菜单时应了解这些属性。表 16-2 并不完整，如果需要完整的列表，可参阅.NET Framework SDK 文档说明。

表 16-2

属 性	说 明
Checked	表示菜单是否被选中
CheckOnClick	这个属性是 true 时，如果菜单项文本左边的复选框没有打上标记，就打上标记，如果该复选框已打上了标记，就去除该标记，否则，该标记就被一个图像替代，使用 Checked 属性确定菜单项的状态
Enabled	把 Enabled 设置为 false ，菜单项就会灰显，不能被选中
DropDownItems	这个属性返回一个项集合，用作与菜单项相关的下拉菜单

16.1.5 给菜单添加功能

下面就可以生成与 Visual Studio 的外观相同的菜单了。只需在单击菜单时执行某些操作即可。显然，这个操作取决于用户。在下面的示例中，将在上一个示例的基础上创建一个非常简单的示例。

为了响应用户做出的选择，就应为 ToolStripMenuItems 发送的两个事件之一提供处理程序。见表 16-3。

表　16-3

事 件 名 称	说　　明
Click	在用户单击菜单项时，引发该事件。大多数情况下这就是要响应的事件
CheckedChanged	当单击带 CheckOnClick 属性的菜单项时，引发这个事件

下面扩展上一个示例 Manual Menus，给对话框添加一个文本框，执行几个事件处理程序。在 File 和 Help 之间再添加一个菜单 Format。在下载代码中，这个项目的名称是 Extended Manual Menus。

试一试：处理菜单事件

(1) 继续使用上一个示例创建的窗体，把一个 RichTextBox 拖放到设计界面上，把它的名称改为 richTextBoxText，其属性 Dock 设置为 Fill。

(2) 选择 MenuStrip，在 Help 菜单项旁边的文本区域中输入 Format，按下回车键。

(3) 选择 Format 菜单项，把它拖动到 Files 和 Help 之间。

(4) 给 Format 菜单添加一个菜单项 Show Help Menu。

(5) 把 Show Help Menu 菜单项的 CheckOnClick 属性设置为 true，将其 Checked 属性设置为 true。

(6) 选择 showHelpMenuToolStripMenuItem，在属性面板的 Events 列表中双击 CheckedChanged 事件，为该事件添加处理程序。

(7) 为事件处理程序添加如下代码：

```
private void showHelpMenuToolStripMenuItem_CheckedChanged(object sender,
EventArgs e)
    {
     ToolStripMenuItem item = (ToolStripMenuItem)sender;
     helpToolStripMenuItem.Visible = item.Checked;
    }
```

(8) 双击 newToolStripMenuItem、saveToolStripMenuItem 和 openToolStripMenuItem，在设计视图中双击 ToolStripMenuItem，会把 Click 事件添加到代码中。输入下面的代码：

```
private void newToolStripMenuItem_Click(object sender, EventArgs e)
    {
      richTextBoxText.Text = "";
    }

    private void openToolStripMenuItem_Click(object sender, EventArgs e)
    {
      try
      {
        richTextBoxText.LoadFile(@"Example.rtf");
      }
      catch { }
    }

    private void saveToolStripMenuItem_Click(object sender, EventArgs e)
    {
      try
      {
```

```
            richTextBoxText.SaveFile("Example.rtf");
        }
        catch { }
    }
```

(9) 运行应用程序，单击 Show Help Menu，Help 菜单就会消失或出现，这取决于 Checked 属性的状态。该程序应可以打开、保存和清除文本框中的文本。

示例的说明

先处理 ShowHelpMeunToolStripMenuItem_CheckedChanged 事件。如果 Checked 属性是 true，这个事件的处理程序就应把 MenuItemHelp 的 Visible 属性设置为 true，否则就设置为 false。这会使该菜单项变成 Help 菜单的切换按钮。

最后，3 个 Click 事件处理程序分别清除 RichTextBox 中的文本，把其中的内容保存到预定义的文件中，打开该文件。注意 Click 和 CheckedChanged 事件是相同的，因为它们都在用户单击菜单项时处理发生的事件，但菜单项的操作并不相同，应根据菜单项的作用来确定。

16.2 工具栏

通过菜单可以访问应用程序中的大多数功能，把一些菜单项放在工具栏中和放在菜单中有相同的作用。工具栏提供了单击访问程序中常用功能(如 Open 和 Save)的方式。

图 16-1 显示了写字板中的一部分工具栏。

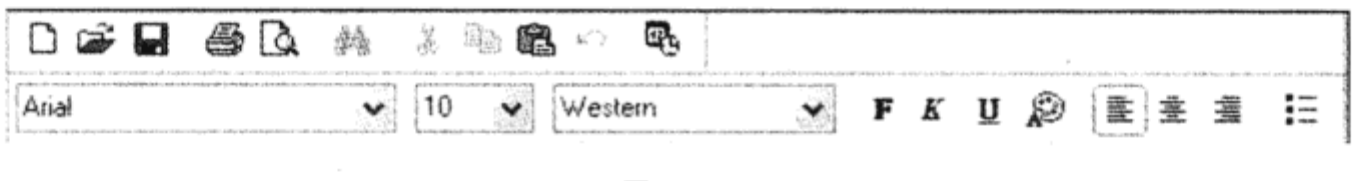


图 16-1

工具栏上的按钮通常包含图片，不包含文本，但它可以既包含图片又包含文本。例如 Word 中的工具栏按钮就不包含文本(参见图 16-1)。包含文本的工具栏按钮有 Internet Explorer 中的工具栏。除了按钮之外，工具栏上偶尔也会有组合框和文本框。如果把鼠标指针停留在工具栏的一个按钮上，就会显示一个工具提示，给出该按钮的用途信息，特别是只显示图标时，这是很有帮助的。

ToolStrip 与 MenuStrip 一样，也具有专业化的外观和操作方式。在用户查看工具栏时，希望能把它移动到自己希望的任意位置上。ToolStrip 就允许用户这么做。

第一次把 ToolStrip 添加到窗体的设计界面上时，它看起来非常类似于前面的 MenuStrip，但存在两个区别：ToolStrip 的最左边有 4 个垂直排列的点，这与 Visual Studio 中的菜单相同。这些点表示工具栏可以移动，也可以停靠在父应用程序窗口中。第二个区别是在默认情况下，工具栏显示的是图像，而不是文本，所以工具栏上项的默认控件是按钮。工具栏显示的下拉菜单允许选择菜单项的类型。

ToolStrip 与 MenuStrip 完全相同的一个方面是，Action 窗口包含 Insert Standard Items 链接。单击这个链接，不会得到与 MenuStrip 相同的菜单项数，而会获得 New、Open、Save、Print、Cut、Copy、Paste 和 Help 等按钮。下面不像前面那样完成一个完整的示例，而是先介绍 ToolStrip 的一些属性和用于填充它的控件。

16.2.1　ToolStrip 控件的属性

ToolStrip 控件的属性控制和管理着控件的显示位置和显示方式。这个控件是前面介绍的 MenuStrip 控件的基础，所以它们具有许多相同的属性。表 16-4 只列出了几个比较重要的属性，如果需要完整的列表，可参阅.NET Framework SDK 文档说明。

表　16-4

属　性	说　明
GripStyle	控制 4 个垂直排列的点是否显示在工具栏的最左边。隐藏手柄后，用户就不能移动工具栏了
LayoutStyle	控制工具栏上的项如何显示，默认为水平显示
Items	包含工具栏中所有项的集合
ShowItemToolTip	确定是否显示工具栏上某项的工具提示
Stretch	默认情况下，工具栏比包含在其中的项略宽或略高。如果把 Stretch 属性设置为 true，工具栏就会占据其容器的总长

16.2.2　ToolStrip 的项

在 ToolStrip 中可以使用许多控件。前面提到，工具栏应能包含按钮、组合框和文本框。除了与这些对应的控件之外，工具栏还可以包含其他控件，如表 16-5 所示。

表　16-5

控　件	说　明
ToolStripButton	表示一个按钮。用于带文本和不带文本的按钮
ToolStripLabel	表示一个标签。这个控件还可以显示图像，也就是说，这个控件可以用于显示一个静态图像，放在不显示其本身信息的另一个控件上面，例如文本框或组合框
ToolStripSplitButton	显示一个右端带有下拉按钮的按钮，单击该下拉按钮，就会在它的下面显示一个菜单。如果单击控件的按钮部分，该菜单不会打开
ToolStripDropDownButton	类似于 ToolStripSplitButton，唯一的区别是去除了下拉按钮，代之以下拉数组图像。单击控件的任一部分，都会打开其菜单部分
ToolStripComboBox	显示一个组合框
ToolStripProgressBar	在工具栏上嵌入一个进度条
ToolStripTextBox	显示一个文本框
ToolStripSeparator	前面在菜单示例中见过这个控件，它为各个项创建水平或垂直分隔符

在下面的示例中，要扩展菜单示例，添加一个工具栏。该工具栏将包含工具栏的标准控件和另外 3 个按钮：Bold、Italic 和 Underline。还有一个组合框用于选择字体(注意，这里选择字体的按钮

使用下载代码中的图像)。

试一试：扩展工具栏

按照下面的步骤用工具栏扩展前面的示例：

(1) 继续使用上一个示例，删除 Format 菜单中使用的 ToolStripMenuItem。选择 Show Help Menu 选项，并按下 Delete 键。然后在它的位置上添加 3 个 ToolStripMenuItem，把它们的 CheckOnClick 属性都改为 true：

- Bold
- Italic
- Underline

(2) 给窗体添加一个 ToolStrip。在 Actions 窗口中，单击 Insert Standard Items，选择并删除 Cut、Copy、Paste 和 Separator 项。插入 ToolStrip 时，RichTextBox 可能不会正确停靠。此时应把 Dock 改为 none，手工重置控件的大小，以填充窗体。这会把 Anchor 属性改为 Top, Bottom, Left, Right。

(3) 在工具栏的最后，选择 Button 三次和 Separator 一次，从而创建三个新按钮和一个分隔符(单击 ToolStrip 的最后一项，就会打开这些选项)。

(4) 创建最后两项，先从下拉列表中选择 ComboBox，再添加一个分隔符，作为最后一个选项。

(5) 选择 Help 项，把它从当前位置拖动到工具栏的最后一个位置上。

(6) 前 3 个按钮分别是 Bold、Italic 和 Underline 按钮，按照表 16-6 所示给控件命名。

表 16-6

ToolStrip 按钮	名 称
Bold 按钮	boldToolStripButton
Italic 按钮	italicToolStripButton
Underline 按钮	underlineToolStripButton
ComboBox	fontsToolStripComboBox

(7) 选择 Bold 按钮，单击 Image 属性中的省略号(...)按钮，选中 Project resource file 单选按钮，单击 Import 按钮。如果已下载了本书的源代码，就可以使用 Chapter16\Toolbars\Images 文件夹下的图像 BLD.ico、ITL.ico 和 UNDRLN.ico。注意 VS 提供的默认扩展名不包含 ICO，所以在浏览图标时，必须从下拉列表中选择 Show All Files。

(8) 对于 Bold 按钮，选择 BLD.ico 图像。

(9) 选择 Italic 按钮，把它的图像改为 ITL.ico。

(10) 选择 Underline 按钮，把它的图像改为 UNDRLN.ico。

(11) 选择 ToolStripComboBox。在 Properties 面板中，按照表 16-7 所示修改属性。

表 16-7

属 性	值
Items	MS Sans Serif Times New Roman
DropDownStyle	DropDownList

(12) 把 Bold、Italic 和 Underline 按钮的 CheckOnClick 属性设置为 true。

(13) 要选择组合框中的初始项，应把下面的代码输入到类的构造函数中：

```
public Form1()
{
  InitializeComponent();

  fontsToolStripComboBox.SelectedIndex = 0;
}
```

(14) 按下 F5 键，运行示例，打开的对话框如图 16-2 所示。

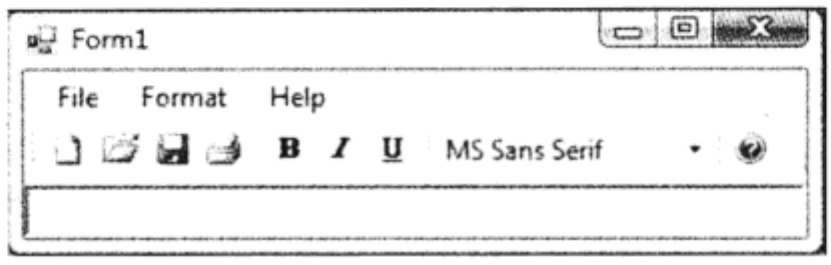


图　16-2

1. 添加事件处理程序

菜单上的 Save、New 和 Open 项已经有了处理程序，显然，工具栏上的按钮应与菜单的操作完全相同。这是很容易实现的，只需给工具栏上按钮的 Click 事件赋予菜单上按钮使用的相同处理程序即可。事件的设置如表 16-8 所示。

表　16-8

ToolStrip 按钮	事　件
New	newToolStripMenuItem_Click
Open	openToolStripMenuItem_Click
Save	saveToolStripMenuItem_Click

下面给 Bold、Italic 和 Underline 按钮添加处理程序。这些按钮都是复选框按钮，所以应使用 CheckedChanged 事件，而不是 Click 事件。给这 3 个按钮添加该事件。添加如下代码：


```
private void boldToolStripButton_CheckedChanged(object sender, EventArgs e)
    {
      Font oldFont, newFont;

      bool checkState = ((ToolStripButton)sender).Checked;
      oldFont = this.richTextBoxText.SelectionFont;

      if (!checkState)
        newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
      else
        newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

      richTextBoxText.SelectionFont = newFont;
      richTextBoxText.Focus();
```

```
        boldToolStripMenuItem.CheckedChanged -= new
EventHandler(boldToolStripMenuItem_CheckedChanged);
        boldToolStripMenuItem.Checked = checkState;
        boldToolStripMenuItem.CheckedChanged += new
EventHandler(boldToolStripMenuItem_CheckedChanged);
    }
    private void italicToolStripButton_CheckedChanged(object sender,
EventArgs e)
    {
        Font oldFont, newFont;

        bool checkState = ((ToolStripButton)sender).Checked;
        oldFont = this.richTextBoxText.SelectionFont;

        if (!checkState)
          newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Italic);
        else
          newFont = new Font(oldFont, oldFont.Style | FontStyle.Italic);

        richTextBoxText.SelectionFont = newFont;
        richTextBoxText.Focus();

        italicToolStripMenuItem.CheckedChanged -= new
EventHandler(italicToolStripMenuItem_CheckedChanged);
        italicToolStripMenuItem.Checked = checkState;
        italicToolStripMenuItem.CheckedChanged += new
EventHandler(italicToolStripMenuItem_CheckedChanged);
    }

    private void UnderlineToolStripButton_CheckedChanged(object sender,
EventArgs e)
    {
        Font oldFont, newFont;

        bool checkState = ((ToolStripButton)sender).Checked;
        oldFont = this.richTextBoxText.SelectionFont;

        if (!checkState)
          newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Underline);
        else
          newFont = new Font(oldFont, oldFont.Style | FontStyle.Underline);

        richTextBoxText.SelectionFont = newFont;
        richTextBoxText.Focus();

        underlineToolStripMenuItem.CheckedChanged -= new
EventHandler(underlineToolStripMenuItem_CheckedChanged);
        underlineToolStripMenuItem.Checked = checkState;
        underlineToolStripMenuItem.CheckedChanged += new
EventHandler(underlineToolStripMenuItem_CheckedChanged);
    }
```

代码段 Chapter16\Toolbars\Form1.cs

事件处理程序简单地把正确的样式设置为 RichTextBox 中使用的字体。在这 3 个方法中，最后 3 行代码分别处理菜单中的对应项。第一行从菜单项中删除事件处理程序，以确保下一行代码运行时不触发事件，第二行代码把 Checked 属性的状态设置为与工具栏按钮相同的值。最后，恢复事件处理程序。

菜单项的事件处理程序应只设置工具栏上按钮的 Checked 属性，让工具栏按钮的事件处理程序完成其他任务。为 CheckedChanged 事件添加处理程序，输入下面的代码：

```
private void boldToolStripMenuItem_CheckedChanged(object sender, EventArgs e)
    {
      boldToolStripButton.Checked = boldToolStripMenuItem.Checked;
    }

    private void italicToolStripMenuItem_CheckedChanged(object sender, EventArgs e)
    {
      italicToolStripButton.Checked = italicToolStripMenuItem.Checked;
    }

    private void underlineToolStripMenuItem_CheckedChanged(object sender,
EventArgs e)
    {
      underlineToolStripButton.Checked = underlineToolStripMenuItem.Checked;
    }
```

剩下的是让用户从 ComboBox 中选择一个字体系列。每当用户改变 ComboBox 中的选项，就会触发 SelectedIndexChanged 事件，所以为该事件添加处理程序，输入下面的代码：

```
private void fontsToolStripComboBox_SelectedIndexChanged(object sender,
EventArgs e)
    {
      string text = ((ToolStripComboBox)sender).SelectedItem.ToString();
      Font newFont = null;

      if (richTextBoxText.SelectionFont == null)
        newFont = new Font(text, richTextBoxText.Font.Size);
      else
        newFont = new Font(text, richTextBoxText.SelectionFont.Size,
                           richTextBoxText.SelectionFont.Style);
      richTextBoxText.SelectionFont = newFont;
    }
```

下面运行代码，就可以在工具栏上设置粗体、斜体和下划线文本了。注意选中或取消选中工具栏上的一个按钮，菜单上的对应选项也会选中或取消选中。

16.2.3　StatusStrip 控件

Strip 控件系列中的最后一个控件是 StatusStrip。这个控件在许多应用程序中表示对话框底部的一栏，它通常用于显示应用程序当前状态的简短信息，例如，在 Word 中键入文本时，Word 会在状态栏中显示当前的页面、列和行等。

StatusStrip 派生于 ToolStrip，在把这个控件拖放到窗体上时，读者应很熟悉其视图。在 StatusStrip 中可以使用前面介绍的 4 个控件中的 3 个：ToolStripDropDownButton、ToolStripProgressBar 和

ToolStripSplitButton。还有一个控件是 StatusStrip 专用的，即 StatusStripStatusLabel，它也是一个默认项。

16.2.4 StatusStripStatusLabel 的属性

StatusStripStatusLabel 使用文本和图像向用户显示应用程序当前状态的信息。标签是一个非常简单的控件，没有太多属性，虽然表 16-9 中介绍的两个属性不是专门用于标签的，但它们十分有用。

表 16-9

属 性	值
AutoSize	AutoSize 在默认状态下是打开的，这不是非常直观，因为在改变状态栏上标签的文本时，不希望该标签来回移动，除非标签上的信息是静态的，否则总是应把这个属性改为 false
DoubleClickEnable	在这个属性中，可以指定是否引发 DoubleClick 事件。也就是说，用户可以在应用程序的另一个地方修改信息。例如，让用户双击包含 Bold 的面板，在文本中启用或禁用粗体格式

在下面的示例中，要为前面的示例创建一个简单的状态栏。该状态栏包含 4 个面板，其中 3 个显示图像和文本，最后一个只显示文本。

试一试：StatusStrip 控件

按照下面的步骤扩展前面的小型文本编辑器：

(1) 在 ToolBox 中双击 StatusStrip，把它添加到对话框中。可能需要重置窗体上 RichTextBox 的大小。

(2) 在 Properties 面板中，单击 StatusStrip 的 Items 属性中的省略号(...)按钮，打开 Items Collection Editor。

(3) 单击 Add 按钮 4 次，给 StaturStrip 添加 4 个面板。面板的属性设置如表 16-10 所示。

表 16-10

面 板	属 性	值
1	Name	toolStripStatusLabelText
	Text	Clear this property
	AutoSize	False
	DisplayStyle	Text
	Font	Arial; 8.25pt; style=Bold
	Size	259,17
	TextAlign	Middle Left

(续表)

面　板	属　性	值
2	Name	toolStripStatusLabelBold:
	Text	Bold
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	47, 17
	Image	BLD
	ImageAlign	Middle-Center
3	Name	toolStripStatusLabelItalic
	Text	Italic
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	48, 17
	Image	ITL
	ImageAlign	Middle-Center
4	Name	toolStripStatusLabelUnderline
	Text	Underline
	DisplayStyle	ImageAndText
	Enabled	False
	Font	Arial; 8.25pt; style=Bold
	Size	76, 17
	Image	UNDRLN
	ImageAlign	Middle-Center

(4) 把下面这行代码添加到 boldToolStripButton_CheckedChanged 方法最后的事件处理程序中：

```
toolStripStatusLabelBold.Enabled = checkState;
```

(5) 把下面这行代码添加到 italicToolStripButton_CheckedChanged 方法最后的事件处理程序中：

```
toolStripStatusLabelItalic.Enabled = checkState;
```

(6) 把下面这行代码添加到 underlineToolStripButton_CheckedChanged 方法最后的事件处理程序中：

```
toolStripStatusLabelUnderline.Enabled = checkState;
```

(7) 选择 RichTextBox，把 TextChanged 事件添加到代码中，输入如下所示的代码：

```
    private void richTextBoxText_TextChanged(object sender, EventArgs e)
    {
      toolStripStatusLabelText.Text = "Number of characters: " +
richTextBoxText.Text.Length;
    }
```

运行应用程序，所创建的对话框如图 16-3 所示。

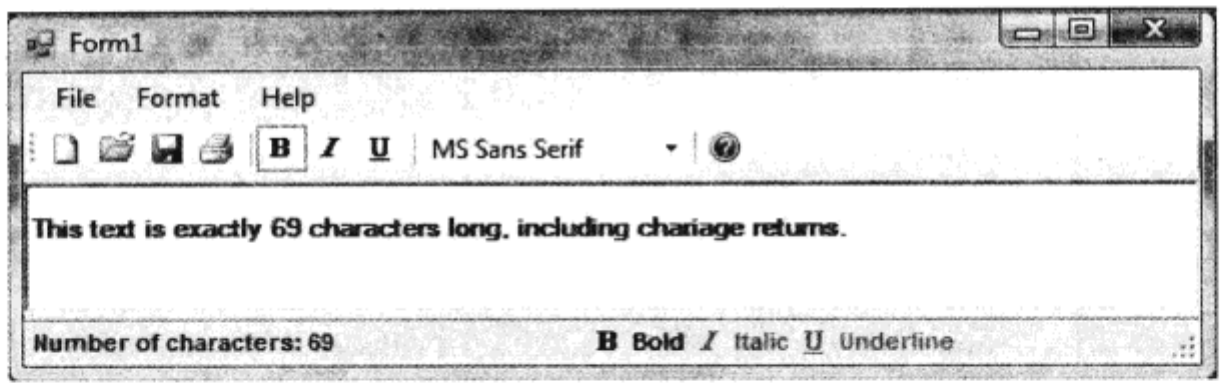


图 16-3

16.3 SDI 和 MDI 应用程序

传统上，可以为 Windows 编写 3 种应用程序，它们是：

- **基于对话框的应用程序**：它们向用户显示一个对话框，该对话框提供了所有的功能。
- **单一文档界面(SDI)**：这些应用程序向用户显示一个菜单、一个或多个工具栏和一个窗口，在该窗口中，用户可以执行任务。
- **多文档界面(MDI)**：这些应用程序的执行方式与 SDI 相同，但可以同时打开多个窗口。

基于对话框的应用程序通常用途比较单一，它们可以完成用户输入量非常少的特定任务，或者专门处理某一类型的数据。这种应用程序的一个示例是 Windows 中的计算器。

单一文档界面通常用于完成一个特定任务，因为它允许用户把要处理的单一文档加载到应用程序中。但这个任务通常涉及到许多用户交互操作，用户也常常希望能保存或加载工作的结果。SDI 应用程序的示例是写字板和画图，它们都是 Windows 附带的程序。本章前面建立的简单文本编辑器也是 SDI 应用程序。

但一次只能打开一个文档，所以如果用户要打开第二个文档，就必须打开一个新的 SDI 应用程序实例，它与第一个实例没有关系，对一个实例的任何配置都不会影响第二个实例。例如，在画图的一个实例中，可以把绘图颜色设置为红色，如果打开画图的第二个实例，绘图颜色仍是默认的黑色。

多文档界面与 SDI 应用程序极为相似，但它可以在任一时刻在不同的窗口中保存多个已打开的文档。MDI 的标识符包含在菜单栏右边的 Window 菜单中，该菜单在 Help 的前面。VS 就是一个 MDI 应用程序。VS 的每个设计器和编辑器都在同一个应用程序中打开，菜单和工具栏会自动调整，以匹配当前的选择。

16.4　生成 MDI 应用程序

创建 MDI 会涉及到什么问题？首先，希望用户能完成的任务应是需要一次打开多个文档的任务。例如，文本编辑器或文本查看器。第二，应在应用程序中提供工具栏来完成最常见的任务，例如，设置字体样式、加载和保存文档等。第三，应提供一个包含 Window 菜单项的菜单，让用户可以重新定位打开的窗口(平铺和层叠)，显示所有已打开窗口的列表。MDI 应用程序的另一个功能是如果打开了一个窗口，该窗口包含一个菜单，则该菜单就应集成到应用程序的主菜单上。

MDI 应用程序至少要由两个截然不同的窗口组成。第一个窗口叫作 MDI 容器(Container)，可以在容器中显示的窗口叫作 MDI 子窗口。MDI 容器既可以叫“MDI 容器”也可以叫“主窗口”，MDI 子容器既可以叫“MDI 子容器”又可以叫“子窗口”。

下面介绍一个小示例，来说明如何完成这些步骤，之后执行更复杂的任务。

试一试：创建一个 MDI 应用程序

创建 MDI 应用程序，首先要像创建其他应用程序那样，在 Visual Studio 中创建一个 Windows 窗体应用程序。

(1) 在 C:\BegVCSharp\Chapter16 目录中创建一个新的 Windows 应用程序，命名为 MDIBasic。

(2) 要把应用程序的主窗口从一个窗体改为 MDI 容器，只需把窗体的 IsMdiContainer 属性设置为 true 即可。改变窗体的背景色，使之表示该窗体现在只有一种背景色，不应放置任何可见的控件(也可以放置控件，在某些情况下这也是合理的，例如创建窗口的停靠区域)。

选择窗体，设置如表 16-11 所示的属性。

表 16-11

属　性	值
Name	frmContainer
IsMdiContainer	True
Text	MDI Basic
WindowState	Maximized

(3) 要创建子窗口，可以选择 Project | Add New Item，在打开的对话框中选择 Windows Form，给项目添加一个新窗体，命名为 frmChild。

(4) 把这个新窗体的 MdiParent 属性设置为主窗口的一个引用，该窗体就变成子窗口了。不能通过 Properties 面板设置这个属性，只能通过代码来设置。修改这个新窗体的构造函数：

```
public frmChild(MdiBasic.frmContainer parent)
{
   InitializeComponent();

   MdiParent = parent;
}
```

(5) 在 MDI 应用程序可以按最基本的模式显示之前，还有两件事要做：必须告诉 MDI 容器显示哪个窗口，再显示它们。为此，创建要显示窗体的一个新实例，再对它调用 Show()。要显示为子窗口的窗体的构造函数应与父容器相关联，方法是把它的 MdiParent 属性设置为 MDI 容器的实例。修改 MDI 父窗口的构造函数：

```
public frmContainer()
{
   InitializeComponent();

   frmChild child = new frmChild(this);

   child.Show();
}
```

示例的说明

显示子窗体需要的所有代码放在窗体的构造函数中。首先看看子窗口的构造函数：

```
public frmChild(MdiBasic.frmContainer parent)
{
   InitializeComponent();

   // Set the parent of the form to the container
   this.MdiParent = parent;
}
```

为了把子窗体绑定到 MDI 容器上，子窗体必须注册到容器中。为此，设置该窗体的 MdiParent 属性，如上面的代码所示。注意，使用的构造函数包括参数 parent。

因为 C#没有为定义了自己构造函数的类提供默认构造函数，所以上面的代码可以防止创建没有绑定到 MDI 容器上的窗体实例。

最后显示窗体，这是在 MDI 容器的构造函数中完成的：

```
public frmContainer()
{
   InitializeComponent();

   frmChild child = new frmChild(this);

   child.Show();
}
```

创建子类的一个新实例，把 this 传递给构造函数，其中 this 表示 MDI 容器类的当前实例。然后对子窗体的新实例调用 Show()就可以了。如果要显示多个子窗口，只需对每个窗口重复上面代码中突出显示的几行代码即可。

现在运行代码，得到如图 16-4 所示的结果(但 MDI Basic 窗体初始时为最大化窗口，这里重新设置了它的大小，以便放在书页中)。

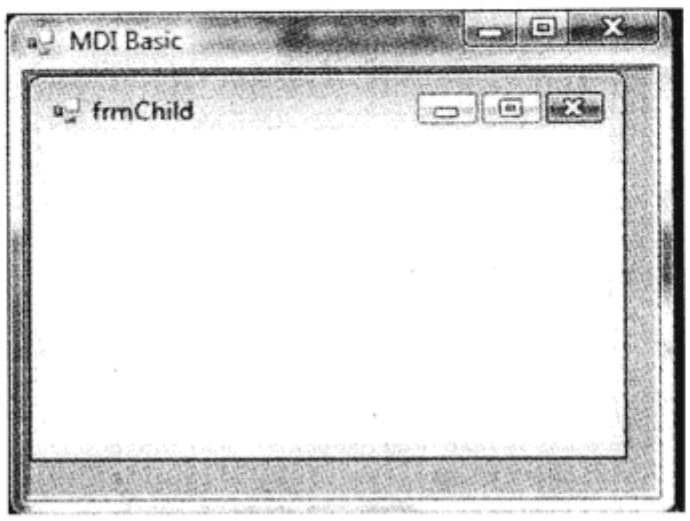


图　16-4

这并不是最吸引人的用户界面，但它是一个好的开端。下一个示例是一个简单的文本编辑器，根据本章前面介绍的菜单、工具栏和状态栏创建。

试一试：创建一个 MDI 文本编辑器

下面先创建一个基本项目，再讨论其中的内容：

(1) 返回前面的状态栏示例，把窗体重新命名为 frmEditor，把它的 Text 属性改为 Editor。

(2) 给项目添加一个新窗体 frmContainer.cs，在窗体上设置下述属性，如表 16-12 所示。

表　16-12

属　性	值
Name	frmContainer
IsMdiContainer	True
Text	Simple Text Editor
WindowState	Maximized

(3) 打开 Program.cs 文件，在 Main 方法中修改包含 Run 语句的代码行，如下所示：

```
Application.Run(new frmContainer());
```

(4) 修改 frmEditor 窗体的构造函数：

```
public frmEditor(frmContainer parent)
{
   InitializeComponent();

      this.toolStripComboBoxFonts.SelectedIndex = 0;
      MdiParent = parent;
}
```

(5) 把菜单项&File 的 MergeAction 属性改为 Replace，把&Format 菜单项的 MergeAction 属性改为 MatchOnly。

把工具栏的 AllowMerge 属性改为 False。

(6) 给 frmContainer 窗体添加一个 MenuStrip。再在 MenuStrip 中添加一个菜单项&File。

(7) 修改窗体 frmContainer 的构造函数：

```
public frmContainer()
{
   InitializeComponent();

   frmEditor newForm = new frmEditor(this);
   newForm.Show();
}
```

现在运行应用程序，得到如图 16-5 所示的结果。

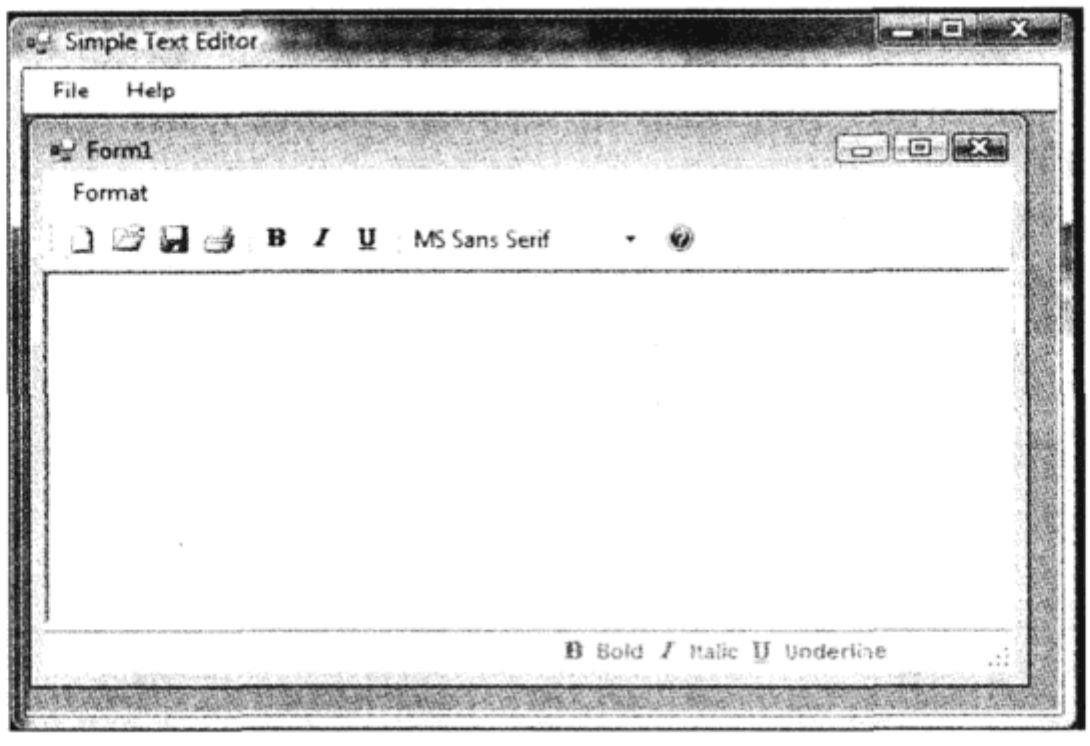


图 16-5

示例的说明

注意这里有一些技巧。File 和 Help 菜单似乎从 frmEditor 窗体中删除了。如果在容器窗口中选择 File 菜单，就会找到 frmEditor 对话框中的菜单项。

如果菜单应包含在子窗口中，则菜单就是这个窗口所特有的。File菜单在所有窗口中都应该有，不应只包含在子窗口中。原因很明显，如果关闭编辑器窗口，File 菜单就不包含任何菜单项。我们希望能在 File 菜单中插入一些菜单项，当子窗口获得焦点时，这些菜单项是专用于该子窗口的，而其他菜单项在主窗口中显示。

控制菜单项的操作的属性如表 16-13 所示。

表 16-13

属 性	说 明
MergeAction	这个属性指定一个菜单项与另一个菜单合并时该如何操作。可能的值有： Append: 该菜单项放在菜单的最后一个位置上 Insert: 插入到满足条件的菜单项的前一位置，该条件可以是菜单项上的文本或菜单项的索引 MatchOnly: 需要匹配，但不插入菜单项 Remove: 删除满足条件的菜单项，以插入新菜单项 Replace: 替换匹配的菜单项，把下拉菜单项添加到新加入的菜单项之后

(续表)

属　性	说　明
MergeIndex	MergeIndex 表示菜单项相对于要合并的其他菜单项的位置。如果要控制所合并菜单项的顺序，就把这个属性设置为大于或等于 0 的值，否则就把它设置为 - 1。在进行合并时，会检查这个值，如果它不是 - 1，该属性就用于匹配菜单项，而不是文本
AllowMerge	把 AllowMerge 设置为 false 表示不合并菜单

在下面的示例中，要继续完成文本编辑器，修改菜单的合并方式，以反映哪些菜单属于哪个窗口。

试一试：合并菜单

按照下面的步骤修改文本编辑器，以便在容器和子窗口中使用菜单。

(1) 在 frmContainer 窗体的 File 菜单中添加如下 4 个菜单项，如表 16-14 所示。注意 MergeIndex 值中的跳跃(jump)。

表　16-14

条　目	属　性	值
&New	MergeAction	MatchOnly
	MergeIndex	0
	ShortcutKeys	Ctrl + N
&Open	MergeAction	MatchOnly
	MergeIndex	1
	ShortcutKeys	Ctrl + O
-	MergeAction	MatchOnly
E&xit	MergeAction	MatchOnly
	MergeIndex	11

(2) 需要一种添加新窗口的方式，所以双击菜单项 New，并添加如下代码。这段代码与为显示第一个对话框而输入到构造函数中的代码相同：

```
private void ToolStripMenuItemNew_Click(object sender, EventArgs e)
{
  frmEditor newForm = new frmEditor(this);
  newForm.Show();
}
```

(3) 在 frmEditor 窗体中，从 File 菜单中删除 Open 菜单项，然后修改其他菜单项的属性，如表

16-15 所示。

表 16-15

条 目	属 性	值
&File	MergeAction	MatchOnly
	MergeIndex	-1
&New	MergeAction	MatchOnly
	MergeIndex	-1
-	MergeAction	Insert
	MergeIndex	2
&Save	MergeAction	Insert
	MergeIndex	3
Save &As	MergeAction	Insert
	MergeIndex	4
-	MergeAction	Insert
	MergeIndex	5
&Print	MergeAction	Insert
	MergeIndex	6
Print Preview	MergeAction	Insert
	MergeIndex	7
-	MergeAction	Insert
	MergeIndex	8
E&xit	Name	closeToolStripMenuItem
	Text	&Close
	MergeAction	Insert
	MergeIndex	9

(4) 运行应用程序。现在两个 File 菜单已合并了，但子对话框中仍有一个 File 菜单，其中只包含一个菜单项 New。

示例的说明

设置为 MatchOnly 的菜单项不能在菜单之间移动，但对于&File 菜单项，两个菜单项的文本匹配，意味着它们的菜单项合并在一起了。

File 菜单中的菜单项根据其 MergedIndex 属性来合并。其 MergedIndex 属性设置为 MatchOnly 的菜单项保持不变，其他菜单项的 MergedIndex 属性设置为 Insert。

在两个不同菜单上单击菜单项 New 和 Save 时比较有趣。子对话框上的 New 菜单项只是清除文本框的内容，而另一个对话框上的 New 菜单项会创建一个新对话框。这是因为这两个菜单项属于不同的窗口，它们都按照希望的那样工作，但 Save 菜单项如何？单击它，它会从当前的对话框移动到父对话框中。

打开几个对话框，在其中输入一些文本，然后单击 Save 菜单项。打开一个新的对话框，单击

Open 菜单项(Save 菜单项总是保存到同一文件中)。选择其他窗口中的一个，单击 Save，然后返回新对话框，再次单击 Open。结果，Save 菜单项总是跟随有焦点的对话框。每次选择一个对话框时，都会再次合并菜单。

刚才给 frmContainer 对话框的 File 菜单中的 New 菜单项添加了一些代码，创建了对话框。几乎所有 MDI 应用程序都包含 Window 菜单，它允许安排对话框的位置，按照某种方式将其列出。下面的示例就把这个菜单添加到文本编辑器中。

试一试：跟踪窗口

按照下面的步骤扩展应用程序，使之可以显示所有打开的对话框，并排列它们。

(1) 给 frmContainer 菜单添加一个顶级菜单项&Window。

(2) 给新菜单添加如表 16-16 中所示的 3 个菜单项。

表　16-16

名　称	文　本
tileToolStripMenuItem	&Tile
cascasdeToolStripMenuItem	&Cascade
WindowsSeperatorMenuItem	-

(3) 选择 MenuStrip 本身，不是选择其中显示的菜单项，将 MDIWindowListItem 属性改为 windowToolStripMenuItem。

(4) 先双击 Tile 菜单项，再双击 Cascade 菜单项，以添加事件处理程序，然后输入下面的代码：

```
private void ToolStripMenuItemTile_Click(object sender, EventArgs e)
{
  LayoutMdi(MdiLayout.TileHorizontal);
}

private void ToolStripMenuItemCascasde_Click(object sender, EventArgs e)
{
  LayoutMdi(MdiLayout.Cascade);
}
```

(5) 将 frmEditor 对话框的构造函数改为：

```
public frmEditor(frmContainer parent, int counter)
{
  InitializeComponent();

  this.ToolStripComboBoxFonts.SelectedIndex = 0;

  // Bind to the parent.
  this.MdiParent = parent;
  this.Text = "Editor " + counter.ToString();
}
```

(6) 在 frmContainer 代码的顶部添加一个私有成员变量，修改构造函数和菜单项 New 的事件处理程序，如下所示：

```
public partial class frmContainer : Form
{
  private int mCounter;

  public frmContainer()
  {
    InitializeComponent();

    mCounter = 1;
    frmEditor newForm = new frmEditor(this, mCounter);
    newForm.Show();
  }

  private void ToolStripMenuItemNew_Click(object sender, EventArgs e)
  {
    frmEditor newForm = new frmEditor(this, ++mCounter);
    newForm.Show();
  }
```

示例的说明

这个示例最有趣的部分是 Window 菜单。让一个菜单列出在 MDI 应用程序中打开的所有对话框，只需在顶级创建一个菜单，把 MdiWindowListItem 属性设置为指向该菜单即可。

然后，Framework 就会为当前打开的每个对话框在该菜单的最后追加一个菜单项，代表当前对话框的菜单项会在旁边显示一个复选标记，单击该列表中的项可以选择另一个对话框。

其他两个菜单项 Tile 和 Cascade 演示了窗体的 MdiLayout 方法，该方法允许采用标准方式排列对话框。

对构造函数和 New 菜单项的修改只是确保给对话框编号。现在运行应用程序，添加几个窗口，注意 Window 菜单总是反映当前选中的窗口。

16.5 创建控件

有时 Visual Studio 提供的控件不能满足用户的需要。原因是多方面的，控件不能以希望的方式绘制自己，或者控件在某个方面有限制，或者需要的控件不存在。为此，Microsoft 提供了创建满足需要的控件的方式。Visual Studio 提供了一个项目类型 Windows Control Library，使用它可以创建自己的控件。

可以开发两种不同类型的自定义控件：

- **用户或组合控件**：这种控件是根据现有控件的功能创建一个新控件。这类控件一般用于把控件的用户界面和功能封装在一起，或者把几个其他控件组合在一起，从而改善控件的界面。
- **定制控件**：当没有控件可以满足要求时，就创建这类控件，即从头创建控件。它要自己绘出整个用户界面，在创建控件的过程中没有现有的控件可以使用。当想要创建的控件的用户界面与其他可用的控件都不同时，一般需要创建这样的控件。

本章主要讨论用户控件，因为从头设计和绘制定制控件超出了本书的讨论范围。

在 Visual Studio 6 中使用的 ActiveX 控件放在扩展名为.ocx 的特殊文件中，这些文件实际上是 COM DLL。在.NET 中，控件存在的方式与其他程序集一样，所以没有.ocx 扩展名了，控件存在于 DLL 中。

用户控件继承于类 System.Windows.Forms.UserControl。这个基类提供的控件具有.NET 中控件应具有的所有基本功能——用户只需创建控件即可。实际上，任何对象都可以创建为一个控件，包括设计俏皮的标签乃至功能全面的网格控件，如图 16-6 所示，底部的框 UserControl1 代表一个新控件。

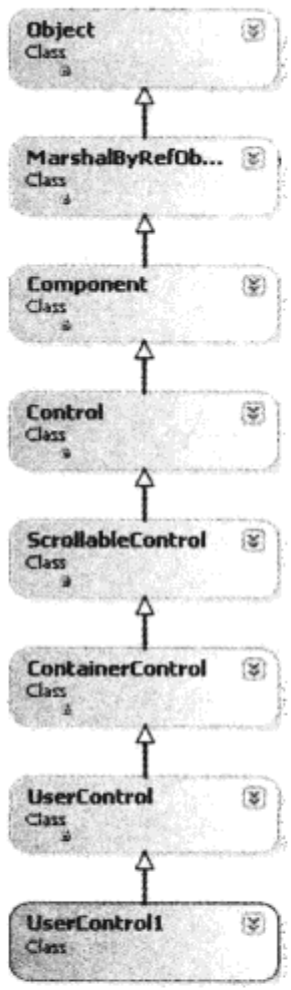


图　16-6

用户控件派生于 System.Windows.Forms.UserControl 类，而定制控件派生于 System.Windows.Forms. Control 类。

在处理控件时，要考虑几个问题。如果控件不满足这些条件，人们就不会使用它。这些条件是：

- 在设计期间，控件的操作方式应尽可能接近运行期间的操作方式。如果将一个标签和一个文本框合并，创建一个 LabelTextbox 控件，标签和文本框就都要在设计期间显示出来，为标签输入的文本也要在设计期间显示出来。在上例中，这是相当简单的，但在比较复杂的情况下，就会出问题，此时需要采取一种折中的方法。

- 应可以在窗体设计器中按合理方式访问控件的属性。例如，ImageList 控件显示了一个对话框，用户可以在该对话框中浏览要包含的图像，导入了图像后，它们就显示在对话框的一个列表中。

在下面的几页中，将通过一个示例说明如何创建控件。这个示例创建 LabelTextbox 控件，并讲述创建用户控件项目、创建属性和事件以及调试控件的基础知识。

从这个控件的名称可以看出，这个控件是使用两个现有控件来创建的。一步执行一个任务在 Windows 编程中非常常见：给窗体添加一个标签，再在该窗体中添加一个文本框，把文本框与标签的位置关联起来。下面看看这个控件的用户可以执行什么操作：

- 用户可以把文本框放在标签的右边或下边，如果文本框放在标签的右边，就可以指定该控件的左边界与文本框之间的固定距离，使文本框对齐。
- 用户应能使用文本框和标签的常用属性和事件。

试一试：LabelTextbox 示例

现在知道要做什么了，启动 Visual Studio，并创建一个新项目：

(1) 创建一个新的 Windows Forms Control Library 项目，命名为 LabelTextbox，将其保存在目录 C:\BegVCSharp\Chapter16 中。

如果使用的是 VS 的 Express 版本，就不能使用这个选项。此时应创建一个新的类库项目，在 Project 菜单中把一个用户控件手动添加到项目中。

窗体设计器显示了一个设计界面，它看起来与常用的界面有一些区别。首先，这个界面较小，其次，它看起来根本不像是对话框。读者不应因为这个新界面而泄气，其工作方式是一样的。主要的区别是前面都是把控件放在窗体上，现在则是创建一个要放在窗体上的控件。

(2) 单击设计界面，打开控件的属性。把控件的 Name 属性改为 ctlLabelTextbox。

(3) 双击工具箱中的标签，把它添加到用户控件中，放在设计界面的左上角。把它的 Name 属性改为 lblTextBox，把 Text 属性设置为 Label。

(4) 双击工具箱中的文本框，把它添加到用户控件中，把它的 Name 属性改为 txtLabelText。

在设计期间，不知道用户会如何放置这些控件，所以要编写代码，给标签和文本框定位。这些代码确定了在把 LabelTextbox 控件放在窗体上时控件的位置。

控件的设计是很鼓舞人心的，只是文本框遮挡了一部分标签，其界面也太大了。但这并没有负面作用，因为与习惯使用的其他控件不同，我们看到的并不是最后得到的结果。给该控件添加的代码会改变控件的外观，但只有在把控件添加到窗体时，才改变它的外观。

首先确定控件彼此的相对位置。用户应能决定控件如何定位，为此，要给控件添加两个属性，而不是一个属性。一个属性叫作 Position，允许用户选择两个选项之一：Right 和 Below。如果用户选择了 Right，就使用另一个属性，这个属性叫作 TextboxMargin，是一个 int，表示控件左边界到文本框的像素数。如果用户指定该属性为 0，文本框的右边界就与控件的右边界对齐。

1. 添加属性

为了让用户可以选择 Right 或 Below，先用这两个值定义一个枚举。返回控件项目，进入代码编辑器，添加如下代码：

```
public partial class ctlLabelTextbox : UserControl
{
   public enum PositionEnum
   {
      Right,
      Below
    }
```

这是一个第 5 章介绍的一般枚举。这里有一个技巧：要把这个位置设置为一个属性，用户可以通过代码和设计器实现这一设置。为此，给 ctlLabelTextbox 类添加一个属性。但首先创建两个成员字段，包含用户选择的值：

```
private PositionEnum mPosition = PositionEnum.Right;
private int mTextboxMargin = 0;
```

然后添加 Position 属性，如下所示：

```
public PositionEnum Position
{
   get { return position; }
   set
   {
      position = value;
      MoveControls();
   }
}
```

像添加其他属性那样将这个属性添加到类中。如果要返回该属性，就返回 position 成员字段，如果要修改 Position，就把值赋给 position，并调用 MoveControls()方法。稍后会介绍 MoveControls()方法，现在知道这个方法可以通过检查 position 和 textboxMargin 的值，来定位两个控件就可以了。

TextboxMargin 属性是一样的，但它处理的是一个整数：

```
public int TextboxMargin
{
   get { return textboxMargin; }
   set
   {
      textboxMargin = value;
      MoveControls();
   }
}
```

2. 添加事件处理程序

在测试两个属性前，还要添加两个事件处理程序。把该控件放在窗体上时，将调用 Load 事件。使用这个事件可以初始化控件和该控件使用的所有资源。处理这个事件是为了移动控件，设置它的大小，使之正好包容它包含的两个控件。

另一个要添加的事件是 SizeChanged。每当改变控件大小时，便会引发这个事件。处理这个事件是为了让控件正确地绘制它自己。选择控件，添加两个事件：SizeChanged 和 Load。

然后添加事件处理程序：

```
private void ctlLabelTextbox_Load(object sender, EventArgs e)
{
     labelCaption.Text = Name;
     Height = textBoxText.Height > labelCaption.Height ?
       textBoxText.Height : labelCaption.Height;
     MoveControls();
}

private void ctlLabelTextbox_SizeChanged(object sender, System.EventArgs e)
{
   MoveControls();
}
```

再调用 MoveControls()，定位控件。在再次测试控件之前，先介绍一下这个方法：

```
private void MoveControls()
{
  switch (position)
  {
    case PositionEnum.Below:
      textBoxText.Top = labelCaption.Bottom;
      textBoxText.Left = labelCaption.Left;
      textBoxText.Width = Width;
      Height = textBoxText.Height + labelCaption.Height;
      break;
    case PositionEnum.Right:
      textBoxText.Top = labelCaption.Top;
      if (textboxMargin == 0)
      {
        int width = Width - labelCaption.Width - 3;
        textBoxText.Left = labelCaption.Right + 3;
        textBoxText.Width = width;
      }
      else
      {
        textBoxText.Left = textboxMargin + labelCaption.Width;
        textBoxText.Width = Width - textBoxText.Left;
      }
      Height = textBoxText.Height > labelCaption.Height ?
                          textBoxText.Height : labelCaption.Height;
      break;
  }
}
```

在 switch 语句中测试 position 的值，确定应把文本框放在标签的下边还是右边。如果用户选择 Below，就把文本框的顶边移动到标签的底边上。然后把文本框的左边界移动到控件的左边界上，把它的宽度设置为控件的宽度。

如果用户选择了 Right，就有两种可能性。如果 TextboxMargin 是 0，就先确定控件中文本框的宽度，然后把文本框的左边界设置为靠近标签文本的右边界，把剩余的空间设置为文本框的宽度。

如果用户指定了边距，就把文本框的左边界放在该位置上，再次设置宽度。

下面准备测试这个控件，在开始测试前，先生成项目。

16.5.1　调试用户控件

调试用户控件与调试 Windows 应用程序大不相同。一般情况下，可以在某个位置添加断点，按下 F5，看看发生什么情况。如果读者仍对调试不熟悉，应参阅第 7 章中的详细论述。

控件需要一个容器来显示它本身，必须提供一个这样的容器，为此，下面的示例创建了一个 Windows Application 项目。

试一试：调试用户控件

(1) 从File菜单中选择Add | New Project，在 Add New Project 对话框中创建一个新的 Windows Application 应用程序，命名为 LabelTextboxTest。这个应用程序只用于测试用户控件，所以最好在 LabelTextbox 项目中创建该项目。

在 Solution Explorer 中，已打开了两个项目。第一个项目是前面创建的 LabelTextbox，以粗体字显示。如果要运行解决方案，调试程序就会把该控件项目用作启动项目。这将会失败，因为控件不是一种独立的项目类型。为了纠正这个问题，右击新项目名 LabelTextboxTest，选择 Set as Startup Project。如果现在运行解决方案，就会运行 Windows 应用程序项目，且不会产生错误。

(2) 现在，在工具箱的顶部应有一个选项卡 LabelTextBox Components。Visual Studio 知道在解决方案中有一个 Windows Control Library，在其他项目中也可能使用这个库提供的控件。所以双击控件 ctlLabelTextbox，把它添加到窗体上。注意，Solution Explorer 中的 References 节点被展开了，这是因为 Visual Studio 刚才添加了 LabelTextBox 项目的引用。

(3) 在代码中搜索新的 ctlLabel，应在整个项目中搜索，在后台文件 Form.Designer.cs 中找到它，Visual Studio 把它自动生成的大多数代码都放在这个后台文件中。注意，不能直接编辑这个文件。

(4) 在下面的代码行上放置一个断点。

```
this.MyControl = new LabelTextbox.ctlLabelTextbox();
```

(5) 运行代码，代码会停止在所设置断点的位置上。现在跟踪代码(如果使用默认的键盘映射，就可以按下 F11)。在跟踪代码时，将进入新控件的构造函数，这正是调试组件的地方，也可以设置断点。按下 F5 键，运行应用程序。

16.5.2　扩展 LabelTextbox 控件

最后，准备测试控件的属性。注意在把 LabelTextbox 控件添加到窗体上时，其中的控件会移动到正确的位置上。因为把 Position 属性的默认值设置为 Right，所以文本框位于标签的旁边，把 Position 属性改为 Below，文本框会移动到标签之下。

1. 添加更多属性

现在还不能对该控件进行什么操作，因为它还不能改变标签和文本框中的文本。下面添加两个属性：LabelText 和 TextboxText。添加这些属性的方式与添加前两个属性的方式相同，也是打开项目，添加如下代码：

```
public string LabelText
{
  get { return labelCaption.Text; }
  set
  {
    labelCaption.Text = labelText = value;
    MoveControls();
  }
}

public string TextboxText
{
  get { return textBoxText.Text; }
  set
  {
    textBoxText.Text = value;
  }
```

还需要声明成员变量 labelText 来保存文本：

```
private string mLabelText = "";

public ctlLabelTextbox()
{
```

如果要插入文本，就把文本赋给标签和文本框控件的 Text 属性，返回 Text 属性的值。如果改变了标签的文本，需要调用 MoveControls()，因为标签文本可能会影响文本框的位置。另一方面，插入到文本框中的文本不会使控件移动，如果文本比文本框长，超出文本框的部分就不会显示出来。

最后，必须修改 Load 事件：

```
private void ctlLabelTextbox_Load(object sender, EventArgs e)
{
  labelCaption.Text = labelText;
  Height = textBoxText.Height > labelCaption.Height ?
    textBoxText.Height : labelCaption.Height;
  MoveControls();
}
```

Load 事件把 labelCaption 控件的文本设置为属性的值。这样，设计期间和运行期间显示的文本就是相同的。

2. 添加更多事件处理程序

现在该考虑控件应提供的事件了。因为该控件派生于 UserControl 类，所以继承了许多无需加以处理的功能。但有许多事件我们不希望以标准的方式交给用户。例如 KeyDown、KeyPress 和 KeyUp 事件。需要修改这些事件的原因是，用户希望在文本框中按下一个键时，就引发这些事件。现在，只有在控件本身获得焦点，且用户按下一个键时，才会引发这些事件。

要改变其操作方式，必须处理文本框引发的事件，把它们发送给用户。给文本框添加 KeyDown、KeyUp 和 KeyPress 事件，并输入下面的代码：

```
private void textBoxText_KeyDown(object sender, KeyEventArgs e)
{
  OnKeyDown(e);
```

```
}

private void textBoxText_KeyPress(object sender, KeyPressEventArgs e)
{
  OnKeyPress(e);
}

private void textBoxText_KeyUp(object sender, KeyEventArgs e)
{
  OnKeyUp(e);
}
```

调用 OnKeyXXX 方法会执行订阅事件的对应方法。

3. 添加定制的事件处理程序

在创建一个基类中不存在的事件时，需要做更多的工作。下面创建一个事件 PositionChanged，当 Position 属性改变时，将引发该事件。为了创建这个事件，需要做 3 件事：

- 需要一个合适的委托，用于调用用户赋给事件的方法。
- 用户必须把一个方法赋给事件，以订阅该事件。
- 必须调用用户赋给事件的方法。

要使用的委托是由.NET Framework 提供的 EventHandler 委托。如第 13 章所述，这是一种特殊的委托，它由其关键字 event 声明。下面的代码声明了一个事件，允许用户订阅该事件：

```
public event System.EventHandler PositionChanged;

public ctlLabelTextbox()
{
```

现在只剩下引发该事件了。当改变 Position 属性时，将引发该事件。所以在 Position 属性的 set 存取器中引发该事件：

```
public PositionEnum Position
{
   get { return position; }
   set
   {
      position = value;
      MoveControls();
      if (PositionChanged != null)
         PositionChanged(this, new EventArgs());
    }
}
```

首先，确保检查 PositionChanged 是否为 null，看看有没有订阅者。如果没有，就调用方法。

可以像订阅其他事件那样订阅新的定制事件，但这里有一个小问题：该事件在事件窗口中显示之前，必须先生成控件。只有生成了控件，才能在 LabelTextboxTest 项目的窗体中选择控件，在属性面板的 Events 部分双击 PositionChanged 事件。接着给事件处理程序添加如下代码：

```
private void ctlLabelTextbox1_PositionChanged(object sender, EventArgs e)
{
   MessageBox.Show("Changed");
}
```

该定制事件处理程序什么都不会做，它只是说明位置改变了。

最后，在窗体上添加一个按钮，双击它，给项目添加该按钮的 Click 事件处理程序，添加如下代码：

```
    private void buttonToggle_Click(object sender, EventArgs e)
    {
      ctlLabelTextbox1.Position = ctlLabelTextbox1.Position ==
LabelTextbox.ctlLabelTextbox.PositionEnum.Right ?
LabelTextbox.ctlLabelTextbox.PositionEnum.Below :
LabelTextbox.ctlLabelTextbox.PositionEnum.Right;
    }
```

当运行应用程序时，就可以在运行期间改变文本框的位置。每次移动文本框，都会触发事件 PositionChanged，显示一个信息框。

这个示例到此就完成了。还可以细化它，这留给读者作为练习。

16.6 小结

本章继续第 15 章的内容，介绍了 MainMenu 和 Toolbar 控件。本章讨论了如何创建 MDI 和 SDI 应用程序，如何在这些应用程序中使用菜单和工具栏。接着论述如何创建自己的控件，设计该控件的属性、用户界面和事件。第 17 章是讨论 Windows 窗体的收官章节，介绍一种特殊类型的窗体：Windows 常用对话框。

16.7 练习

(1) 以 LabelTextbox 示例为基础，创建一个新属性 MaxLength，以存储文本框中可以输入的最大字符数，然后创建两个新事件 MaxLengthChanged 和 MaxLengthReached。MaxLengthChanged 事件应在修改 MaxLength 属性时引发，MaxLengthReached 事件应在用户输入一个字符后，使文本框中的文本长度等于 MaxLength 属性值时引发。

(2) StatusBar 包含一个属性，允许用户双击状态栏上的一个字段，引发一个事件。修改 StatusBar 示例，允许用户双击状态栏，给文本设置粗体、斜体和下划线样式。确保工具栏、菜单和状态栏上的显示总是同步的：激活粗体样式时，把文本 Bold 改为粗体，斜体和下划线样式亦是如此。

附录 A 给出了练习答案。

16.8 本章要点

主 题	重 要 概 念
菜单	使用 MenuStrip 在窗体上显示专业外观的菜单
工具栏	使用 ToolStrip 控件在窗体上显示工具栏
状态栏	StatusStrip 提供了一种显示应用程序当前状态的信息的方式
MDI 应用程序	创建 MDI 应用程序，用于进一步扩展文本编辑器
定制控件	以已有的控件为基础创建自己的控件

第17章

部署 Windows 应用程序

本章内容：

- 部署选项概述
- 用 ClickOnce 部署 Windows 应用程序
- 创建 Windows 安装程序的部署软件包
- 用 Windows 安装程序安装应用程序

可以采用多种方式安装 Windows 应用程序，简单的应用程序可以使用简单的 xcopy 部署来安装，但对于上百个客户的安装，xcopy 部署就没那么有用了。在这种情况下，可以使用 ClickOnce 部署，也可以使用 Microsoft Windows 安装程序来安装应用程序。

在 ClickOnce 部署中，可以通过单击某个网站的链接来安装应用程序。如果用户选择在其中安装应用程序的目录，或者如果需要一些注册表项，就应使用 Windows 安装程序部署选项。

本章介绍安装 Windows 应用程序的两个选项。

17.1 部署概述

部署就是把应用程序安装到目标系统上的进程。传统上，这种安装是通过调用安装程序来完成的。如果需要在成百上千个客户机上安装，安装过程就非常耗时。为了缓解这个问题，系统管理员可以创建批处理脚本，自动完成安装过程。但是，这仍需要做大量工作来安装和支持不同客户的 PC 和不同版本的操作系统。

由于存在这些问题，虽然 Windows 应用程序可以有更丰富的用户界面，许多公司也把它们的内联网应用程序转换为 Web 应用程序。Web 应用程序只需部署到服务器上，客户机就可以自动获取更新后的用户界面。

编写 Silverlight 应用程序是为多客户应用程序提供基于 Web 的部署的一个选项。

使用 ClickOnce 部署，可以解决在部署 Windows 应用程序时遇到的很多问题。通过单击 Web 页面中的一个链接即可安装应用程序。客户系统上的用户不需要有管理权限，因为应用程序安装在用户特定的目录下。使用 ClickOnce，可以安装带有丰富用户界面的应用程序。应用程序安装在客户机上，所以在安装完成后，不需要保留与客户系统的连接。换言之，应用程序可以脱机使用。这样，应用程序图标位于 Start 菜单上，安全问题更容易解决，应用程序也很容易卸载。

ClickOnce 的一个出色功能是，当客户应用程序启动时，更新过程会自动进行，或者在客户应用程序的运行过程中，更新过程会作为一个后台任务来执行。

但是，ClickOnce 部署也有一些限制：如果需要在全局程序集缓存上安装共享组件，或者应用程序所需的 COM 组件需要注册表设置，或者希望用户来确定安装应用程序的目录，就不能使用 ClickOnce。在这些情况下，必须使用 Windows 安装程序。Windows 安装程序是安装 Windows 应用程序的传统方式。下面先介绍 ClickOnce 部署，再介绍 Windows 安装程序包。

17.2 ClickOnce 部署

在使用 ClickOnce 部署时，不需要在客户系统上启动安装程序。客户系统的用户只需单击 Web 页面上的一个链接，应用程序就会自动安装。安装完成后，客户机就可以脱机——客户机不再需要访问从中安装应用程序的服务器。

ClickOnce 安装可以在网站、UNC 共享或文件位置(例如 CD)上进行。利用 ClickOnce，应用程序会安装在客户系统上，可以从 Start 菜单上启动，使用 Add/Remove Programs 对话框卸载。

ClickOnce 部署由清单文件(manifest files)描述。应用程序清单描述了应用程序及其需要的权限。部署清单描述了部署配置信息，如更新策略。在本节的示例中，将为第 16 章创建的 MDI Editor 配置 ClickOnce 部署，而且将再次需要这些代码文件。

17.2.1 创建 ClickOnce 部署

在下面的示例中，要修改应用程序的名称，定义有用的程序集设置。

试一试：准备应用程序

(1) 在 Visual Studio 中打开第 16 章中的 MDI Editor 示例。如果读者没有创建这个示例，可以从 Chapter16Code.zip 中复制完整的文件夹 MDI Editor。使用 Visual Studio 菜单 File | Open | Project/Solution . . .打开 MDI Editor 文件夹中的解决方案文件 Manual Menus.sln。

(2) 在 Solution Explorer 中选择项目的 Properties，再选择 Application 选项卡，如图 17-1 所示。

(3) 把 Assembly name 改为 MDIEditor。

(4) 单击 Assembly Information . . .按钮。

(5) 修改 Title、Description、Company、Product 和 Copyright 信息，如图 17-2 所示。

(6) 选择 Build | Build Solution，生成项目。

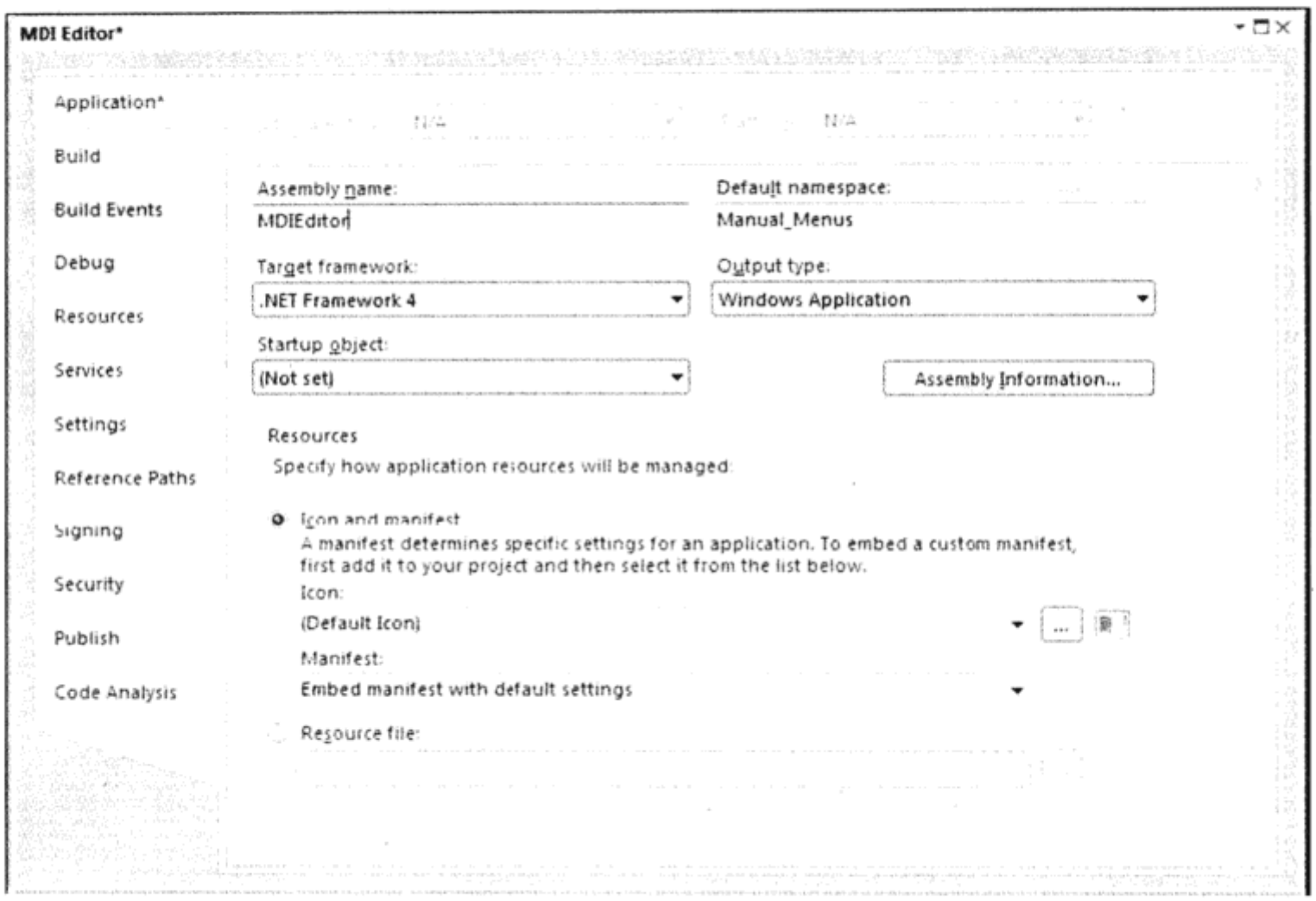


图　17-1

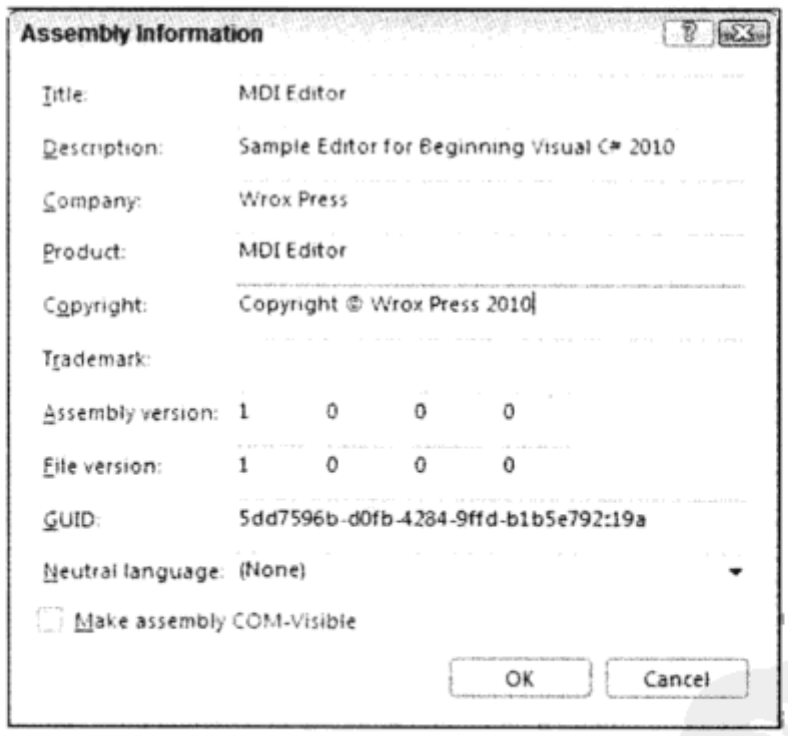


图　17-2

示例的说明

程序集名定义了在生成过程中创建的程序集的名称。在安装应用程序时，需要部署这个程序集。通过 Assembly Information 对话框修改的属性改变了 AssemblyInfo.cs 文件中的程序集特性。这些元数据信息由部署工具使用。还可以选择可执行文件，单击菜单中的 Properties，以便从 Windows 资源管理器中读取元数据信息。在 Details 选项卡中可以看到前面添加的信息。

在网络上成功部署程序集，需要使用清单，清单必须有证书签名。该证书会向安装应用程序的用户显示创建安装程序的机构。这样用户就可以确定是否相信该部署。在下面的示例中，要创建一个与 ClickOnce 清单相关的证书。

试一试：签署 ClickOnce 清单

(1) 在 Solution Explorer 中为项目选择 Properties，再选择 Signing 选项卡，如图 17-3 所示。

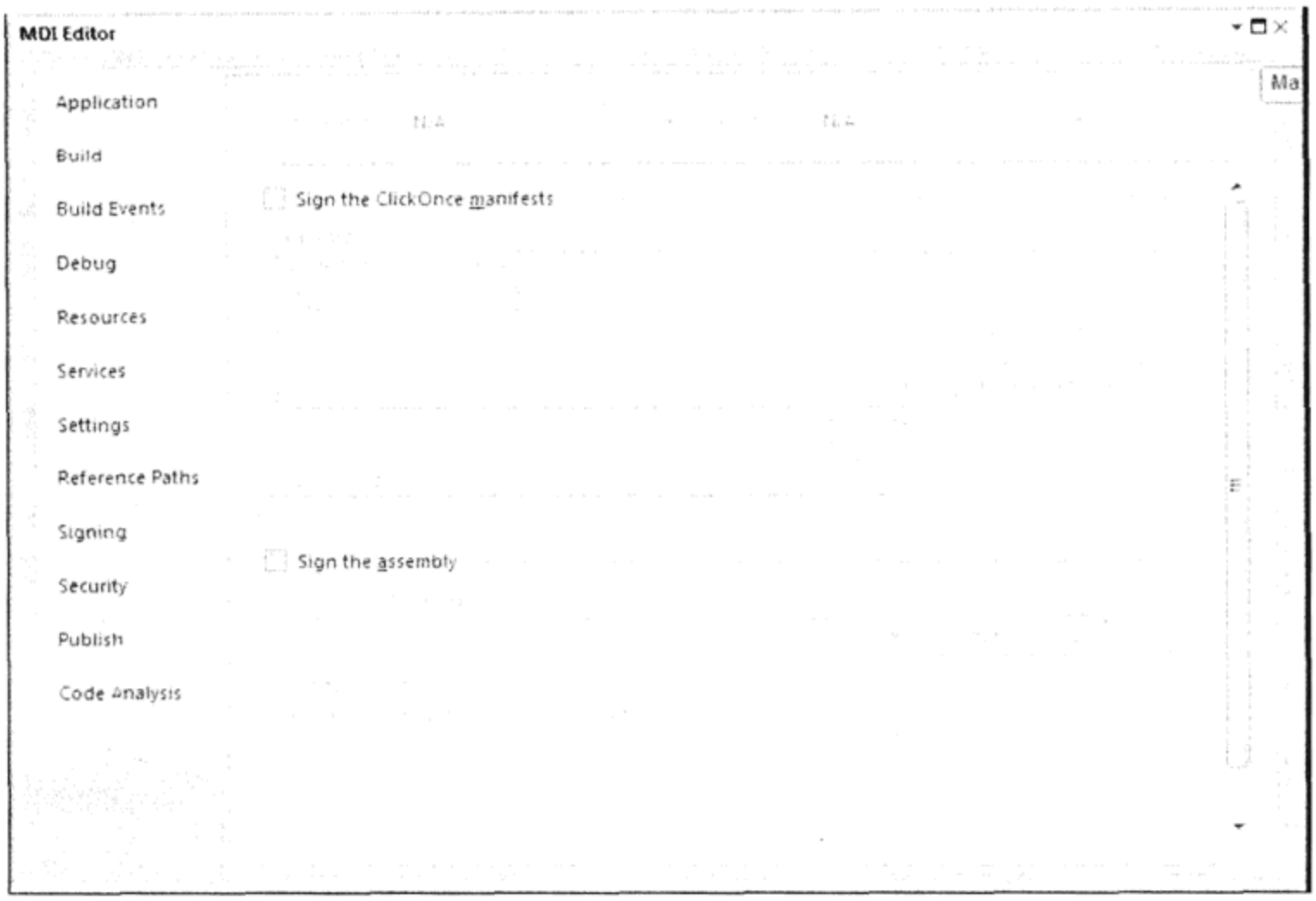


图 17-3

(2) 选中 Sign the ClickOnce manifests 复选框。

(3) 单击 Create Test Certificate...按钮，创建一个与 ClickOnce 清单相关的测试证书。根据要求给证书输入一个密码。必须记住密码，才能在以后进行设置。然后单击 OK 按钮。

(4) 单击 More Details 按钮，可以查看证书信息，如图 17-4 所示。

示例的说明

安装应用程序的用户可以使用证书来辨识安装软件包的创建者。阅读证书的内容，可以确定是否能信任此安装软件包，从而满足安全要求。

创建好测试证书后，用户并没有获得真正值得信任的信息，而是会接收到一个警告，说明这个证书不能信任，如后面所述。这个证书只供测试之用。在应用程序准备好部署之前，必须从证书颁发机构(如 VeriSign)处获得一个真正的证书。如果应用程序只在内联网中部署，还可以从安装在本地网络的本地证书服务器处获得一个证书。Microsoft 证书服务器可以与 Windows Server 2003 或 2008 一起安装。如果有了证书，就可以通过从 Signing 选项卡中单击 Select from File，来配置它。

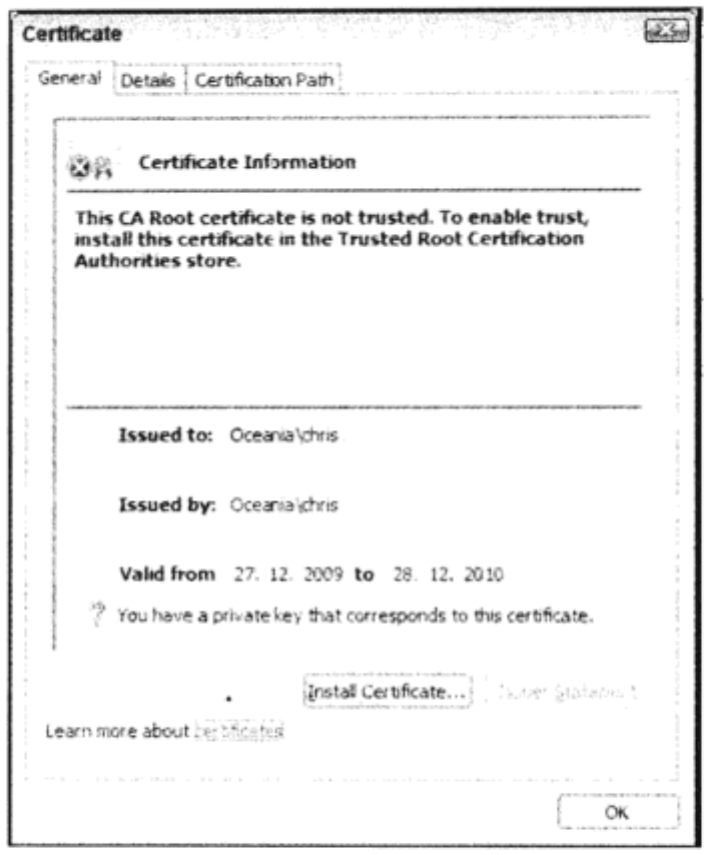


图　17-4

下面的示例将配置程序集的安全要求。在把程序集安装到客户机上时，必须定义所要求的信任。

试一试：定义安全要求

(1) 在 Solution Explorer 中为项目选择 Properties，再选择 Security，如图 17-5 所示。选中 Enable ClickOnce security settings 复选框。使用默认配置，以完全信任应用程序。

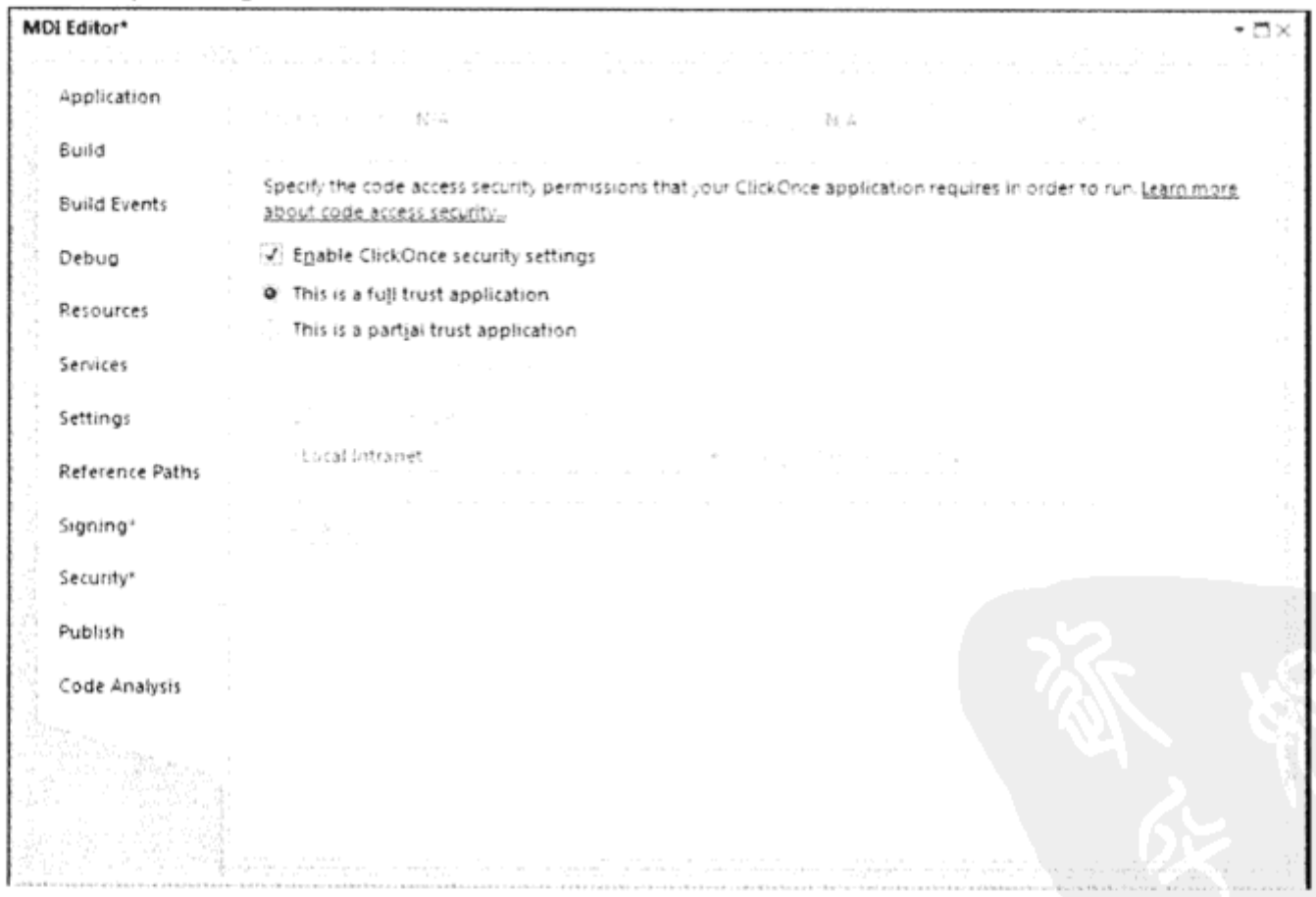


图　17-5

示例的说明

通过 ClickOnce 设置可以把应用程序配置为需要完全信任，或者运行在沙箱中，只能部分信任。

有了完全信任权限，应用程序就可以对系统进行全面的访问，可以执行运行应用程序的用户所允许执行的任何操作。在安装应用程序时，将警告用户：应用程序需要完全信任权限。部分信任的应用程序不能访问文件系统，只能访问独立的存储器或注册表，这种应用程序在沙箱模式下运行。由于 MDI Editor 应用程序需要访问文件系统，所以需要完全信任权限。

定义了安全需求后，就可以开始创建部署清单，以便发布应用程序。使用 Publish 向导很轻松地完成这项任务，如以下示例所示。

试一试：更多的发布配置选项

(1) 选择项目属性中的 Publish 选项卡。单击 Options 按钮，打开 Publish Options 对话框，如图 17-6 所示。选择左边列表中的 Description，输入发布者名称、系列名称、产品名称和支持的 URL。

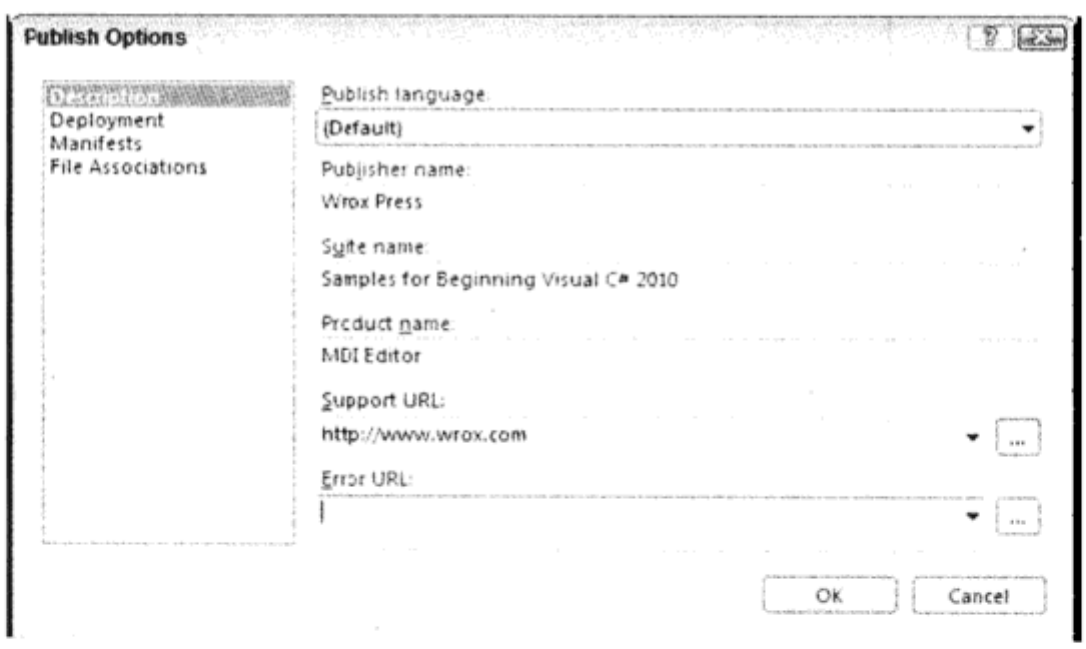


图 17-6

(2) 选择 Updates 按钮来配置 Update 选项，然后选中 The application should check for updates 复选框，如图 17-7 所示。

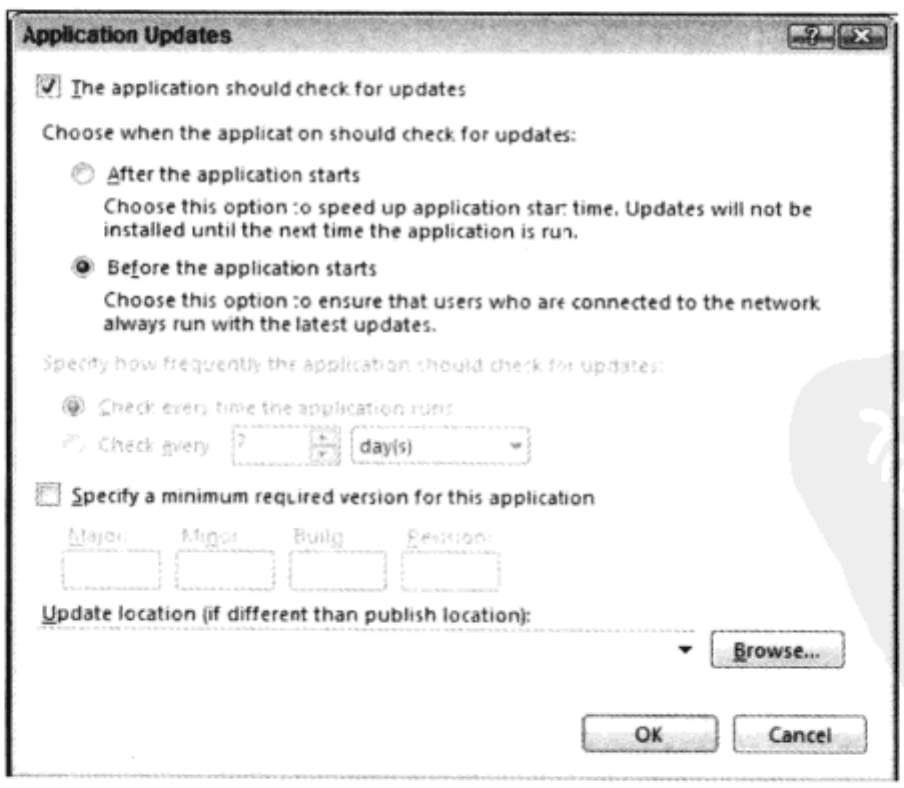


图 17-7

试一试：使用 Publish 向导

(1) 选择 Build | Publish SimpleEditor 菜单，启动 Publish 向导。输入网站的路径 http://localhost/MDIEditor，如图 17-8 所示。单击 Next 按钮。

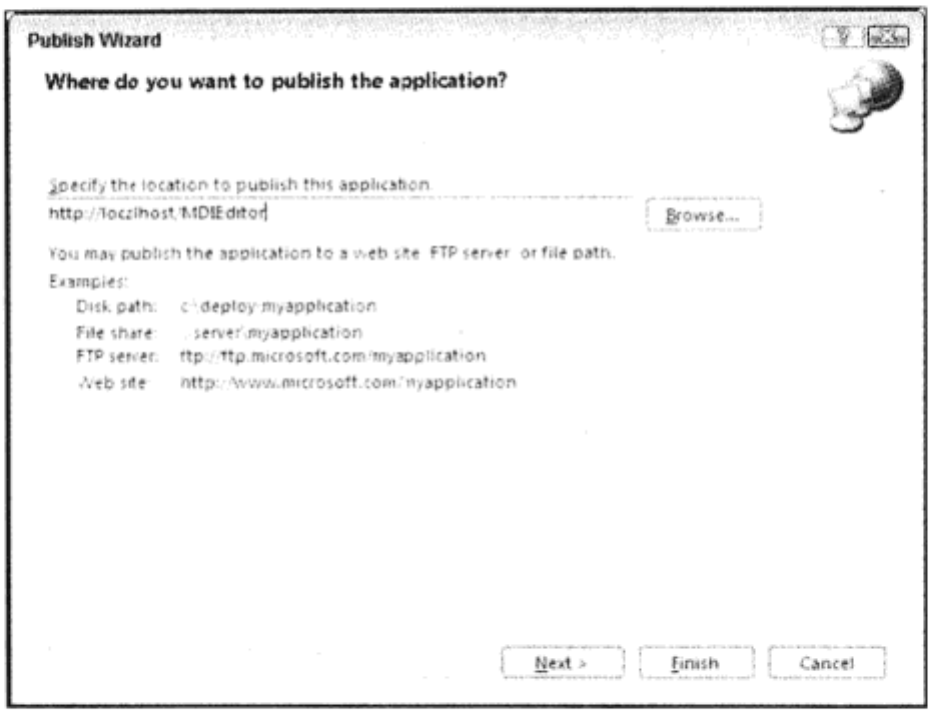


图　17-8

要在 Windows 7 或 Windows Vista 上把应用程序发布到 Web 服务器上，必须以 elevated 模式启动 VS2010，且必须拥有管理权限，还需要安装 IIS。如果没有安装 IIS，应选择发布到本地文件系统上。

(2) 在 Publish 向导的第(2)步，选中 Yes, this application is available online or offline 单选按钮，如图 17-9 所示。单击 Next 按钮。

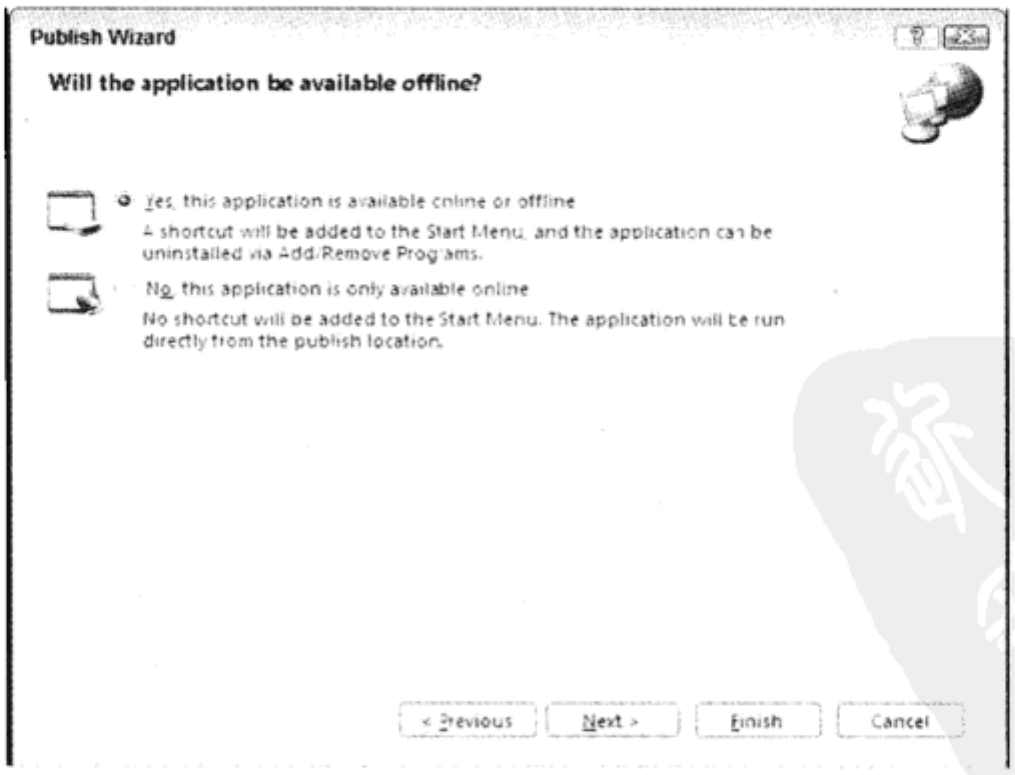


图　17-9

(3) 最后一个对话框给出了准备发布的汇总信息，如图 17-10 所示。单击 Finish 按钮。

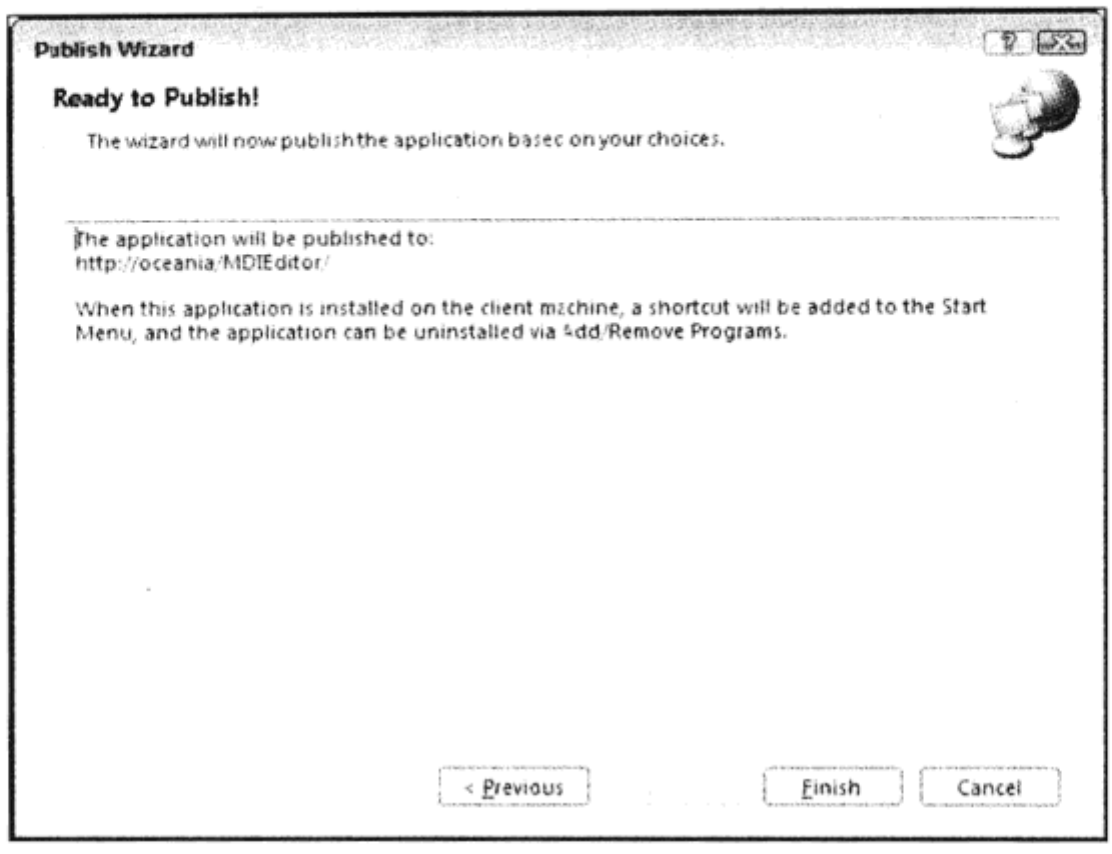


图 17-10

示例的说明

Publish 向导在本地的 Internet Information Services Web 服务器上创建了一个网站。把应用程序的程序集(可执行文件和库)、应用程序和部署清单、setup.exe 和一个示例 Web 页面 publish.htm 复制到 Web 服务器上。部署清单描述了安装信息，如下所示。使用 Visual Studio，可以在 Solution Explorer 中打开 MDIEditor.application 文件，从而打开部署清单。在这个清单中，通过 XML 元素<dependentAssembly>与应用程序清单建立依赖关系。

```
 <deployment install="true"mapFileExtensions="true">
   <subscription>
     <update>
       <beforeApplicationStartup />
     </update>
   </subscription>
   <deploymentProvider
      codebase="http://oceania/MDIEditor/MDIEditor.application" />
</deployment>
<compatibleFrameworks xmlns="urn:schemas-microsoft-com:clickonce.v2">
   <framework targetVersion="4.0" profile="Full" supportedRuntime="4.0.21205" />
</compatibleFrameworks>
<dependency>
 <dependentAssembly dependencyType="install"
        codebase="ApplicationFiles\MDIEditor_1_0_0_0\MDIEditor.exe.manifest"
      size="7416">
    <assemblyIdentity name="MDIEditor.exe" version="1.0.0.0"
       publicKeyToken=" 4e48aff44fcfc18a" language="neutral"
       processorArchitecture="x86" type="win32" />
    <hash>
```

```
      <dsig:Transforms>
        <dsig:Transform
            Algorithm="urn:schemas-microsoft-com:HashTransforms.Identity" />
      </dsig:Transforms>
        <dsig:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <dsig:DigestValue>+fiBvjYoMSuDkHZ680iLW2P4y+g=</dsig:DigestValue>
    </hash>
  </dependentAssembly>
</dependency>
```

选择如图 17-9 所示的选项，以便指定应用程序可以在线和离线使用。这样应用程序会安装在客户系统上，并可以从 Start 菜单中访问。还可以使用 Add/Remove Programs 来卸载应用程序。如果选择应用程序只能在线获得，就必须总是单击网站链接，从服务器上加载应用程序，在本地启动它。

属于应用程序的文件由项目输出定义。单击 Application Files 按钮，就可以在 Publish 设置下看到应用程序文件和应用程序的属性。Application Files 对话框如图 17-11 所示。默认情况下会部署程序集和应用程序清单文件。

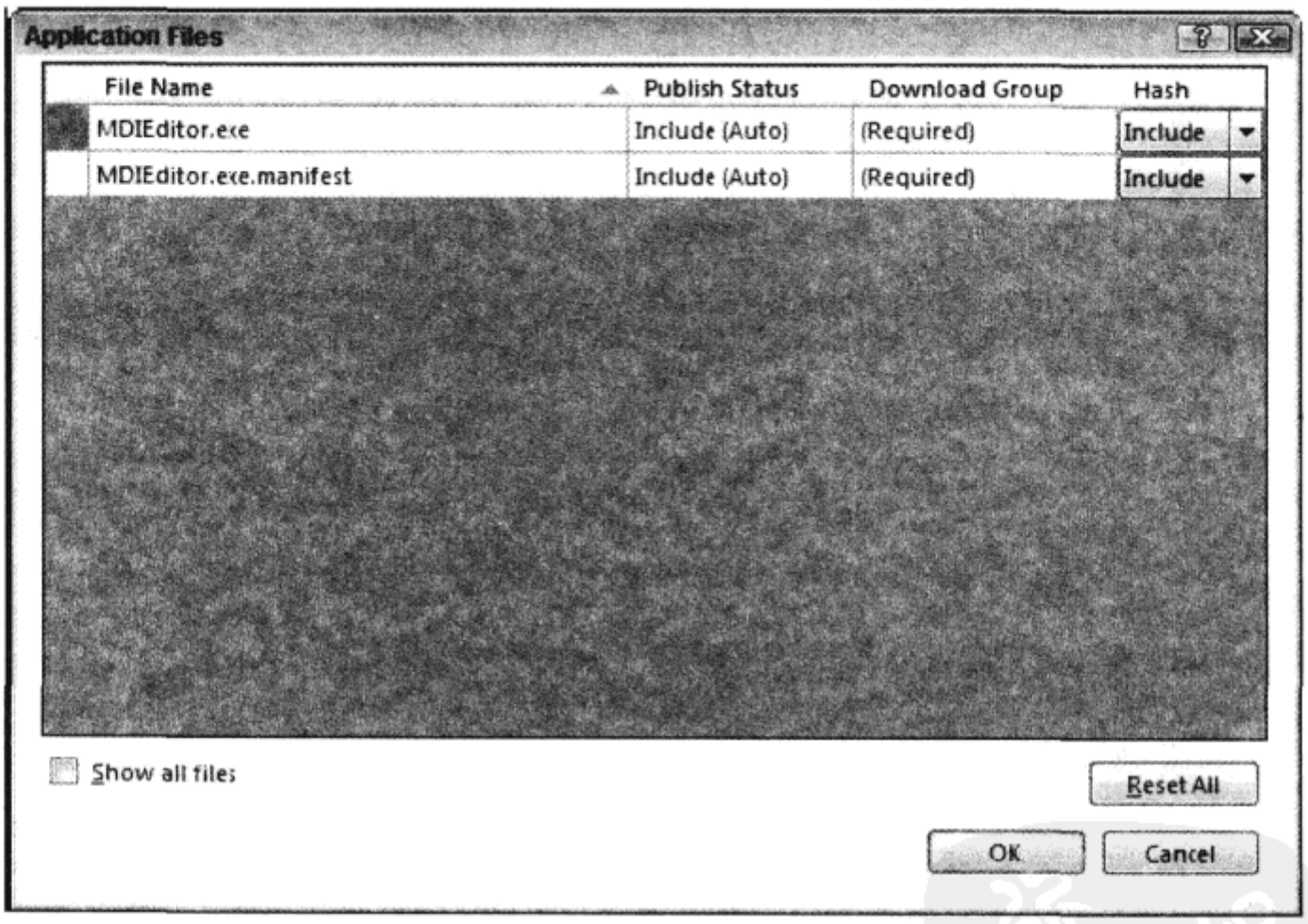


图　17-11

通过 Prerequisites 对话框来定义预先安装的软件包，如图 17-12 所示，单击 Prerequisites 按钮，就可以访问该对话框。对于.NET 4 应用程序，.NET Framework 4 会自动检测预先安装的软件包，如图 17-12 所示。利用这个对话框还可以选择其他预先安装的软件包。

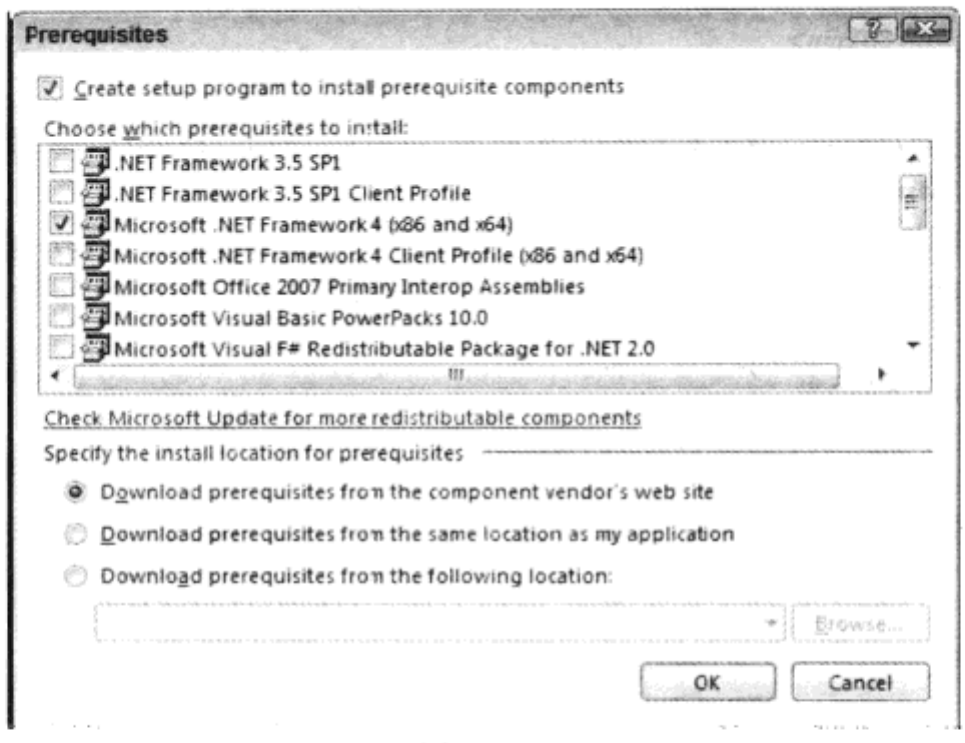


图 17-12

要安装 ClickOnce 应用程序，不需要管理权限。但如果没有在客户系统上安装需预先安装的软件包，就需要管理权限完成安装。

17.2.2 用 ClickOnce 安装应用程序

现在可以按照下面示例中的步骤安装应用程序了。

试一试：安装 MDI Editor 应用程序

(1) 打开 Web 页面 publish.htm，如图 17-13 所示。

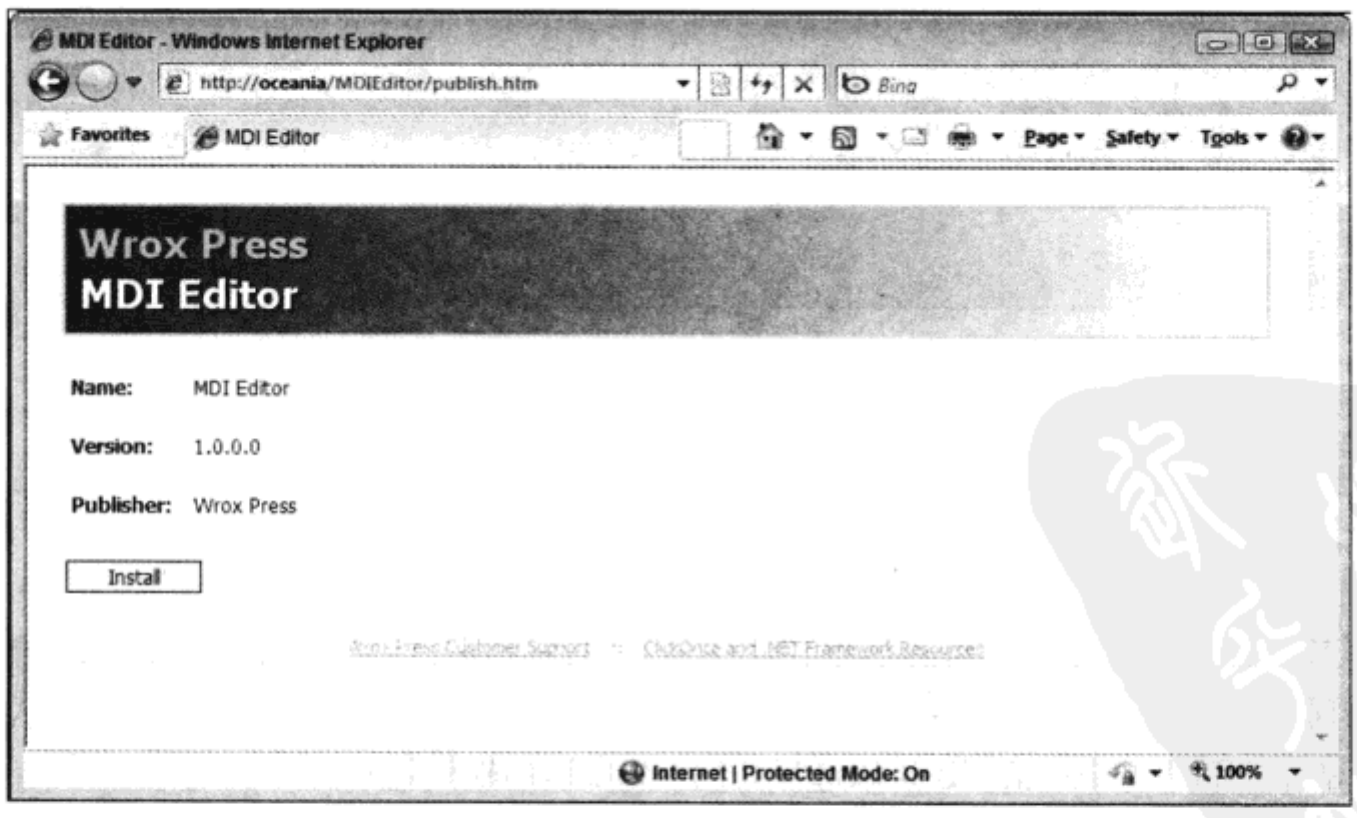


图 17-13

(2) 单击 Install 按钮，安装应用程序。此时会弹出一个安全警告，如图 17-14 所示。

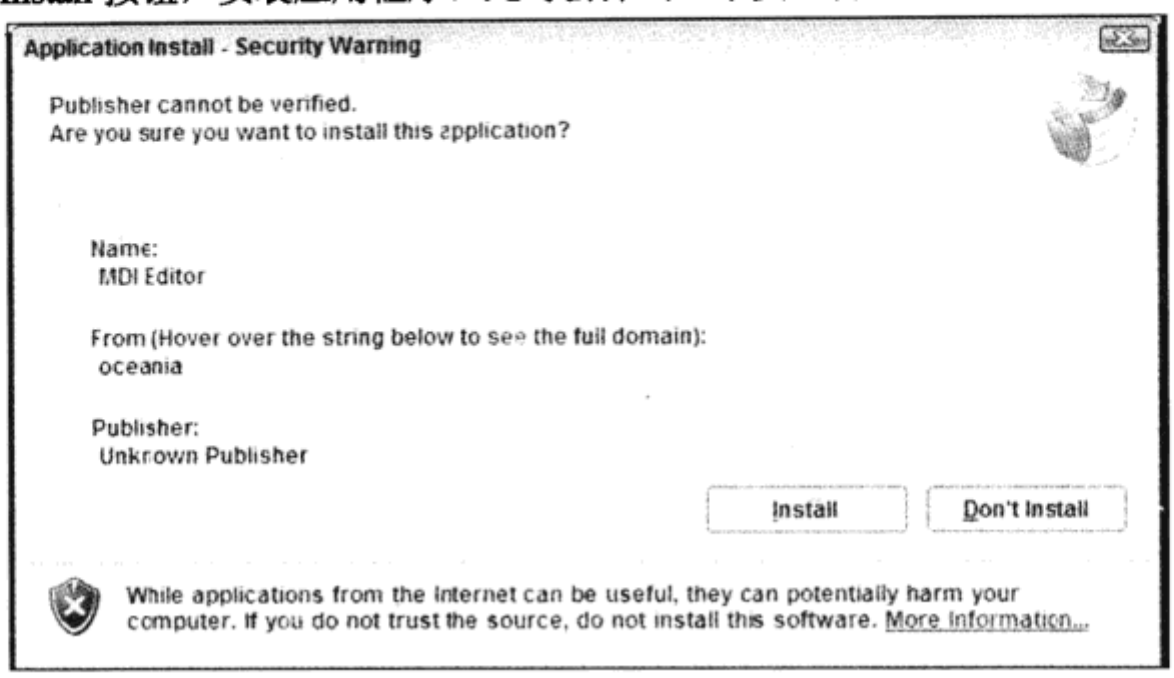


图　17-14

(3) 单击 More Information...链接，查看与应用程序相关的潜在安全问题。阅读此对话框的类别信息，如图 17-15 所示。

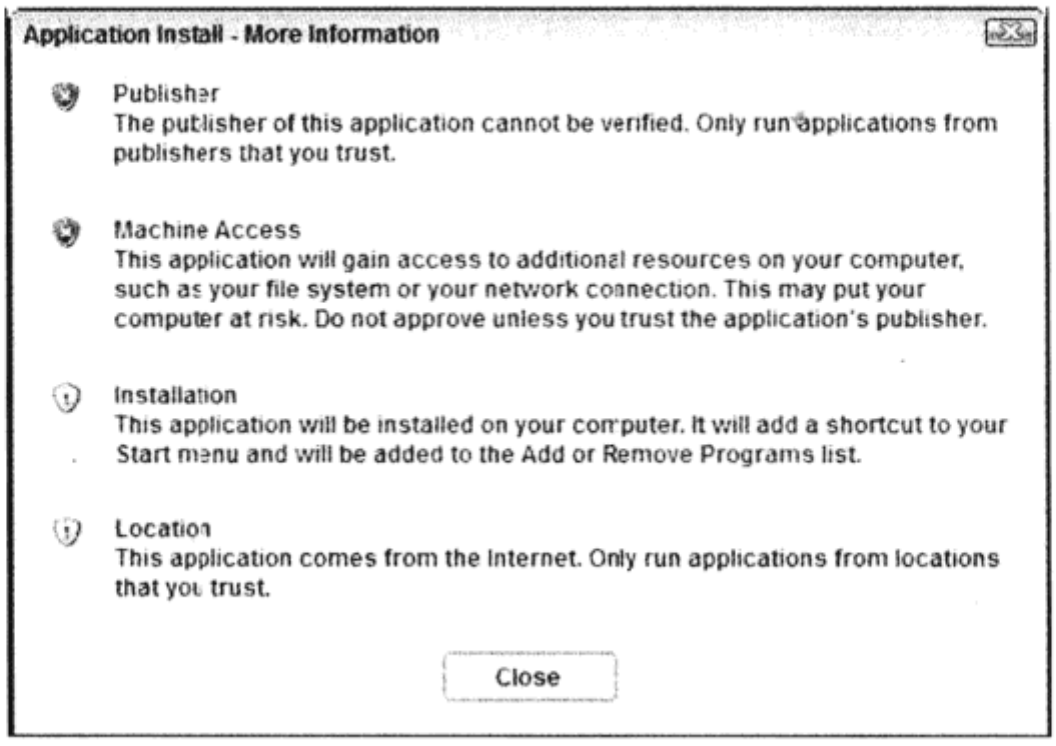


图　17-15

(4) 阅读完对话框的信息后，单击 Close 按钮，如果信任所创建的应用程序，就单击 Application Install 对话框中的 Install 按钮。

示例的说明

打开文件 publish.htm 时，会为目标应用程序检查是否存在第 4 版的.NET 运行库。这个检查由 HTML 页面中的一个 JavaScript 函数来进行。如果运行库未安装，就在安装客户应用程序之前安装运行库。利用默认的发布设置，将从 Microsoft 站点上复制运行库。

单击安装应用程序的链接时，会打开部署清单，以安装应用程序。接着告诉用户应用程序存在的一些可能的安全问题。如果用户单击 OK 按钮，就安装应用程序。

17.2.3 创建和使用应用程序的更新包

有了前面配置的更新选项，客户应用程序会自动检查 Web 服务器中是否有新版本。下面的“试一试”示例将对 MDI Editor 应用程序进行这项检查。

试一试：更新应用程序

(1) 修改 MDI Editor 应用程序，例如，设置 frmEditor.cs 文件中多格式文本框的背景色。

(2) 在项目属性中选择 Publish 部分，验证发布版本号改为一个新值。

(3) 生成应用程序，在项目属性的 Publish 部分单击 Publish Now 按钮。

(4) 不要单击 Web 页面上的 publish.htm 链接，而是从 Start 菜单中启动应用程序。在应用程序启动后，就会弹出如图 17-16 所示的 Update Available 对话框，询问是否要下载新版本。单击 OK 按钮，下载新版本。新版本下载完毕后，就会看到带有彩色多格式文本框的应用程序。

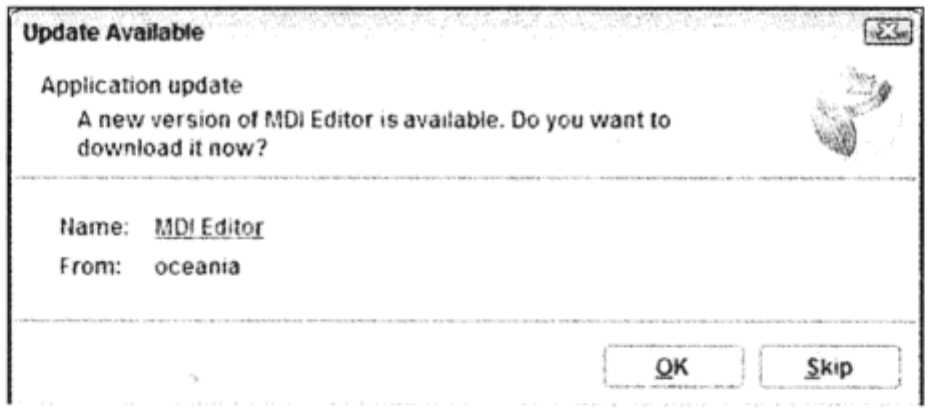


图 17-16

示例的说明

由部署清单中的一个设置和 XML 元素<update>来定义更新策略。使用 Updates 按钮和 Publish 设置可以改变更新策略。一定要使用项目的这些属性来访问 Publish 设置。Application Updates 对话框如图 17-17 所示。

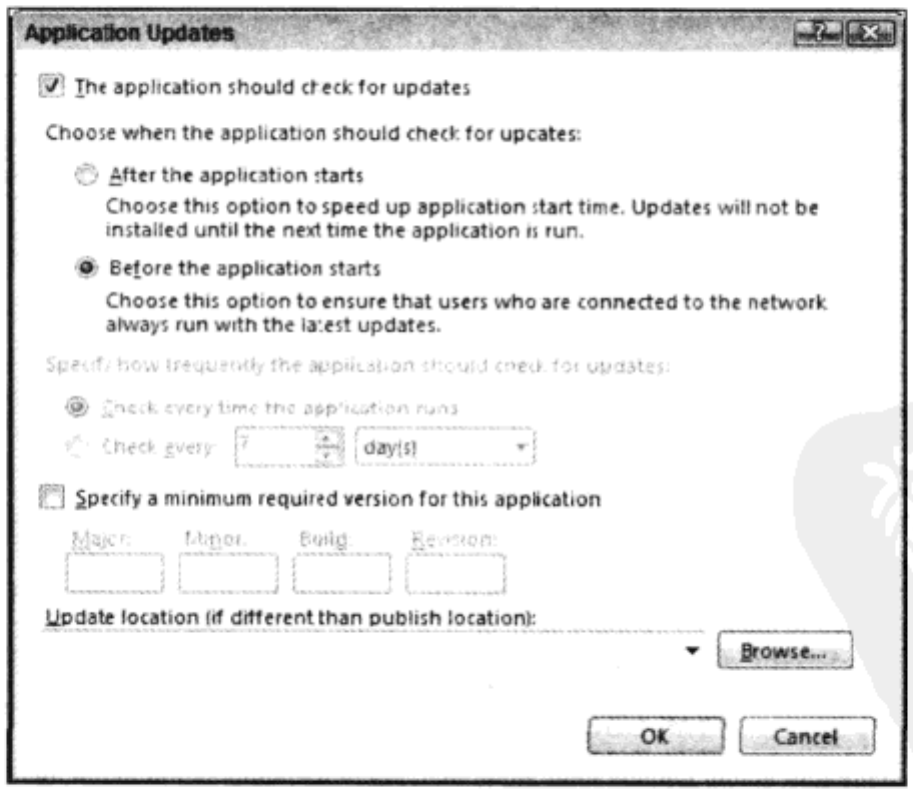


图 17-17

利用这个对话框可以定义客户机是否要查找更新版本。如果要查找更新版本，就可以定义查找是在应用程序启动之前进行，还是在应用程序运行过程中在后台进行更新。如果在后台进行更新，就可以设置更新的时间间隔：是每次启动应用程序时更新，还是每隔特定的小时数、天数或星期数更新一次。

17.3　Visual Studio 安装和部署项目类型

使用菜单打开 Visual Studio 的 Add New Project 对话框，从 Other Project Types | Visual Studio Installer 类别的 Installed Templates 窗格中选择 Setup and Deployment 选项之后，就可以看到如图 17-18 所示的窗口。

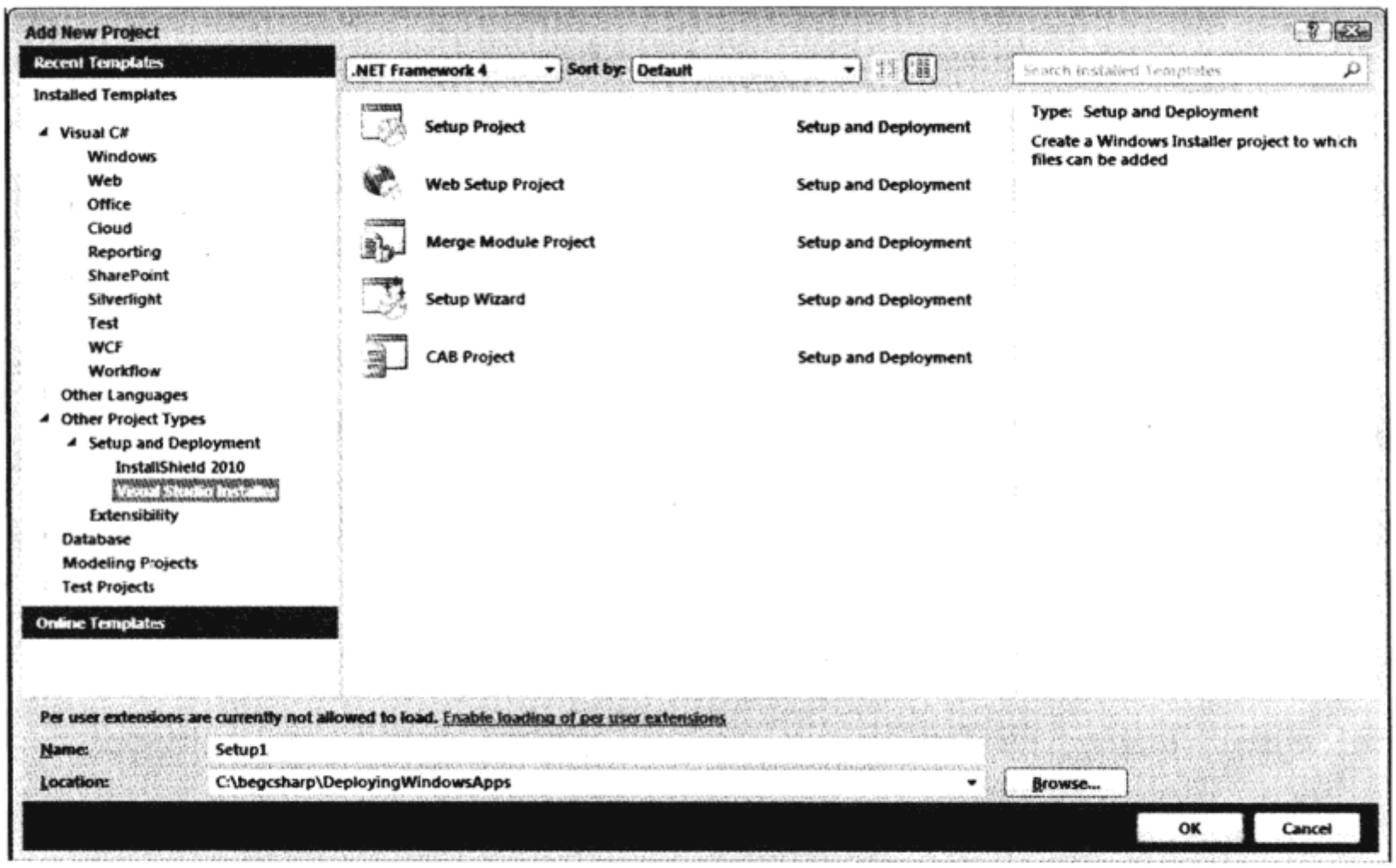


图　17-18

下面是项目类型以及它们的作用：

- 我们要使用 Setup Project 模板。此模板用于创建 Windows 安装软件包，所以可以用来部署 Windows 应用程序。
- Web Setup Project 模板用于安装 Web 应用程序，这个项目模板将在第 20 章使用。
- Merge Module Project 模板用于创建 Windows Installer 合并模块。合并模块(merge module)是安装程序文件，可以包括在多个 Microsoft Installer 安装软件包中。对于随多个安装程序一起安装的组件而言，可以创建一个合并模块，以在安装软件包中包括此模块。合并模块

的一个示例是.NET 运行库本身：它在合并模块中提供，因此可以在应用程序的安装软件包中包括.NET 运行库。在示例应用程序中将要使用一个合并模块。

- Setup Wizard是选择其他模板的一种方式。需要回答的第一个问题是：是不要创建一个安装程序，以安装应用程序或可供重新分发的软件包？根据不同的选择，可以创建 Windows 安装软件包、合并模块或 CAB 文件。
- Cab Project 模板允许创建 cabinet 文件。Cabinet 文件可以用于将多个程序集合并到一个文件中，并进行压缩。因为 Cabinet 文件可以压缩，所以 Web 客户机可以从服务器上下载较小的文件。

17.4 Microsoft Windows 安装程序结构

在 Windows Installer 推出之前，程序员必须创建定制的安装程序。生成安装程序要做大量繁琐的工作，而且许多程序并不遵循 Windows 规则。通常要改写旧版本的系统 DLL，因为安装程序不会检查版本。另外，应用程序文件的复制目录通常是错误的。例如，如果使用硬编码的目录字符串 C:\Program Files 但系统管理员改变了默认驱动器号，或使用了操作系统的国际化版本(其中此目录的命名方式各不相同)，安装就会失败。

Windows Installer 的第一个版本随 Microsoft Office 2000 发布，它也可以作为可分发软件包发布，这些可分发软件包可以包含在其他应用程序软件包中。Windows Installer 版本 1.1 新增了对注册 COM+组件的支持，版本 1.2 支持 Windows ME 的文件保护机制。版本 2.0 新增了对.NET 程序集安装和 Windows 64 位版本的支持。而在.NET 4 中，允许使用的 Windows Installer 最低版本是 3.1。

17.4.1 Windows 安装程序术语

使用 Windows 安装程序时，必须理解 Windows 安装程序技术使用的一些术语：软件包、功能和组件。

在 Windows 安装程序的环境中，组件与.NET Framework 使用的术语“组件”不同。Windows 安装程序组件仅仅是一个文件(或者是在逻辑上作为一个整体的多个文件)。这些文件是可执行文件、DLL 或简单的文本文件。

如图 17-19 所示，软件包包含一个或多个功能块。软件包是单一的 Microsoft 安装程序(MSI)数据库。功能是用户眼中的产品功能，由多个特性和组件构成。组件是开发人员从安装的角度来看的；它是最小的安装单元，由一个或多个文件组成。区分功能和组件的原因是单一的组件可以包含在多个功能中，如图 17-19 中的组件 2。一个功能不能包含在多个功能中。

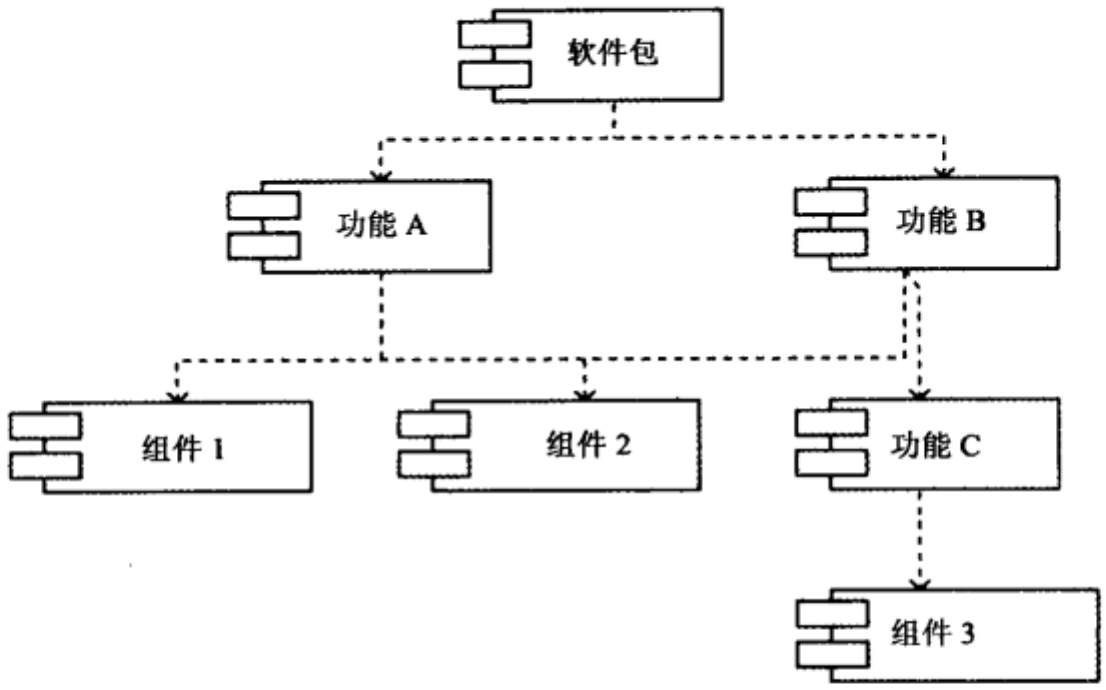


图　17-19

下面介绍一个功能的实际示例：Visual Studio 2010。使用控制面板中的 Programs and Features 选项，单击工具栏上的 Uninstall/Change 按钮，可以在安装之后改变 Visual Studio 的已安装功能，如图 17-20 所示。

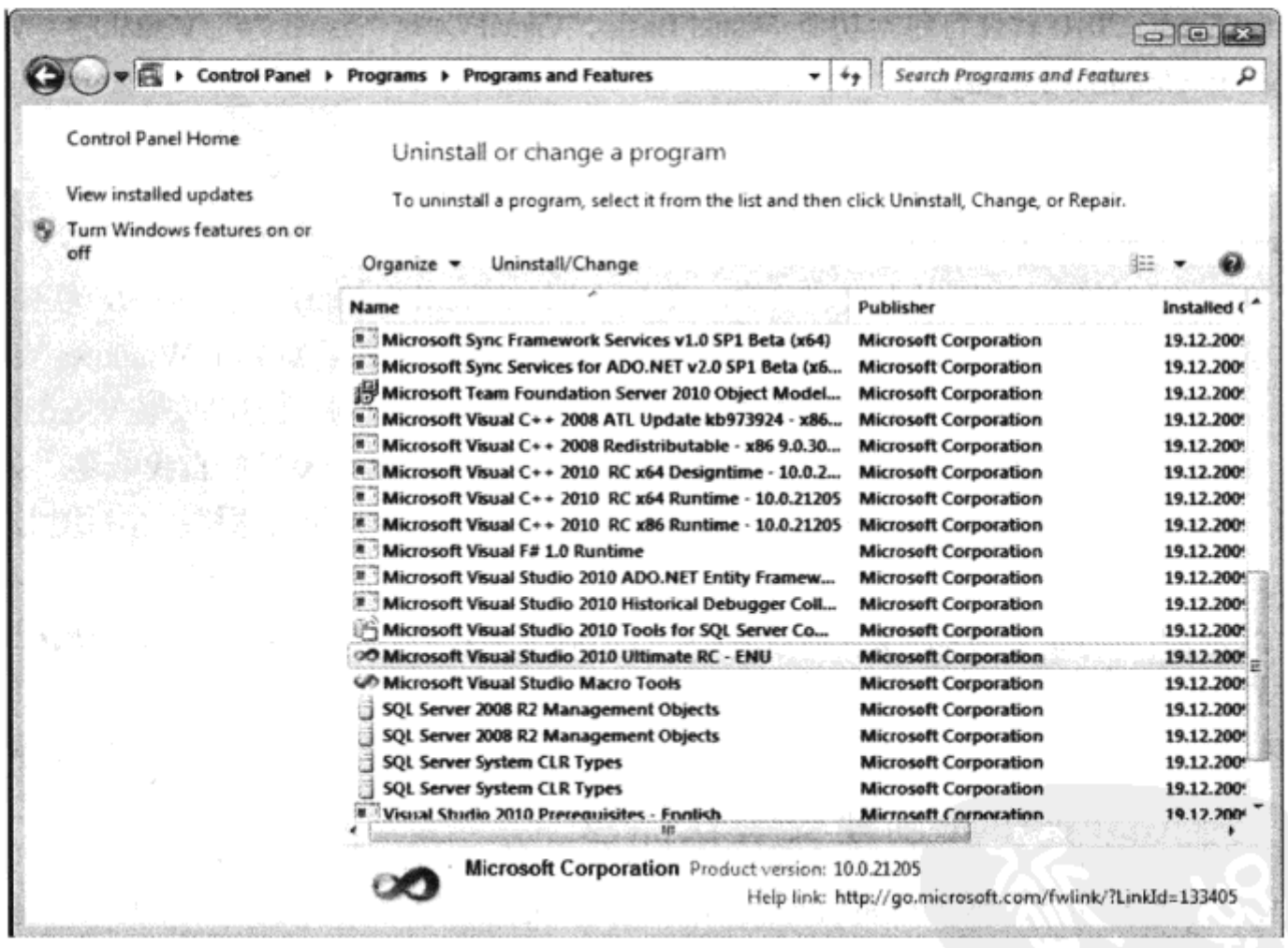


图　17-20

如图 17-21 所示，单击 Uninstall/Change 按钮，可以访问 Visual Studio 2010 Maintenance 向导，这是查看正在使用的功能的有效方法。单击左边树形视图中的加号和减号，就可以查看 Visual Studio 2010 软件包的所有功能。

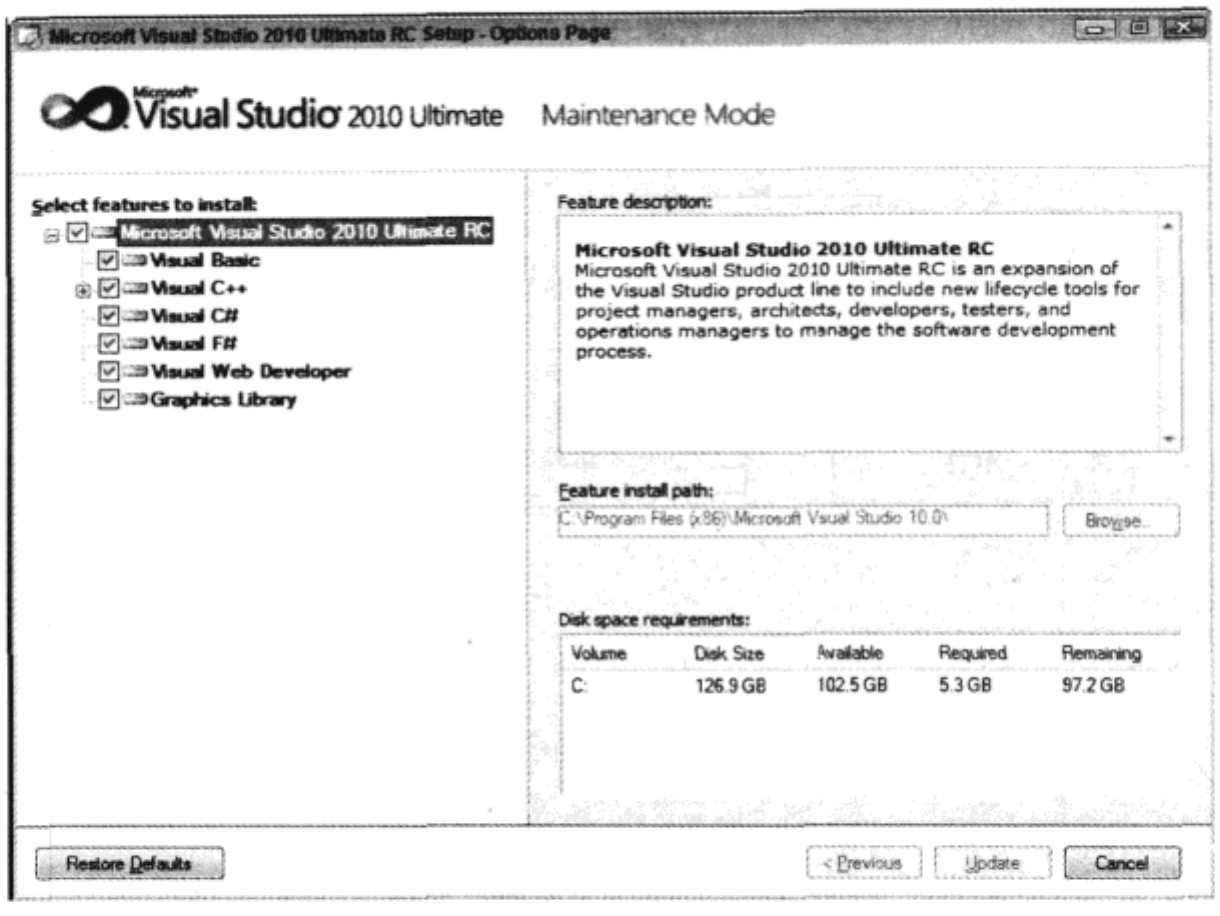


图 17-21

Visual Studio 2010 软件包包括功能 Visual Basic、Visual C++、Visual C#、Visual F#、Visual Web Developer 和 Graphics Library。

17.4.2 Windows 安装程序的优点

Windows 安装程序的优点如下：

- 可以安装功能，也可以不安装功能，或进行通知(advertise ment)。有了通知，软件包的功能会在首次使用时安装。或许您在使用 Microsoft Word 时已经注意到了 Windows 安装程序的用法。如果未安装 Word 的通知功能，只要您使用此功能，就会自动安装 Word。
- 如果应用程序受损，可以通过 Windows 安装程序软件包的修复功能自我修复。
- 如果安装失败，就会自动回滚。安装失败之后，所有内容都保持原样：在系统上没有附加的注册表项和文件等。
- 使用卸载功能，可以删除所有的文件，注册表项等。这样就可以完全卸载应用程序。不会留下临时文件，注册表也可以恢复原样。

阅读 MSI 数据库文件的表，可以获取以下信息：复制了什么文件，写入了什么注册表项等。

17.5 为 MDI Editor 创建安装软件包

本节将使用第 16 章的 MDI Editor 解决方案，通过 Visual Studio 2010 创建 Windows 安装软件包。当然，在执行这些步骤时，也可以使用以前开发的其他 Windows 窗体或 WPF 应用程序；只要改变一些名称即可。

17.5.1 规划安装内容

在开始生成安装程序之前，必须规划安装内容。首先要考虑一些问题：

- **应用程序需要什么文件**？当然是可执行文件和一些组件程序集。无需标识这些项之间的依赖关系，因为这种依赖关系会自动包括。或许还需要其他一些文件。如文档文件、readme.txt、许可文件、文档模板、图像和配置文件等。我们必须了解所有需要的文件。

对于第 16 章开发的 MDI Editor 应用程序而言，需要可执行文件，还要包含文件 readme.rtf、license.rtf 和显示在安装对话框中的 Wrox Press 位图。

- **应该使用什么目录**？应用程序文件应该安装在 Program Files\Application name 中。Program Files 目录的命名因操作系统使用的语言而异。而且，管理员也可以为此应用程序选择不同的路径。无需知道此目录的位置，因为 API 函数调用可以获取此目录。有了此安装程序，我们就可以使用一个预定义的特定文件夹在 Program Files 目录中放置文件。

任何情况下目录都不应该是硬编码的。对于国际版本，这些目录可以有不同的命名！即使应用程序仅支持 Windows 的 English 版本(实际上不会这样)，系统管理员也可以将这些目录移到不同的驱动器中。

MDI Editor 应用程序将可执行文件放在默认的应用程序目录中，除非安装用户选择了另一条路径。

- **用户如何访问应用程序**？可以在 Start 菜单中为可执行文件设置快捷方式，在桌面上放置图标等。如果希望在桌面上放置图标，就应该考虑用户是否乐意。对于 Windows XP，其原则是尽可能地使桌面干净。在 Windows 7 中，用户可以在桌面上放置小组件(活动的小程序)，这是桌面应整洁，用户应该根据需要安排图标和 gadget 的一个原因。MDI Editor 应该可以从 Start 菜单上访问。
- **分发介质是什么**？希望将安装软件包放在 CD、软盘或网络共享中吗？
- **用户应回答什么问题**？用户应接受许可信息，查看 ReadMe 文件，并输入安装路径吗？安装需要其他选项吗？

Visual Studio 2010 Installer 提供的默认对话框足以满足本章后面创建的 Windows Installer 项目的要求。我们会要求用户提供安装程序的目录(用户可以选择不同于默认路径的路径)，显示 ReadMe 文件，并要求用户接受许可协议。

17.5.2　创建项目

在了解了安装软件包的内容之后，就可以使用 Visual Studio 2010 安装程序来创建安装程序项目，并添加所有应该安装的文件。下面的示例会使用 Project 向导，配置项目。

试一试：创建 Windows Installer 项目

(1) 打开第 16 章创建的 MDI Editor 项目的解决方案文件。在现有解决方案中添加安装项目。如果没有在第 16 章创建该解决方案，可以从 Chapter16Code.zip 文件中复制完整的文件夹 MDI Editor。在 Visual Studio 中使用 File | Open | Project/Solution 菜单，选择文件夹 MDI Editor 中的解决方案文件 Manual Menus.sln，打开项目。

(2) 添加一个安装项目 MDIEditorSetup：选择 File | Add New Project 菜单，再选择 Other Project Types | Setup and Deployment | Visual Studio Installer，最后选择 Setup Project 模板，如图 17-22 所示，