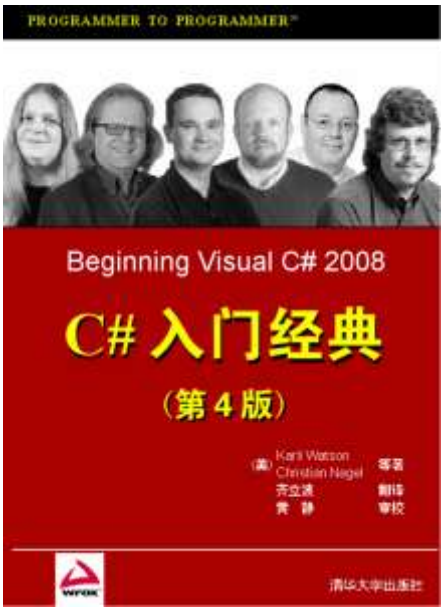

C#入门经典

（第 4 版）

(美) Karli Watson Christian Nagel 等著

齐立波 翻译

黄 静 审校

清华大学出版社

北 京

Karli Watson，Christian Nagel，et al.
Beginning Microsoft Visual C# 2008
EISBN：978-0-470-19135-4

北京市版权局著作权合同登记号　图字：01-2008-2590

图书在版编目(CIP)数据

C#入门经典(第 4 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，2008.12
书名原文：Beginning Microsoft Visual C# 2008
ISBN 978-7-302-18587-1

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(2008)第 145131 号

责任编辑：王　军　李　阳
装帧设计：孔祥丰
责任校对：成凤进
责任印制：
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：
装 订 者：
经　销：全国新华书店
开　本：185×260　印　张：64.25
　字　数：1604 千字
版　次：2009 年 1 月第 1 版　印　次：2009 年 1 月第 1 次印刷
印　数：1~5000
定　价：118.00 元

本书卖点：

1．Wrox 畅销书，C#经典名著！是2006～2008 年最畅销的 C#入门教程。

2．自第 1 版出版以来，全球销量达数十万册，在中国就有尽 8 万册。已成为广大初级 C＃程序员首先的入门教程，也是目前国内市场上最畅销的 C＃专业店销书。

3．曾获得过国内外多项大奖。2005 年被评为“最权威的十大 IT 图书”；2006 年被 CSDN、《程序员》等机构和读者评为“最受读者喜爱的十大技术开发类图书”。

4．2008 年在“第八届全国高校出版社优秀畅销书”评选活动中获得一等奖！

5．第 4 版面向 C# 2008 和.NET 3.5，继续完善上一版本内容，同时全面介绍.NET 的最新技术和特性。

内容简介：

本书全面阐述了 C#编程的所有方面，包括 C#语言本身、Windows 编程、Web 编程及数据源的使用等内容。

学习了新的编程技巧后，本书介绍了如何高效地部署应用程序和服务，论述了许多高级技术，如图形化编程。另外，还探讨了如何使用 Visual C# Express 2008、Visual Web Developer Express 2008 和 Visual Studio 2008 的功能。所有这些内容都已更新，以反映.NET Framework 3.5 和 Visual Studio 2008 的变化。各章的样例代码和示例还可以用于创建强大且安全的应用程序。

本书语言简练，条理清晰，涵盖了 Visual C# 2008 和.NET Framework 3.5 的全部内容。通过学习本书，您将能够快速入门，轻松开发出自己的应用程序。

本书主要内容：

- C#的所有基础知识和面向对象编程
- 通过 C#语言编写和部署 Windows 应用程序
- 开发定制的 Web 应用程序和利用 Web 服务的技巧
- 数据访问技术，包括使用 LINQ 处理 XML 数据
- 通过单击按钮将应用程序发布到 Web 上
- 使用 Windows Presentation Foundation、Windows Workflow Foundation、Windows Communication Foundation、GDI+和联网等其他技术

本书读者对象

本书适合于想学习使用.NET Framework 编写 C#程序的读者，也适合于已了解.NET，又想学习.NET 3.5 或 Visual Studio 2008 最新功能的读者。

源代码下载

http://www.wrox.com

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

译者：

齐立波，计算机专业硕士、大学讲师，一直从事 C/C++和 C＃程序设计方面的教学与科研工作，曾参与编写多本计算机教

材，并为一些软件公司翻译过大量的技术文档。

审校：

黄静，计算机专业博士，在程序设计、软件工程和网络安全等方面有着多年的研究和实战经验，曾为.NET 技术在国内的推广作出了重要贡献，翻译和发表了多篇有关.NET 的技术文档和文章。

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#的快速流行，使之成为使用.NET Framework 的 Windows 和 Web 开发人员无可争议的语言选择。他们喜欢 C#的一个原因是其派生于 C/C++的简洁语法，这种语法简化了以前困扰一些程序员的问题。尽管做了这些简化，但C#仍保持了C++原来的功能，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 3.0 是.NET Framework 3.5 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。其中一些功能取自于 C++(至少从表面上看是这样)，而另外一些功能是全新的。Visual Studio 的最新版本和开发工具的 Express 系列也有许多变化和改进，这大大简化了编程，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，最后是一些高级技术，例如，图形编程。我们还将学习 Visual C# Express 2008、Visual Web Developer Express 2008 和 Visual Studio 2008 的功能和利用它进行应用程序开发的各种方式。本书界面友好，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。本书尽量不使用技术术语，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，希望读者能通过本书成为最优秀的程序员。

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一样的(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，然后快速跳读后面的几章，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，但以前没有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# 3.0 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。

本书章节的编写目的有两个：这些章节可以按顺序阅读，是 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，每一章还包含一组练习，完成这些练习有助于读者理解所学习的内容。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。练习的答案在 www.wrox.com 的本书 Web 页面上在线提供。

0.2 本书结构

本书分为 6 个部分。

- **引言**：概述本书的内容和作者。
- **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，应有尽有。
- **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**：介绍在应用程序中如何使用数据，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，包括属性、XML 文档说明、网络和 GDI+图形编程。还探讨了由.NET 3.0 引入且由.NET 3.5 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。

0.2.1 C#语言(第 1~14 章)

第 1 章介绍 C#以及它与.NET 的关系，了解在这个环境下编程的基础知识，以及 Visual C# Express(VCE)和 VS 与它的关系。

第 2 章开始介绍 C#应用程序开发，学习 C#的语法，并把 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使用的基本窗口和工具。

接着，第 3 章学习 C#语言的更多基础知识，理解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 章介绍一些高级的变量类型，如数组。第 6 章开始以函数的形式封装代码，这样就更易于执行重复的操作，代码更容易理解。

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？”。这个 OOP 初看起来是比较难的问题。我们将用整整一章的篇幅来介绍它，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，而这正是 C#的强大之处。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#的一个非常有用的新特性——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)完成 C#语言和 OOP 的讨论。最后，第 14 章介绍 C# 3.0 中引入的新特性。

0.2.2 Windows 编程(第 15~18 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，并构建知识体系。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 如何以图形化的方式建立 Windows 应用程序，以最少的时间和精力创建高级应用程序。

第 17 章介绍一些常用的功能。利用这些功能可以方便地添加专业功能，例如，文件管理、打印等。第 18 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。

0.2.3 Web 编程(第 19~23 章)

这个部分的结构与 Windows 编程部分类似。首先，第 19 章描述了构成最简单的 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。第 20 章以此为基础，介绍了更高级的技术、各种控件、Web 环境下的状态管理，以及 Web 标准的遵循。

第 21 章将涉足 Web 服务的精彩世界，它可以编程访问 Internet 上的信息和功能，可以把复杂的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。

第 22 章介绍 Ajax 编程，这种方式可以给 Web 应用程序添加动态的客户端功能。.NET Framework 3.5 通过 ASP.NET Ajax 提供了 Ajax 功能，本章会解释如何使用它。

最后，第 23 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 新特性。

0.2.4 数据访问(第 24~29 章)

第 24 章介绍了应用程序如何保存和检索磁盘上的数据，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，用逗号隔开的值(CSV)文件)，如何监视和处理文件系统的变化。

第 25 章学习数据交换的事实标准 XML。前面的章节接触过 XML 几次，而这一章将了解 XML 的基本规则，论述 XML 的所有功能。

本部分的其余章节介绍 LINQ，这是内置于.NET Framework 最新版本中的查询语言。第 26 章简要介绍 LINQ，第 27 章使用 LINQ 访问数据库中的数据。第 28 章介绍如何联合使用 LINQ 和旧的 ADO.NET 数据访问技术。最后，第 29 章学习如何使用 LINQ 处理 XML 数据。

0.2.5 其他技术(第 30~36 章)

本书的最后一部分将介绍 C#和.NET 主题的其他技术。第 30 章将探讨属性，可以在程序集中包含类型的其他信息，添加用其他方式很难实现的功能。

第 31 章研究 XML 文档说明，并介绍如何在源代码中给应用程序添加注释。我们将学习如何添加这些信息，如何使用和提取它们，从而从代码中生成 MSDN 样式的文档说明。

第 32 章介绍网络，应用程序如何相互通信，如何与各种网络上的其他服务通信。第 33 章从本书前面学习的各种技术中解脱

出来，研究用 GDI+进行图形编程的主题，理解如何操纵图形，设定应用程序的样式，这一章打开了一条通往各种 C#应用程序的大门。

最后要讨论.NET Framework 最新版本中的几个新技术。第 34 章介绍 Windows Presentation Foundation(WPF)，了解它给 Windows 和 Web 开发带来了哪些变化。第 35 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的通信技术。本书的最后一章是第 36 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，以特定的顺序执行，这对许多类型的应用程序都很有帮助。

0.3 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 3.5。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 Visual C# Express 2008 作为主要开发工具，一些章节则使用了 Visual Web Developer Express 2008。另外，一些功能只能在 Visual Studio 2008 中使用，这会在相应的地方明确指出。

0.4 源代码

在读者学习本书中的示例时，可以手工输入所有的代码，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，就可以获得所有的源代码。

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 978-0-470-19135-4。

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。

0.5 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。

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。

0.6 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按扭。

(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按扭。

(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 3 章

变量和表达式

要想高效地学习 C#的用法，重要的是理解创建计算机程序时需要做什么。计算机程序最基本的描述也许是一系列处理数据的操作，即使是最复杂的示例，这个论述也正确，例如，Microsoft Office 套装软件之类的大型多功能的 Windows 应用程序。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。

如果计算机程序是在对数据执行操作，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。

本章的主要内容：

C#的基本语法

变量及其用法

表达式及其用法

在开始之前，应先了解一下 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。但是，在 C#编程中，使用的样式是比较清晰的，不用花太多的力气就可以编写出可读性很强的代码。

与其他语言的编译器不同，无论代码中是否有空格、回车符或 tab 字符(这些字符统称为空白字符)，C#编译器都不考虑这些字符。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。因为空格被忽略，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，这样就不能在一行上放置多个语句了。但一句代码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*>部分并不是真正的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。

在这个简单的代码块中，还使用了缩进格式，使C#代码的可读性更高。这是一个标准规则，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。

注释：

在能通过 Tools | Options 访问的 VCE Options 对话框中，显示了 VCE 用于格式化代码的规则。在 Text Editor | C# | Formatting 节点的子目录下，包含了完整的格式化规则。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，所有的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。注释并不是严格意义上的 C#代码，但代码最好有注释。注释就是解释，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。在开始处理比较长的代码段时，注释可用于给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，可以在注释的开头加上“/*”，在末尾加上“*/”。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! */
```

但下面的语句会产生错误：

```
/* 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。

另一个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
```

前面说过有两种给 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
```

在正常情况下，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。具体内容见第 31 章。

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 在代码的输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。

在输入代码的过程中，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，更准确地说，是运行花括号中的代码块。如前所述，注释行不做任何事情，包含它们只为了简洁而已。其他两行代码在控制台窗口中输出了一些文本，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 语句将在本章的后面解释。

注释：

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。以后会阐述这些代码的重要性。

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，可以避免混淆它们。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。声明了变量后，就可以把它们用作存储单元，存储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，代码就不会编译，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，存储各种复杂的数据。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组成部件的类型，例如，数值和布尔值(true 或 false)。简单类型与复杂类型不同，不能有子类型或属性。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
```

如果要存储更大的数，就需要更多的位(例如，3 位可以存储 0~7 的数)。

这个论点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－32768~32767 之间的整数
ushort	System.UInt16	在 0 ~65535 之间的整数
int	System.Int32	在－2147483648~2147483647 之间的整数
uint	System.UInt32	在 0~4294967295 之间的整数
long	System.Int64	在－9223372036854775808~9223372036854775807 之间的整数
ulong	System.UInt64	在 0~18446744073709551615 之间的整数

注意：

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号，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 的值随着类型的不同而不同。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	224	－149	104	1.5×10^{-45}	3.4×10^{38}
double	System.Double	0	253	－1075	970	5.0×10^{-324}	1.7×10^{308}
decimal	System.Decimal	0	296	－26	0	1.0×10^{-28}	7.9×10^{28}

除了数值类型外，还有另外 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 0~65535 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，下面用一个小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。

试试看：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) 给 Program.cs 添加如下代码：

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

(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。

提示：

变量的命名是有限制的，不能使用任意的字符序列。本节的后面将介绍变量的命名规则。

接下来的两行代码给变量赋值：

```
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引号字符的其他字符(即转义序列)来替代它们。在本例中，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回车换行符的转义序列，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，有如下两类参数：

一个字符串

一个用逗号分隔开的变量列表，这些变量的值将插入到输出字符串中

输出的字符串是"{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。整数以 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：

```
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
```

下列变量名不正确：

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

记住，C#是区分大小写的，所以必须小心，不要忘了在声明变量时使用正确的大小写。在程序中引用它们时，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 记号法。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，说明该变量的类型。

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 camelCasing 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

在本章的前面，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 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."
```

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

显然，Unicode 转义序列还有更多的用途。

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。唯一的例外是双引号字符的转义，它们必须指定，以避免结束字符串。为此，可以在该字符串的前面加一个@字符：

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

这个字符串可以用一般的方式指定，但需要使用下面这种方式：

```
@"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，这段代码的可读性就比较高。下面的字符串与上面的等价：

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

注意：

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

快速回忆一下，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;
```

注意：

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，包括前面已经使用过的=赋值运算符。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，它是计算的基本建立块。

运算符的范围非常广泛，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，而复杂的操作则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 2 个有二元和一元两种形式。表 3-6 列出了这些运算符，并用一个小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 的值除乘以 - 1

注释：

+(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。如果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)，编译器会报告出错。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 中的字符串的连接值

但其他数学运算符不能用于字符串的处理。

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，考虑下面的代码：

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

要把什么值赋予 var1？在表达式计算前，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。代码提示用户输入一个字符串和两个数字，然后显示计算结果。

试试看：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();
}
```

(3) 执行代码，结果如图 3-2 所示。

(4) 输入名称，按下回车键，如图 3-3 所示。

图　3-2

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。

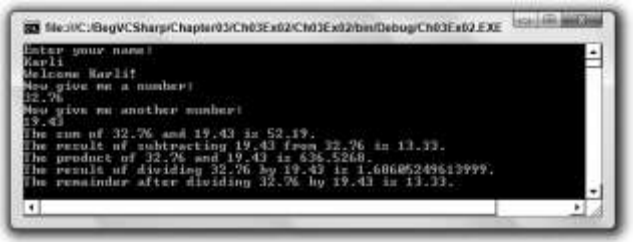

图　3-4

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的概念，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);
```

这段代码把已赋值变量 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这种方式进行。

其余的代码以相同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);
```

这种语法可以使代码的可读性比较强，可以减少需要编写的代码量。

3.4.2　赋值运算符

直到现在，我们一直在使用简单的=赋值运算符，其实还有其他赋值运算符，而且它们都非常有用。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;
```

注意：

+=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，可以使代码更容易阅读。

3.4.3　运算符的优先级

在计算表达式时，每个运算符都会按顺序处理。但这并不意味着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;
```

括号中的内容先计算，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)

注意：

括号可用于重写优先级顺序，如上所述。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)。

在默认情况下，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继承信息。基本上，这意味着，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。限定名称在不同的名称空间级别之间使用句点字符(.)。

例如：

```
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。

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。这最好用一个示例来说明。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 必须被引用为 LevelOne.LevelTwo.NameTwo，在 LevelOne 名称空间中，则可以被引用为 LevelTwo.NameTwo，在 LevelOne.LevelTwo 名称空间中，则可以被引用为 NameTwo。

要注意的是，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，就可以使用 using 语句简化对它们包含的名称的访问。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 名称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，代码是不能编译的，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名称空间中。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
   }
}
```

这样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。

这里有一点要注意：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，使之更合理。

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.Text;

namespace ConsoleApplication1
{
   ...
}
```

以 using 关键字开头的 3 行代码声明在这段 C#代码中使用 System、System.Collections.Generic 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 应用程序的根名称空间，包含控制台应用程序所需要的所有基本功能。其他两个名称空间常常用于控制台应用程序，所以该程序包含了这 3 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#应用程序的许多基础知识，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。

本章的主要内容是变量的使用。我们描述了变量，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。

本章还介绍了如何把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，

以及执行它们的顺序。

最后介绍了名称空间。随着本书内容的深入，名称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，完整的论述见后面的内容。

本章学习了：

C#的基本语法

在创建控制台应用程序项目时 VS 所做的工作

变量的理解和使用

表达式的理解和使用

名称空间的含义

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，使代码的效率更高。

3.6　练习

(1) 在下面的代码中，如何引用名称空间 fabulous 中的名称 great？

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

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

(2) 下面哪些不是合法的变量名?

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：可以考虑使用 Convert.ToDouble()命令，该命令可以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 C

第 4 章

流 程 控 制

到目前为止，我们看到的 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。如果所有的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程的两种方法。程序流程就是 C#代码的执行顺序。这两种方法是：

分支——有条件地执行代码。条件取决于计算的结果，例如，“如果 myVal 小于 0，就只执行这行代码”。

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，但没有讨论它。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。

本章的主要内容：

布尔逻辑的含义及其用法

如何控制代码的执行

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的结果。

注意：

英国数学家乔治·布尔在 19 世纪中叶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，即需要知道比较的布尔结果。

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 则可以是各种类型。

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

(续表)

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

在代码中，可以对数值使用这些运算符，如下所示：

```
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。

注意：

假定 val1 < val2 是 false，则 val1 > val2 为 true，此时会产生一个错误。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)

(续表)

运 算 符	类　别	示例表达式	结　果
^	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，可能根本就不需要第二个操作数的值(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。它们的操作数总是要计算的。

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 | ，性能会有一定的提高。在大量使用这些运算符的应用程序中是比较明显的。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，后者是可能的，也是可以检测到的)。

4.1.1　位运算符

在上一节的讨论中，读者可能会问，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。

下面先讨论&。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 4-4 所示。

| 运算符与此类似，但得到的结果是不同的，如表 4-5 所示。

例如，考虑下面代码中的操作：

表　4-4

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　4-5

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

```
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：

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：

^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 4-6 所示。

表　4-6

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 4-7 所示。

表　4-7

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
```

提示：

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。

本节介绍的这些位运算符在某些情况下是非常有用的，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，进行如下配置，如表 4-8 所示。

表　4-8

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
```

最后一行代码把值 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，在一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 个位运算符外，本节还要介绍另外两个运算符，如表 4-9 所示。

表　4-9

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，最好用一个示例来说明：

```
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，就是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。

4.1.2　布尔赋值运算符

本节要介绍的最后一类运算符是把前面的赋值运算符组合起来，非常类似于第 3 章中的数学赋值运算符(+=, *=等)。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;	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。

注意：

&=和 |= 使用 & 和 |，而不是 &&和 ||，使用这些较简单的运算符会产生额外的开销。

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

下面看一个示例。这个示例让用户输入一个整数，然后代码使用该整数执行各种布尔运算。

试试看：使用布尔和位运算符

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

(2) 把下述代码添加到 Program.cs 中：

```
static void Main(string[] args)
{
  Console.WriteLine("Enter an integer:");
  int myInt = Convert.ToInt32 (Console.ReadLine());
  Console.WriteLine("Integer less than 10? {0}", myInt < 10);
```

```
            Console.WriteLine("Integer between 0 and 5? {0}",
                              (0 <= myInt) && (myInt <= 5));
            Console.WriteLine("Bitwise AND of Integer and 10 = {0}", myInt & 10);
            Console.ReadKey();
        }
```

(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。

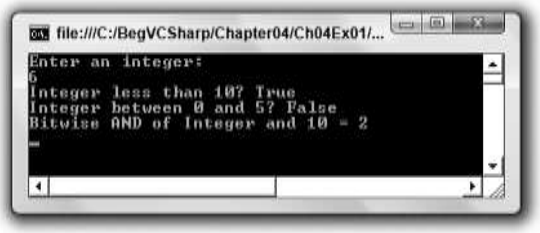

图　4-1

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()命令位于同一个系列中。

剩下的 3 行代码对得到的数字进行各种操作，并显示结果。在执行这段代码时，假定用户输入了 6，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。

第二个输出涉及的计算比较多：(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，最终的结果就是(true) && (false)，即 false，如图 4-1 所示。

最后，对 myInt 的值进行按位 AND 操作。另一个操作数是 10，它的二进制值是 1010。如果 myInt 是 6，其二进制值是 110，则这个操作的结果是 10，即十进制中的 2，如下所示：

```
  0 1 1 0     6
& 1 0 1 0   & 10
-----------------
  0 0 1 0     2
```

4.1.3　运算符的优先级更新

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。

这样表中增加了好几个级别，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。

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)。上面的表达式也可以写为：

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

要解决这个问题，可以明确指定计算的顺序。

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)

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。该语句有其优缺点。主要的优点是：这是控制什么时候执行哪些代码的一种非常简单的方式。主要的缺点是：过多地使用这个技巧将很难读懂代码。

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，把控制转到标为 myLabel:的代码行上。

myInteger 的值写到控制台上。

下面的第 3 行代码没有执行。

```
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，就会发现编译代码时，Error List 窗口会显示一个警告，即 Unreachable code detected 和一个行号。在无法执行的代码中，myInteger 下面还有绿色的波浪线。

goto 语句有它们的用途，但也可能使代码陷入混乱之中。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，但非常难读懂，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。后面再讨论这个语句，因为本章的其他一些结构将使用该语句(但最好不要使用它)。

4.3　分支

分支是控制下一步要执行哪些代码的过程。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

进行比较最简单的方式是使用第 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>。

三元运算符的示例如下所示：

```
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"。

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

4.3.2　if 语句

if 语句的功能比较多，是进行决策的有效方式。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，就执行该语句下面的代码。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，指定其他的代码。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 语句的结果不能被赋予一个变量，所以要单独把值赋给变量：

```
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，很容易阅读和理解，其灵活性也比较大。

试试看：使用 if 语句

(1) 在目录 C:\BegVCSharp\Chapter04 下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：

```
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();
}
```

(3) 执行代码，在提示行下输入两个数字，如图 4-2 所示。

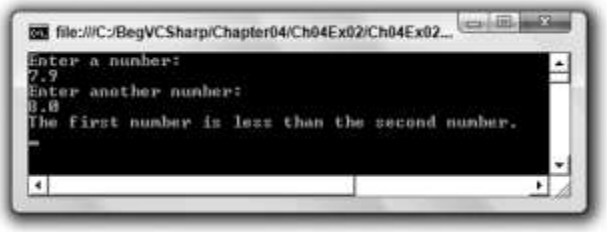

图　4-2

示例的说明

代码的第一部分我们已经很熟悉了，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，把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完成任务的一种方式，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";
```

这个方法的缺点是无论 var1 和 var2 的值是什么，都要执行 3 个比较操作。在第一种方

法中，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恼人的嵌套代码，例如：

```
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
      }
   }
}
```

注意：

在编写第 3 个条件时，常常会错误地将代码写为 if (var1 == 3 || 4)。由于运算符有优先级，因此==运算符先执行，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，就要使用略有不同的缩进模式，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的语句，它们的功能与上述代码相同。但是这样代码会更易于阅读。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。

注意：

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，还有另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。可以使用 return 语句，中断当前函数的运行，这远胜于中断 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)，其后加一行代码，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>都必须是一个常量。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后，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，因为它们通常很容易阅读，在比较时，看不到要比较的值。

除了变量类型外，还可以使用关键字 const 声明常量，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，试图用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，把不同的字符串写到控制台上。

试试看：使用 switch 语句

(1) 在目录 C:\BegVCSharp\Chapter04 下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：

```
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();
}
```

(3) 执行代码，输入一个姓名，结果如图 4-3 所示。

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。

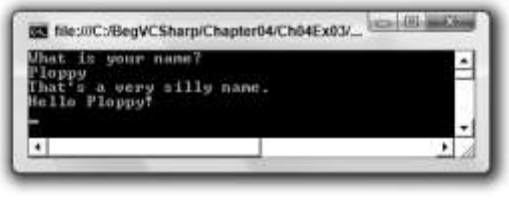

图　4-3

在比较输入的姓名(在变量 name 中)和常量时，首先要用 name.ToLower()把输入的姓名转换为小写。name.ToLower()是一个标准方法，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 语句尝试把输入的字符串与定义的常量相匹配，并输出一个个性化的信息。如果成功，就会用一个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，但这需要一定的时间。

4.4　循环

循环就是重复执行一些语句。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，假定计算每年的利息，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;
```

相同的代码编写 10 次很浪费时间，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略微简单一些(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。

提示：

while 语句后面的分号是必须有的。

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 年后的结余。这次使用一个循环，根据起始的金额和利率，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。

试试看：使用 do 循环

(1) 在目录 C:\BegVCSharp\Chapter04 下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：

```
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();
}
```

(3) 执行代码，输入一些值，结果如图 4-4 所示。

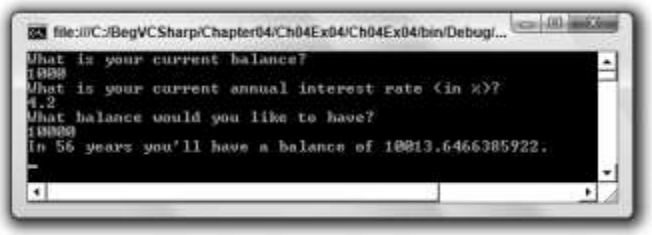

图　4-4

示例的说明

这段代码利用固定的利率，对年度计算结余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);
```

然后就可以使用这个计数器变量作为输出结果的一部分：

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

注意：

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。

但这段代码并不是完美的，考虑一下目标结余小于当前结余的情况，则结果应如图 4-5 所示。

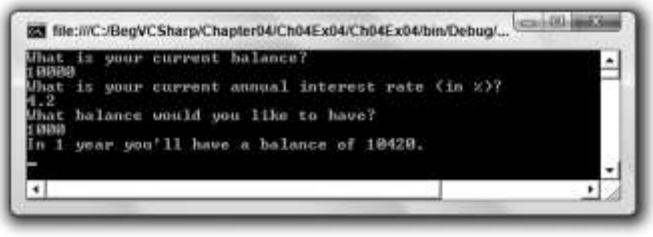

图　4-5

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);
```

但这看起来像是添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，但有一个重要的区别：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后面的代码。

while 循环以下述方式指定：

```
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。

试试看：使用 while 循环

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

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

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

(3) 再次执行代码，但这次使用小于起始结余的目标结余，如图 4-6 所示。

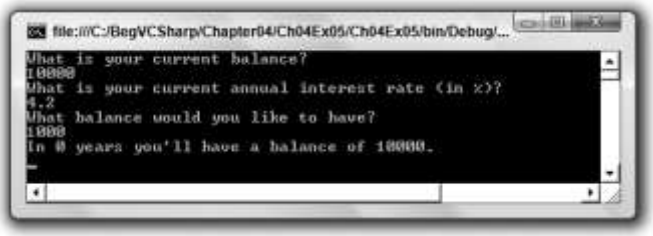

图　4-6

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，就不需要执行循环，而是直接跳转到输出结果上。

当然，这种情况还有另一个解决方案。例如，可以检查用户输入，确保目标结余大于起始结余。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。

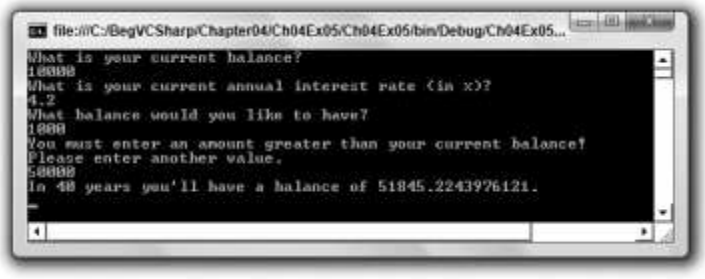

图　4-7

在进行应用程序的设计时，用户输入的有效性检查是一个很重要的问题，本书将介绍更多这方面的示例。

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，需要下述信息：

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，它的起始值是 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 for 循环才执行，所以循环中的代码可能根本就不会执行。

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，变量 i 就不能在循环外部使用(参见第 6 章中有关变量作用域的部分)。

下面介绍一个使用 for 循环的示例。它将显示一个 Mandelbrot 图像(使用纯文本字符，看起来不会那么吸引人)!

试试看：使用 for 循环

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

(2) 把下述代码添加到 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();
    }
```

(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 图像，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 图像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 坐标对应于虚数的 x 和 y 部分。

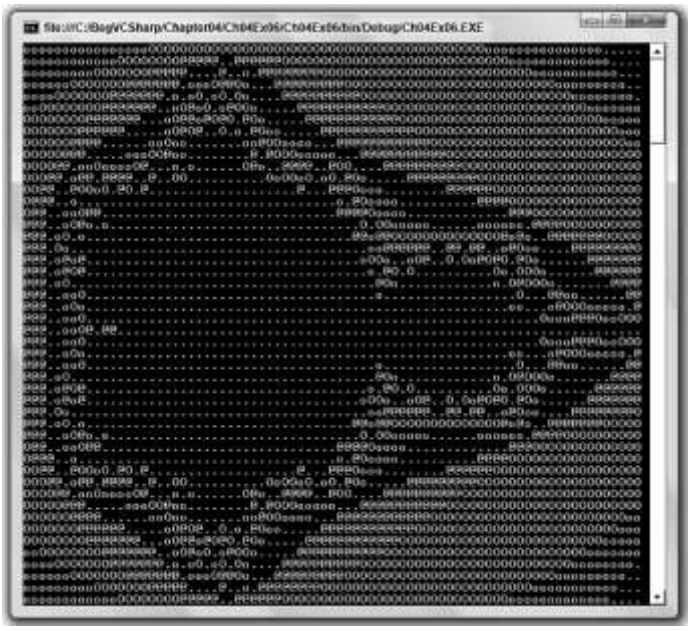

图　4-8

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，或迭代执行的次数多于指定的次数为止。

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，这是一种常见的强有力的技术)：

```
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 个不同的字符，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一行字符的结束符。

```
          }
          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

有时需要在循环代码的处理上有更精细的控制。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 时终止了循环。

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，定义永远不终止的循环，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 Task Manager 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。注意仍可以用正常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，这个问题是非常重要的。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。如果把所有的代码都放在屏幕的左端，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，让用户输入两个数字，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) 下面的代码有什么错误？

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

(4) 修改 Mandelbrot 图像 边界，显示选中的图像部分。当前代码输出的字符应正好能 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

第 11 章

集合、比较和转换

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：集合可以维护对象组。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强制转换为另一种类型。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进行这些处理的类，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，它们只是集合类中的一种。集合类一般用于处理对象列表，其功能比简单数组要多，这些功能是通过实现 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(包括基本函数，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，获得特定 suit 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，并可以访问这些项，以及其他一些与项列表相关的功能。

IDictionary(继承于 IEnumerable 和 ICollection)类似于 IList，但提供了可通过键码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，但实现的方式比 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。

试试看：数组和高级集合

(1) 在目录 C:\BegVCSharp\Chapter11\下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：

```
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);
        }
    }
}
```

(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)
        {
        }
    }
}
```

(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)
        {
        }
    }
}
```

(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();
        }
    }
}
```

(7) 运行该应用程序，其结果如图 11-1 所示。

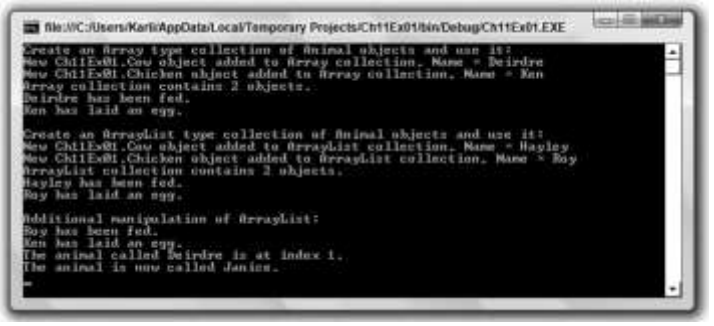

图　11-1

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的代码和结果，讨论一下这两种集合的类似操作。首先是集合的创建。对于简单的数组来说，必须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，所以可以使用下面的代码创建列表 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，容量就会自动增大一倍。

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"));
```

除了语法略有不同外，还可以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重写它们，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。后面将详细讨论这一点。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。它们分别根据项目引用或索引把项从数组中删除。在本例中，我们使用后一个方法删除添加到列表中的第一项，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，可以用下面的方式访问它：

```
((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 来访问 Chicken 对象，而不是 1。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

于给数组排序和重新排序的方法。

最后，再回过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，但这比较花时间，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，但只提供了一些要求的执行代码，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。

为了便于完成任务，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执行代码)：

```
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 执行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。这个语法是富有逻辑的，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，即可以编写下述代码：

```
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。

试试看：实现 Animals 集合

(1) 在目录 C:\BegVCSharp\Chapter11\下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) 从目录 C:\BegVCSharp\Chapter11\Ch11Ex01\Ch11Ex01 下选择 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 Ch11Ex02
```

(5) 添加一个新类 Animals。

(6) 修改 Animals.cs 中的代码，如下所示：

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

#endregion

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;
            }
         }
      }
   }
```

(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();
      }
```

(8) 执行应用程序，其结果如图 11-2 所示。

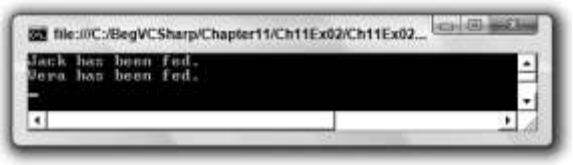

图　11-2

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。在目录 C:\BegVCSharp\Chapter11\ 下创建一个新的类库Ch11CardLib，再从 Project | Add Existing Item 中选择 C:\BegVCSharp\

Chapter10\Ch10CardLib\Ch10CardLib 目录下的 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“试试看”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。

注意：

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 Ch11CardLib 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;
```

```
            }
        }

        // 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.
        public void CopyTo(Cards targetCards)
        {
            for (int index = 0; index < this.Count; index++)
            {
                targetCards[index] = this[index];
            }
        }

        // 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.
        public bool Contains(Card card)
        {
            return InnerList.Contains(card);
        }
    }
}
```

然后，需要修改 Deck.cs，以利用这个新集合，而不是数组：

```
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, 0);
        }
    }
}
```

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，而不是依赖于 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，这样集合的用户友好性就更高了。

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 接口，提供了对任何集合都相同的集合处理功能。

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 也执行Clear()和 Count(),但不执行 RemoveAt()。这是因为 RemoveAt()是 IList 接口上的一个方法，不是 IDictionary 接口上的一个方法。但是，Dictionary 有一个 Remove()方法，这是一个应在基于 DictionaryBase 的定制集合类上执行的方法。

下面的代码是上一节 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执行代码：

```
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。我们的执行代码带有一个 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());
}
```

需要使用下面的代码：

```
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，等等，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。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。只是这里为了简单起见，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;
```

这个信息就足以建立一个非常简单的示例了，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();
}
```

提示：

为了测试这些代码，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 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。

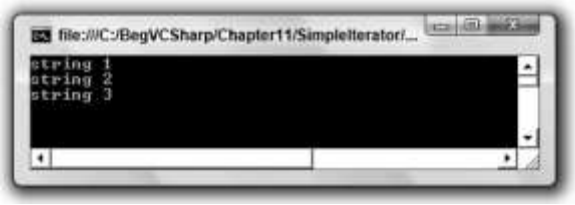

图　11-3

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 代码行，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。

试试看：实现一个迭代器

(1) 在目录 C:\BegVCSharp\Chapter11\下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;
            }
        }
    }
}
```

(3) 修改 Program.cs 中的代码，如下所示：

```
static void Main(string[] args)
{
    Primes primesFrom2To1000 = new Primes(2, 1000);
    foreach (long i in primesFrom2To1000)
        Console.Write("{0} ", i);

    Console.ReadKey();
}
```

(4) 执行应用程序，结果如图 11-4 所示。

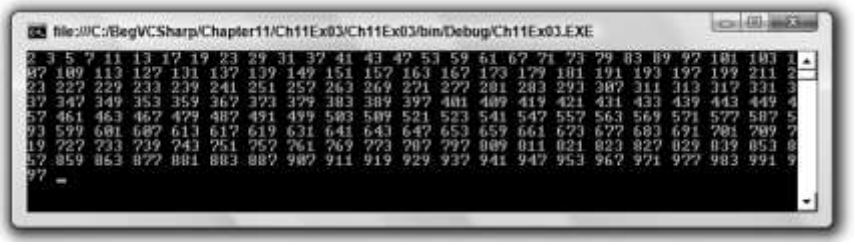

图　11-4

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。有趣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，所以先假定从这个数开始，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，获得需要的操作：

```
    public new IEnumerator GetEnumerator()
    {
      foreach (object animal in Dictionary.Values)
        yield return (Animal)animal;
    }
```

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);
      }
```

提示：

在本章的下载代码中，这些代码位于 DictionaryAnimals 项目中。

11.1.7　深度复制

第 9 章介绍了如何使用受保护的方法 System.Object.MemberwiseClone()进行浅度复制(shallow copy)。使用了一个 GetCopy()方法，如下所示：

```
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()得到的浅度复制包括一个字段，它引用的对象与源对象相同。下面的代码使用这个类来说明浅度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，需要执行深度复制。修改上面的 GetCopy()方法就可以进行深度复制，但最好使用.NET Framework 的标准方式。为此，实现ICloneable 接口，该接口有一个方法 Clone()，这个方法不带参数，返回一个对象类型，其签名和上面使用的 GetCopy()方法相同。

修改上面的类，可以使用下面的深度复制代码：

```
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(MyContent)中的 Content 对象的 Val 字段，创建一个新 Cloner 对象。这个字段是一个值类型，所以不需要深度复制。

使用与上面类似的代码测试浅度复制，但使用 Clone()而不是 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 字段也需要深度复制，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，简化了创建一个新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 添加深度复制

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，因为只需进行浅度复制(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：

```
public class Card : ICloneable
{
  public object Clone()
  {
    return MemberwiseClone();
  }
```

注意，ICloneable 接口的这段实现代码只是一个浅度复制，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。这个过程略复杂一些，因为涉及到源集合中的每个 Card 对象，所以需要进行深度复制：

```
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;
  }
```

最后，需要在 Deck 类上执行 ICloneable 接口。这里存在一个问题：因为没有洗牌，所以 Deck 类无法修改它包含的扑克牌。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，该函数可以传送指定的 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;
  }
```

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，以便于测试)：

```
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();
```

其结果如图 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。在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，事情会变得比较复杂：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对对象排序，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，常常需要知道它们的类型，这样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：封箱和拆箱。

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。要进行验证，可以把 MyStruct 改为一个类(不考虑这个类名不合适的情况)：

```
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，只要它们执行这个接口即可。例如，假定 MyStruct 类型执行 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，它允许使用集合中的值类型(如ArrayList)，集合中项的类型是 object。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。

最后要注意的是，在访问值类型的内容前，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的类型，或者是否可以转换为给定的类型，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，或者它被拆箱到该类型中，则结果为 true。

下面用几个示例说明如何使用该运算符。

试试看：使用 is 运算符

(1) 在目录 C:\BegVCSharp\Chapter11\下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：

```
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();
      }
    }
}
```

(3) 运行代码，其结果如图 11-6 所示。

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，并把它们用作类的方法的参数，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。如果类型没有继承一个类，就不会与该类兼容。

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)。

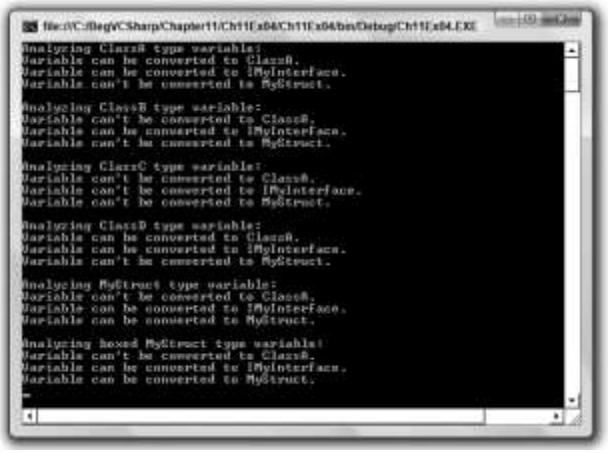

图　11-6

11.2.2　值比较

考虑两个表示人的 Person 对象，它们每个都有一个 Age 整型属性。下面要比较它们，看看哪个人年龄比较大。为此可以使用下面的代码：

```
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执行代码，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执行代码，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，详见后面的内容。

例如，考虑一个简单的类 AddClass1，如下所示：

```
public class AddClass1
{
   public int val;
}
```

这仅是 int 值的一个包装器(wrapper)，但可以用于说明规则。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，因为我们还没有定义要执行的操作。下面的代码则可执行，但得不到希望的结果：

```
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;
```

在合适时，可以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，因为它不知道要使用哪个运算符。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;
}
```

下述运算符可以重载：

一元运算符： +, - , !, ~, ++, --, true, false

二元运算符： +, - , *, /, %, &, |, ^, <<, >>

比较运算符： ==, !=, <, >, <=, >=

注意：

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+=，但这些运算符使用它们的对应运算符，例如+，所以不必担心它们。重载+表示+=按照希望的那样执行。=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(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
}
```

在比较复杂的运算符定义中，这可以减少代码，且只要修改一个实现代码，其他运算符也会修改。

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执行代码。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 运算符检查代码中的对象类型，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的类型，就会抛出一个异常。当然，这可能并不是我们希望的操作。我们要把一个类型的对象转换为另一个类型，此时需要更多的分支结构。另外，还可以限制对这些类型的比较，只允许进行相同类型的比较，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，给王牌指定花色，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
{
  // Flag for trump usage. If true, trumps are valued higher
  // than cards of other suits.
  public static bool useTrumps = false;

  // Trump suit to use if useTrumps is true.
  public static Suit trump = Suit.Club;

  // Flag that determines whether Aces are higher than Kings or lower
  // than deuces.
  public static bool isAceHigh = true;
```

这里要注意，这些规则应用于应用程序中每个 Deck 的所有 Card 对象上。遵循不同规则的每个Deck 不可能包含相同的牌。但这适用于这个类库，因为可以假定如果一个应用程序要使用不同的规则，就可以包含这些规则，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，以用不同的特性初始化扑克牌：

```
  public Deck()
  {
```

```
        for (int suitVal = 0; suitVal < 4; suitVal++)
        {
          for (int rankVal = 1; rankVal < 14; rankVal++)
          {
              cards.Add(new Card((Suit)suitVal, (Rank)rankVal));
          }
        }
      }

      // Nondefault constructor. Allows aces to be set high.
      public Deck(bool isAceHigh) : this()
      {
        Card.isAceHigh = isAceHigh;
      }

      // Nondefault constructor. Allows a trump suit to be used.
      public Deck(bool useTrumps, Suit trump) : this()
      {
        Card.useTrumps = useTrumps;
        Card.trump = trump;
      }

      // Nondefault constructor. Allows aces to be set high and a trump suit
      // to be used.
      public Deck(bool isAceHigh, bool useTrumps, Suit trump) : this()
      {
        Card.isAceHigh = isAceHigh;
        Card.useTrumps = useTrumps;
        Card.trump = trump;
      }
```

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(和推荐的重写运算符)：

```
      public Card(Suit newSuit, Rank newRank)
      {
        suit = newSuit;
        rank = newRank;
      }

      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);
        }
```

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 假定为最先出的牌。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 的级别在这个枚举中是 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);
}
```

代码的其他部分主要考虑的是 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;
}
}
```

还有一个运算符(>=)使用与此类似的代码，其他运算符是非常简单的，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);
```

其结果如图 11-7 所示。

在两种情况下，在应用运算符时都考虑了指定的规则。这在结果的最后 4 行中尤其明显，说明王牌总是胜过其他牌。

图　11-7

3. IComparable 和 IComparer 接口

IComparable 和 IComparer 接口是.NET Framework 中比较对象的标准方式。这两个接口之间的区别如下：

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。

一般情况下，我们使用 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，所以可以实现它，以便把 Person 对象传送给它，说明这个人是否比当前的人更年老或年轻。实际上，这个方法返回一个 int，所以可以使用下面的代码说明第二个人更年老还是更年轻：

```
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。

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。

试试看：给列表排序

(1) 在目录 C:\BegVCSharp\Chapter11\下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.");
         }
      }
   }
}
```

(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.");
         }
      }
   }
}
```

(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();
         }
      }
   }
```

(5) 执行代码，结果如图 11-8 所示。

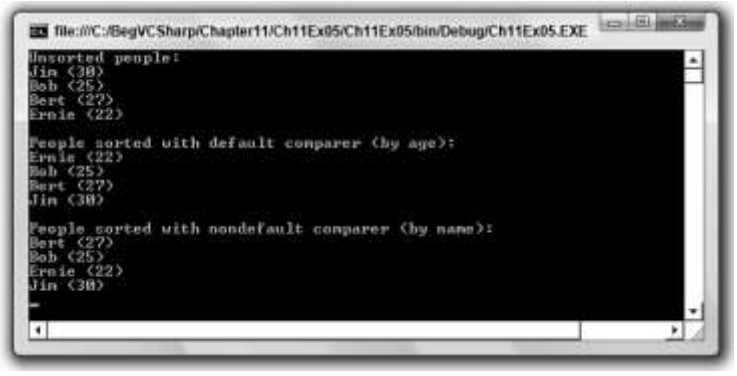

图　11-8

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。如果不行，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，很容易使用：

```
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 可以采用相同的方式隐式转换为 long 或 double，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。本节的第一部分就介绍这个内容。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，即编写下述代码：

```
ConvClass1 op1 = new ConvClass1();
ConvClass2 op2 = op1;
```

另外，还可以定义一个显式转换，在下面的代码中调用：

```
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;
```

但反方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;
```

注意，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。

注意，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个方面，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;
```

其中 obj3 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;
```

上面的 as 表达式只会把 null 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来掌握，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，还讨论了如何对比较和转换重载运算符，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大大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的姓名，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，使用上面定义的重载运算符，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 类上执行 I[illegible]能。

(5) 给 People 类添加一个[illegible]获取所有成员的年龄：

```
foreach(int age in myP
{
  //Display ages.
}
```

Windows 编程基础

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高质量的用

户界面，其编程语言的易学易用，为快速开发应用程序提供了尽可能好的环境，所以赢得了广泛的好评。Visual Basic 等快速应用程序开发(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。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 提供的控件和花钱买来的定制控件，为程序员提供了空前巨大的重用代码池，以及仅通过鼠标单击就可以完全测试的代码。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。

本章的主要内容：

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当前状态、让用户修改状态的控件，如 RadioButton 和 Check Button 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

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。

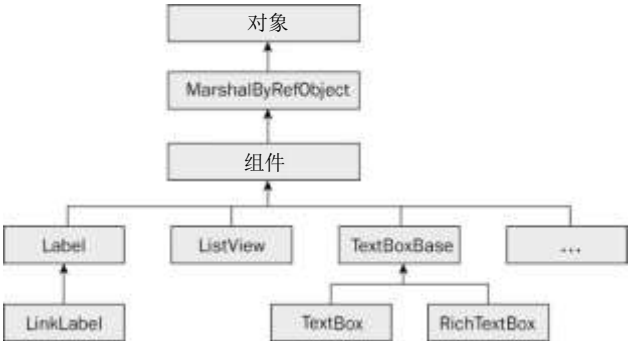

图　15-1

15.1.1　属性

所有的控件都有许多属性，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，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，如果要查看类的所有属性，请参阅.NET Framework SDK 文档说明。

表　15-1

名　称	描　述
Anchor	使用这个属性，可以指定当控件的容器大小发生变化时，该控件如何响应。参见下面对这个属性的详细解释
BackColor	控件的背景色
Bottom	设置这个属性，可以指定控件的底部距离窗口的顶部有多远。这与指定控件的高度不同
Dock	可以使控件靠在窗口的边界上。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	控件从底部到顶部的距离
Left	控件的左边界到窗口左边界的距离
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	控件的右边界到窗口左边界的距离
TabIndex	控件在容器中的标签顺序号

TabStop	指定控件是否可以用 Tab 键访问

(续表)

名　称	描　述
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 Form 设计器给这个属性赋值时，就只能给它赋一个字符串值
Text	保存与该控件相关联的文本
Top	控件的顶部距离窗口顶部的距离
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，窗体设计器默认改为使用栅格状的界面，并使用捕捉线来定位控件，使控件整齐地排列在界面上。选择 Tools 菜单上的 Options 选项，在树型视图中选择 Windows Forms Designer 节点，设置 Layout Mode，就可以在两种设计样式之间切换。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。

试试看：使用捕捉线

按照下面的步骤使用 Windows Forms 设计器中的捕捉线：

(1) 创建一个 Windows Forms 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。

(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。图 15-2 显示了在把按钮移动到窗体的左上角时 Visual Studio 显示的捕捉线。

(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。图 15-3 显示了两个按钮在移近时的情况。

(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，如图 15-4 所示，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。

图　15-2

图　15-3

图　15-4

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不容易，应确保窗口看起来不显得很乱，并编写许多代码行来达到这个目的。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 属性用于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 属性用于指定控件应停放在容器的边框上。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，如果指定控件停放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。

参见本章后面的文本框示例，了解 Anchor 属性的用法。

15.1.3　事件

第 13 章介绍了事件的概念及其用法。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，在用户单击或按下按钮时，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

名　称	描　述
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

(续表)

名　称	描　述
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
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，表示有效性验证已经完成
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，被验证有效性的控件是失去焦点的控件，而不是获得焦点的控件

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。

处理事件有 3 种基本方式。第一种是双击控件，进入控件默认事件的处理程序，这个事件对于不同的控件来说是不同的。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 15-5 所示的闪电图标按钮，就会显示 Events 列表。

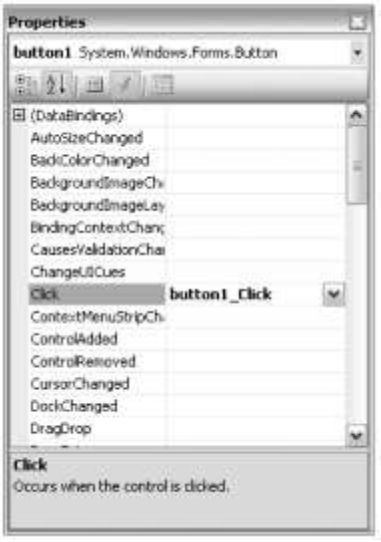

图　15-5

灰显的事件就是控件的默认控件。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。本章和下一章常常这么做，即把代码添加到窗体构造函数中的 InitializeComponent()调用之后。在输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，它实现了按钮控件所需的基本功能，所以程序员可以从这个类中派生，创建定制的按钮控件。

System.Windows.Forms 名称空间提供了 3 个派生于 ButtonBase 的控件 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。

按钮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)。

对按钮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，明白该如何操作它。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

名　称	描　述
FlatStyle	按钮的样式可以用这个属性改变。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	使用这个属性，可以设置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。

试试看：按钮的测试

按照下面的步骤创建一个小型 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\Chapter14 目录下创建一个新的 Windows 应用程序 ButtonTest。

(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 15-6 所示。

(3) 右击一个按钮，选择 Properties，在 Properties 窗口上选择(Name)编辑字段，输入相关的文本，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 ButtonTest 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，使文本不从图像的右边开头。为此，可以选择每个按钮，把文本的开头放在按钮的右边缘上。

(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 15-7 所示。

图　15-6

图　15-7

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。

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)
{
   this.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一个组合。

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 EventArgs e 包含所发生事件的信息。在本例中，不需要这些信息。

返回窗体设计器，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)
{
   this.Text = "Taler du dansk?";
}
```

这个方法与 buttonEnglish_Click 相同，但 Danish 按钮的文本有所不同。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，单击按钮，将得到如图 15-8 所示的结果。

图　15-8

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)。

在图 15-9 中，拖放了这两种类型的标签。

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。

图　15-9

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，这些属性存在于 Label 和 LinkLabel 控件中。

表　15-4

名　称	描　述
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	这个属性可以指定要在标签上显示的图像(位图，图标等)
ImageAlign	图像的对齐方式
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
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	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，也可以限制用户只输入数值。

.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。这里不讨论什么对象派生于什么类，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，所以这里仅列出最常见的属性，如表 15-5 所示。

表　15-5

名　称	描　述
CausesValidation	当控件的这个属性设置为 true，且该控件获得了焦点时，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。下面会讨论相关的事件
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: 文本框中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: 文本框中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 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
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 控件中文本的有效性验证会使一些用户很高兴，使另一些用户很生气。当用户单击了 OK 按钮后，对话框只验证其内容，此时用户会非常生气。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	描　述
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。它的一个示例是用户单击了 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 Control 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
TextChange	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。

试试看：TextBoxTest

先建立用户界面：

(1) 在 C:\BegVCSharp\Chapter15 目录下创建一个新的 Windows 应用程序 TextBoxTest。

(2) 创建如图 15-10 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 txtAddress 和 txtOutput 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　15-10

(3) 给控件命名，如图 15-10 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 TextBoxTest。

(5) 把两个控件 txtOutput 和 txtAddress 的 Scrollbars 属性设置为 Vertical。

(6) 把 txtOutput 控件的 ReadOnly 属性设置为 true。

(7) 把按钮 btnHelp 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。下面一次设置所有类型的控件的 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，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 textBox
Output 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 txtOutput 而不是让它停放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。如果把该控件停放在窗体的底部，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 MinSize 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 MinSize 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。

添加事件处理程序

在设计视图中，双击 buttonOK 按钮，对其他按钮重复这个过程。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：

```
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;
}
```

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 属性用于显示连接起来的文本。

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否是程序员。

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，设置这个属性：

```
public Form1()
{
  InitializeComponent();
  this.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。通知控件该事件应由方法 txtBoxEmpty_Validating()来处理，所以这是用于两个不同控件的一个事件处理方法。

还需要知道控件的状态。为此，使用文本框的 Tag 属性。回忆一下本章前面对这个属性的讨论，在窗体设计器中，只能给 Tag 属性赋予字符串值。但由于 Tag 属性是在代码中赋值，所以可以对它进行其他处理。Tag 属性是一个 object，所以这里输入一个布尔值更合适。

在构造函数中，添加下述语句：

```
  this.buttonOK.Enabled = false;

// Tag values for testing if the data is valid
     this.textBoxAddress.Tag = false;
     this.textBoxAge.Tag = false;
     this.textBoxName.Tag = false;
     this.textBoxOccupation.Tag = false;

// Subscriptions to events
     this.textBoxName.Validating += new System.ComponentModel.@@ta
CancelEventHandler(this.textBoxEmpty Validating);
     this.textBoxAddress.Validating += new

System.ComponentModel.CancelEventHandler(this.textBoxBoxEmpty_Validating);
```

注意在输入事件处理代码的+=部分时，VS 会检测到，我们给对象添加了事件处理程序，所以会自动输入剩余的文本。如果按一次 Tab 键，就会自动插入文本(包括新语句)和推荐的方法名。如果再次按下 Tab 键，就会插入处理事件的方法，该名称是我们指定的名称。我们常常只是按两

次 Tab 键，但这里在第二次按下 Tab 键之前，要把方法名改为 textBoxBoxEmpty_ Validating，才能用同一个方法处理两个事件。

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 的一个专业化版本。这个事件需要专业化处理程序的原因是，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)
{
   // We know the sender is a TextBox, so we cast the sender object to that
   TextBox tb = (TextBox)sender;

   // If the text is empty we set the background color of the
   // Textbox to red to indicate a problem. We use the tag value
   // of the control to indicate if the control contains valid
   // information.
   if (tb.Text.Length == 0)
   {
      tb.BackColor = Color.Red;
      tb.Tag = false;

      // In this case we do not want to cancel further processing,
      // but if we had wanted to do this, we would have added this line:
      // e.Cancel = true;
   }
   else
   {
      tb.BackColor = System.Drawing.SystemColors.Window;
      tb.Tag = true;
   }

   // Finally, we call ValidateOK which will set the value of
   // the OK button.
   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。

注意：

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。所以要在构造函数中添加一行代码：

```
this.textBoxOccupation.Validating += new
System.ComponentModel.CancelEventHandler(this.textBoxOccupation_Validating);
```

然后添加处理程序本身：

```
private void textBoxOccupation_Validating(object sender,
                                System.ComponentModel.CancelEventArgs e)
{
   // Cast the sender object to a textbox
   TextBox tb = (TextBox)sender;

   // Check if the values are correct
   if (tb.Text.CompareTo("Programmer") == 0 || tb.Text.Length == 0)
   {
      tb.Tag = true;
      tb.BackColor = System.Drawing.SystemColors.Window;
   }
   else
   {
      tb.Tag = false;
      tb.BackColor = Color.Red;
   }

   // Set the state of the OK button
   ValidateOK();
}
```

倒数第二个要做的操作是处理年龄文本框。我们希望用户只输入正数(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。

首先，把 textBoxAge 控件的 MaxLength 属性设置为 3，然后在 Properties 窗口的 Events 列表(单击闪电图标，来选择 Events 列表)中双击 KeyPress 事件，以订阅它。KeyPress 事件也是专业化的，其中提供了 System.Windows.Forms.KeyPressEventHandler，因为事件需要有关被按下键的信息。

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; // Remove the character
}
```

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 tnue，告诉控件不应对字符进行任何操作，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，看看是否输入了所有的信息。这是很简单的，因为前面已经编写了执行该检查的方法，只要在构造函数中添加如下一行代码，订阅 Age 控件的 Validating 事件：

```
this.textBoxAge.Validating += new
    System.ComponentModel.CancelEventHandler(this.textBoxEmpty_Validating);
```

最后必须对所有的文本框控件进行处理。如果用户在所有的文本框中输入了有效的文本，接着修改了某些文本，使它们不再有效，则 OK 按钮仍是可用的。所以必须为所有的文本框编写最后一个事件处理程序：TextChange 事件，如果任何文本字段包含无效的数据，它就禁用 OK 按钮。

当控件中的文本发生了改变，就激发 TextChange 事件。选择所有 4 个文本框，在 Events 列表中输入名称 textBox_TextChanged，以订阅该事件。这会为 4 个事件生成一个处理程序。

TextChange 事件使用 Click 事件的标准事件处理程序。最后，添加该事件本身：

```
private void textBox_TextChanged(object sender, System.EventArgs e)
{
   // Cast the sender object to a Textbox
   TextBox tb = (TextBox)sender;

   // Test if the data is valid and set the tag and background
   // color accordingly.
   if (tb.Text.Length == 0 && tb != textBoxOccupation)
   {
      tb.Tag = false;
      tb.BackColor = Color.Red;
   }
   else if (tb == textBoxOccupation &&
           (tb.Text.Length != 0 && tb.Text.CompareTo("Programmer") != 0))
   {
      // Don't set the color here, as it will color change while the user
      // is typing
      tb.Tag = false;
   }
   else
   {
      tb.Tag = true;
      tb.BackColor = SystemColors.Window;
   }

   // Call ValidateOK to set the OK button
```

```
        ValidateOK();
    }
```

这次，必须知道哪个控件调用了事件处理程序，因为在用户开始输入时，我们不希望把 Occupation 文本框的背景色改为红色。为此，应检查文本框的 Name 属性，该属性放在 sender 参数中。

最后一件事：激活或禁用 OK 按钮的 ValidateOK 方法：

```
    private void ValidateOK()
    {
        // Set the OK button to enabled if all the Tags are true
        this.buttonOK.Enabled = ((bool)(this.textBoxAddress.Tag) &&
                                 (bool)(this.textBoxAge.Tag) &&
                                 (bool)(this.textBoxName.Tag) &&
                                 (bool)(this.textBoxOccupation.Tag));
    }
```

如果所有的 Tag 属性都是 true，这个方法就把 OK 按钮的 Enabled 属性设置为 true。需要把 Tag 属性的值转换为布尔值，因为它存储为对象类型。

如果现在测试该程序，就会得到如图 15-11 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。

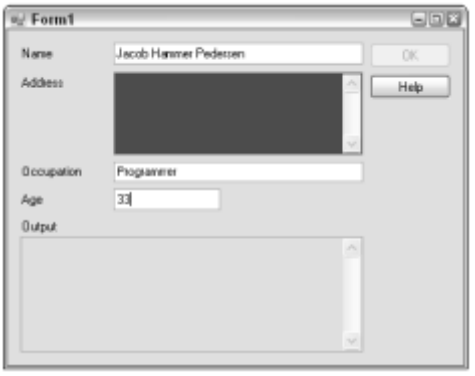

图　15-11

与本章前面的示例相比，刚完成的示例比较长，这是因为后面将以这个示例为基础，而不是从头开始。

注意：

可以从 www.wrox.com 或上 www.tupwk.com.cn/downpage 上下载本书中所有示例的源代码。

15.5　RadioButton 和 CheckBox 控件

如前所述，RadioButton 和 CheckBox 控件与按钮控件有相同的基类，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。在要给用户提供几个互斥选项时，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，以反映组框中惟一被选中的选项。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，左边是一个带有标记的小方框。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、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，所以要描述的属性只有几个，如表 15-7 所示。完整的列表请参阅.NET Framework SDK 文档说明。

表 15-7

名 称	描 述
Appearance	RadioButton 可以显示为一个圆形选中标签，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检查单选按钮
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 RadioButtons 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

名 称	描 述
CheckChanged	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 CheckChanged 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，且只改变以前未选中的控件的 Checked 属性。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

名　称	描　述
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，但其结果是不同的，如表 15-10 所示。

表　15-10

名　称	描　述
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 控件常常用于逻辑地组合一组控件，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。

试试看：RadioButton 和 CheckBox 示例

下面修改用于说明文本框用法的 TextBoxTest 示例。在该示例中，惟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 15-12 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　15-12

(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)。CheckBox 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) 在窗体的构造函数中，删除引用 textBoxOccupation 的两行代码，这包括对 Validating 事件的订阅，以及把 Tag 属性设置为 false 的代码行。

(2) 彻底删除 textOccupation_Validating()方法。

示例的说明

textBox_TextChanged 方法可以用于测试调用的控件是否为 textBoxOccupation 文本框。我们现在知道该控件并不是 textBoxOccupation 文本框(因为已删除了它),所以要修改该方法，即删除 else if 块，并修改 if 测试代码，如下所示：

```
private void textBox_TextChanged(object sender, System.EventArgs e)
{
   // Cast the sender object to a Textbox
   TextBox tb = (TextBox)sender;

   // Test if the data is valid and set the tag's background
   // color accordingly.
   if (tb.Text.Length == 0)
   {
      tb.Tag = false;
      tb.BackColor = Color.Red;
   }
   else
   {
      tb.Tag = true;
      tb.BackColor = SystemColors.Window;
   }

   // Call ValidateOK to set the OK button
   ValidateOK();
}
```

在 ValidateOK()方法中检查已删除的文本框的值，彻底删除检查代码，则最终代码变成：

```
private void ValidateOK()
{
   // Set the OK button to enabled if all the Tags are true
   this.buttonOK.Enabled = ((bool)(this.textBoxAddress.Tag) &&
                            (bool)(this.textBoxAge.Tag) &&
                           (bool)(this.textBoxName.Tag));
}
```

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。我们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;
    }
```

下面只剩下修改帮助文本了，所以不要对帮助方法感到惊讶。在 OK 方法中，它显得稍微有趣一点：

```
    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，可以不选中任何一个单选按钮，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其中一个单选按钮

现在运行示例，得到如图 15-13 所示的结果。

图　15-13

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短文本字符串，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 的一些常用属性。

表　15-11

名　称	描　述
CanRedo	如果上一个被撤销的操作可以使用 Redo 重复，这个属性就是 true
CanUndo	如果可以在 RichTextBox 上撤销上一个操作，这个属性就是 true，注意，CanUndo 在 TextBoxBase 中定义，所以也可以用于 TextBox 控件
RedoActionName	这个属性包含通过 Redo 方法执行的操作名称

DetectUrls	把这个属性设置为 true，可以使控件检测 URL，并格式化它们(在浏览器中是带有下划线的部分)
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。如果把这些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 的左边就会出现一个页边距，这将使用户更易于选择文本
UndoActionName	如果用户选择撤销某个动作，该属性将获取该动作的名称
SelectionProtected	把这个属性设置为 true，可以指定不修改文本的某些部分

从上面的列表可以看出，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 中有几个有趣的新事件。

表　15-12

名　称	描　述
LinkedClick	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

(续表)

名　称	描　述
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，就可以在该事件中禁止修改

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，这个示例被加载和保存到固定的文件中。

试试看：RichTextBox 示例

与往常一样，首先设计窗体：

(1) 在 C:\BegVCSharp\Chapter15 目录下创建一个新的 C# Windows 应用程序，命名为 RichTextBoxTest。

(2) 创建窗体，如图 15-14 所示。文本框 textSize 应是一个 TextBox 控件。richTextBoxText 文本框应是一个 RichTextBox 控件。

图　15-14

(3) 如图 15-14 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，下面添加代码。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：

```
private void buttonBold_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 bold style now, we should remove the
    // Formatting
    if (oldFont.Bold)
      newFont = new Font(oldFont, oldFont.Style & ~FontStyle.Bold);
    else
      newFont = new Font(oldFont, oldFont.Style | FontStyle.Bold);

    // Insert the new font and return focus to the RichTextBox
    this.richTextBoxText.SelectionFont = newFont;
    this.richTextBoxText.Focus();
}
```

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。然后检查一下选中文本是否为黑体。如果是，就去除黑体设置；否则就设置黑体。使用 oldFont 作为原型，创建一个新字体，但根据需要添加或删除黑体格式。

最后，把新字体赋给选中的文本，把焦点返回给 RichTextBox。Font 对象详见第 33 章。

buttonItalic 和 buttonUnderline 的事件处理程序的代码与上面的代码相同，但检查相关样式的代码不同。双击 Italic 和 Underline 两个按钮，添加下面的代码：

```
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，添加下面的代码：

```
    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();
    }
```

这里必须检查一下另一个属性 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开关那样运作。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 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)
    {
        // Remove all characters that are not numbers, backspace and enter.
        if ((e.KeyChar < 48 || e.KeyChar > 57) &&
                                        e.KeyChar != 8 && e.KeyChar != 13)
        {
            e.Handled = true;
        }
        else if (e.KeyChar == 13)
        {
            // Apply size if the user hits enter
            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)
  {
    TextBox txt = (TextBox)sender;

    ApplyTextSize(txt.Text);
    this.richTextBoxText.Focus();
  }

  private void ApplyTextSize(string textSize)
  {
    // Convert the text to a float because we'll be needing a float shortly
    float newSize = Convert.ToSingle(textSize);
    FontFamily currentFontFamily;
    Font newFont;

    // Create a new font of the same family but with the new size
    currentFontFamily = this.richTextBoxText.SelectionFont.FontFamily;
    newFont = new Font(currentFontFamily, newSize);

    // Set the font of the selected text to the new font
    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，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数据类型。

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，还可以居中文本。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的上面时，鼠标指针就会变成手的形状。单击该文本，就会打开一个网页。我们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，给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。它用于提供单击链接的文本，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)
{
   // Load the file into the RichTextBox
   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。保存文件也是这样，双击 Save 按钮，添加下面的代码：

```
private void buttonSave_Click(object sender, EventArgs e)
{
   // Save the text
   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。

这是一个 RichTextBox 示例。运行它时，应得到如图 15-15 所示的结果。

图　15-15

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。(例如一起工作的工人列表框)。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。

另一种类型的列表框称为 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

除非显式声明，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	描 述
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的第一个选项
ColumnWidth	在包含多个列的列表框中，这个属性指定列的宽度
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或设置列表框中列的个数
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照字母顺序排序
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
CheckedIndicies	(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
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	描　述
ClearSelected()	清除列表框中的所有选项
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
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

正常情况下，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	描　述
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。

试试看：使用 ListBox 控件

创建如下所示的对话框：

(1) 在 C:\BegVCSharp\Chapter15 目录下创建一个新的 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 15-16 所示。

(3) 把按钮的 Text 属性改为 Move。

(4) 把 CheckedListBox 的属性 CheckOnClick 改为 true。

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 CheckedItems 中的所有选中标记。

现在需要移动 CheckedListBox 中的内容。可以在设计模式下添加选项，方法是选择 Perperties 窗口上的 Items 属性，在其中添加选项，如图 15-17 所示。

图　15-16

图　15-17

另外，还可以在代码中添加选项。例如在窗体的构造函数中添加下面的代码：

```
public Form1()
{
   //
   // Required for Windows Form Designer support.
   //
   InitializeComponent();
```

```
        // Add a tenth element to the CheckedListBox.
        this.checkedListBoxPossibleValue.Items.Add("Ten");
    }
```

这里给 checkedListBox 添加了第 10 个选项，因为已经在设计器中输入了 9 个选项。

这是一个列表框示例，如果现在运行它，将得到如图 15-18 所示的结果。选择 Two、Four 和 Six，单击 Move 按钮，就得到了这个结果。

图　15-18

添加事件处理程序

现在准备添加一些代码。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。双击该按钮，输入下面的代码：

```
    private void buttonMove_Click(object sender, EventArgs e)
    {
        // Check if there are any checked items in the CheckedListBox
        if (this.checkedListBoxPossibleValues.CheckedItems.Count > 0)
        {
            // Clear the ListBox we'll move the selections to
            this.listBoxSelected.Items.Clear();

            // Loop through the CheckedItems collection of the CheckedListBox
            // and add the items in the Selected ListBox
            foreach (string item in this. checkedListBoxPossibleValues.CheckedItems)
            {
                this.listBoxSelected.Items.Add(item.ToString());
            }

            // Clear all the checks in the CheckedListBox
            for (int i = 0; i < this. checkedListBoxPossibleValues.Items.Count; i++)
                this. checkedListBoxPossibleValues.SetItemChecked(i, false);
        }
    }
```

15.8　ListView 控件

图 15-19 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方式，例如，显示大图标、详细视图等。

图　15-19

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 15-19 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，并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机器选择的值 ● 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的选项对齐的方式。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
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

(续表)

属　性	说　明
LabelWrap	如果这个属性是 true 时，标签就会自动换行，以显示所有的文本
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
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 种不同的模式显示其选项： • LargeIcon: 所有的选项都在其旁边显示一个大图标(32x32)和一个标签 • SmallIcon: 所有的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	描　述
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

(续表)

方　法	描　述
EndUpdate()	在调用 BeginUpdate 之后调用这个方法。在调用这个方法时，列表视图会显示出其所有的选项
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 的选项

15.8.3　ListView 控件的事件

表 15-19 列出了要处理的 ListView 控件事件。

表　15-19

事　件	描　述

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 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个列。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它本身的控件。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，其中包含所有的组件。这个功能可以防止不是用户界面一部分的控件妨碍窗体设计器。这个控件的处理方式与其他控件相同，但不能移动它。

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。为了简单起见，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信息图标。

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。

试试看：使用 ListView 控件

与往常一样，首先创建用户界面：

(1) 在 C:\BegVCSharp\Chapter15 目录下创建一个新 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 15-20 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 15-20 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　15-20

(4) 把单选按钮和按钮的 Text 属性值改为其名称，但没有控件名称，把窗体的 Text 属性设置为 ListView。

(5) 清除标签的 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 选项卡上。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 imageListLage 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的顶部。

(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) 设置控件 listViewFilesAndFolder 的属性，如表 15-20 所示。

表　15-20

属　　性	值
LargeImageList	imageListLarge
SmallImageList	imageListSmall
View	Details

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很慢，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(); // Folders
FileInfo[] files = dir.GetFiles();            // Files
```

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()方法也返回一个对象集合，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 字符串集合：

```
InitializeComponent();
// Init ListView and folder collection
folderCol = new System.Collections.Specialized.StringCollection();
CreateHeadersAndFillListView();
PaintListView(@"C:\");
folderCol.Add(@"C:\");
```

为了允许用户双击 ListView 中的选项，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)
{
   // Cast the sender to a ListView and get the tag of the first selected
   // item.
   System.Windows.Forms.ListView lw = (System.Windows.Forms.ListView)sender;
   string filename = lw.SelectedItems[0].Tag.ToString();

   if (lw.SelectedItems[0].ImageIndex != 0)
   {
      try
      {
         // Attempt to run the file.
         System.Diagnostics.Process.Start(filename);
      }
      catch
      {
         // If the attempt fails we simply exit the method.
         return;
      }
   }
   else
   {
      // Insert the items
      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，给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：

```
    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.List;
    }

    private void radioButtonSmallIcon_CheckedChanged(object sender, EventArgs e)
    {
        RadioButton rdb = (RadioButton)sender;
        if (rdb.Checked)
            this.listViewFilesAndFolders.View = View.SmallIcon;
    }

    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;
    }
```

检查单选按钮，看看它是否改为 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 15-21 所示的结果。

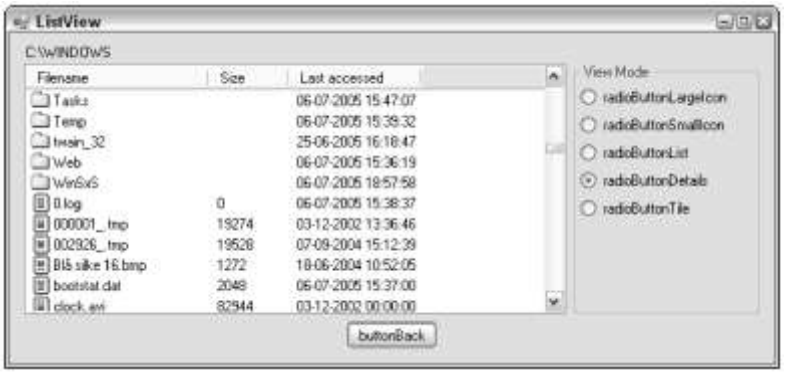

图　15-21

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
{
   // Member field to hold previous folders
   private System.Collections.Specialized.StringCollection folderCol;
```

在窗体设计器中没有创建任何列标题，现在要在代码中创建。我们使用方法 CreateHeadersAndFillListView()来创建它们：

```
private void CreateHeadersAndFillListView()
{
   ColumnHeader colHead;

   // First header
   colHead = new ColumnHeader();
   colHead.Text = "Filename";
   this.listViewFilesAndFolders.Columns.Add(colHead); // Insert the header

   // Second header
   colHead = new ColumnHeader();
   colHead.Text = "Size";
   this.listViewFilesAndFolders.Columns.Add(colHead); // Insert the header

   // Third header
   colHead = new ColumnHeader();
   colHead.Text = "Last accessed";
   this.listViewFilesAndFolders.Columns.Add(colHead); // Insert the header
}
```

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
   {
      // Two local variables that is used to create the items to insert
      ListViewItem lvi;
      ListViewItem.ListViewSubItem lvsi;

      // If there's no root folder, we can't insert anything
      if (root.CompareTo("") == 0)
         return;

      // Get information about the root folder.
      DirectoryInfo dir = new DirectoryInfo(root);

      // Retrieve the files and folders from the root folder.
      DirectoryInfo[] dirs = dir.GetDirectories(); // Folders
      FileInfo[] files = dir.GetFiles();            // Files

      // Clear the ListView. Note that we call the Clear method on the
      // Items collection rather than on the ListView itself.
      // The Clear method of the ListView remove everything, including column
      // headers, and we only want to remove the items from the view.
      this.listViewFilesAndFolders.Items.Clear();

      // Set the label with the current path
      this.labelCurrentPath.Text = root;

      // Lock the ListView for updates
      this. listViewFilesAndFolders.BeginUpdate();

      // Loop through all folders in the root folder and insert them
      foreach (System.IO.DirectoryInfo di in dirs)
      {
         // Create the main ListViewItem
         lvi = new ListViewItem();
         lvi.Text = di.Name; // Folder name
         lvi.ImageIndex = 0; // The folder icon has index 0
         lvi.Tag = di.FullName; // Set the tag to the qualified path of the
                        // folder

         // Create the two ListViewSubItems.
         lvsi = new ListViewItem.ListViewSubItem();
         lvsi.Text = ""; // Size - a folder has no size and so this column
                     // is empty
```

```
            lvi.SubItems.Add(lvsi); // Add the sub item to the ListViewItem.

            lvsi = new ListViewItem.ListViewSubItem();
            lvsi.Text = di.LastAccessTime.ToString(); // Last accessed column
            lvi.SubItems.Add(lvsi); // Add the sub item to the ListViewItem.

            // Add the ListViewItem to the Items collection of the ListView
            this. listViewFilesAndFolders.Items.Add(lvi);
          }

          // Loop through all the files in the root folder
          foreach (FileInfo fi in files)
          {
            // Create the main ListViewItem
            lvi = new ListViewItem();
            lvi.Text = fi.Name; // Filename
            lvi.ImageIndex = 1; // The icon we use to represent a folder has
            // index 1
            lvi.Tag = fi.FullName; // Set the tag to the qualified path of the
            // file.

            // Create the two sub items
            lvsi = new ListViewItem.ListViewSubItem();
            lvsi.Text = fi.Length.ToString(); // Length of the file
            lvi.SubItems.Add(lvsi); // Add to the SubItems collection

            lvsi = new ListViewItem.ListViewSubItem();
            lvsi.Text = fi.LastAccessTime.ToString(); // Last Accessed Column
            lvi.SubItems.Add(lvsi); // Add to the SubItems collection

            // Add the item to the Items collection of the ListView
            this. listViewFilesAndFolders.Items.Add(lvi);
          }

          // Unlock the ListView. The items that have been inserted will now
          // be displayed
          this. listViewFilesAndFolders.EndUpdate();
        }
        Catch (System.Exception err)
        {
          MessageBox.Show("Error: " + err.Message);
        }
      }
```

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富有逻辑的部分，以便根据控

件顶部的标签来访问。TabControl 包含 TabPages，它的工作方式与 GroupBox 控件非常类似，也是把控件组合在一起，但它比较复杂。

图 15-22 是 Word 2003 中的 Tools | Options 对话框。注意对话框顶部的 3 行标签。单击其中的每一个标签都会在对话框的剩余空间中显示不同的控件集合。它清楚地说明了如何使用 TabControl 控件来组合相关信息，使用户易于查找需要的信息。

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标签，再把要显示的控件拖放到各个页面上。

图　15-22

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 容器的外观，特别是正在显示的选项卡。

表　15-21

属　性	描　述
Alignment	控制标签在标签控件的什么位置显示。默认的位置为控件的顶部
Appearance	控制标签的显示方式。标签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标签时，其外观就会改变
Multiline	如果这个属性设置为 true，就可以有几行标签
RowCount	返回当前显示的标签行数
SelectedIndex	返回或设置选中标签的索引
SelectedTab	返回或设置选中的标签。注意这个属性在 TabPages 的实例上使用
TabCount	返回标签的总数

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页面的标签页的容器。在工具箱中双击 TabControl 时，就会显示一个已添加了两个 TabPage 的控件，如图 15-23 所示。

把鼠标移动到该控件的上面，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，用于访问选中控件的属性和方法。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。TabControl 的 Actions 窗口可以方便地在设计期间添加和删除 TabPages。

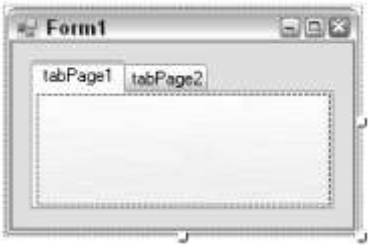

图　15-23

上面给 TabControl 添加标签页的过程可以让用户很快使用和运行该控件。另一方面，如果要改变标签的操作方式或样式，就应使用 TabPages 对话框，在选择 Properties 面板上的 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。

试试看：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标签控件的不同页面上：

(1) 在 C:\BegVCSharp\Chapter15 目录下创建一个新的 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，打开如图 15-24 所示的对话框。

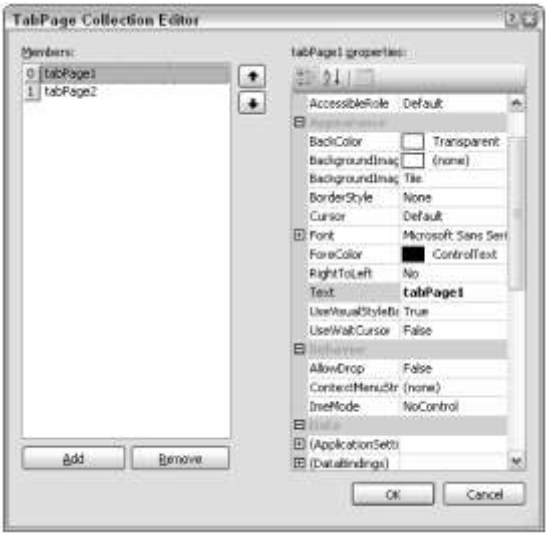

图　15-24

(4) 把标签页的 Text 属性分别改为 Tab One 和 Tab Two。单击 OK，关闭该对话框。

(5) 单击控件顶部的标签，选择要处理的选项卡。选择标有 Tab One 的选项卡。在控件上拖放一个按钮。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，而不是标签控件上。

(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 的标签，把一个文本框控件拖放到 TabControl 上。把这个控件命名为 textBoxMessage，并清除 Text 属性。

(8) 这两个标签应如图 15-25 和 15-26 所示。

图　15-25

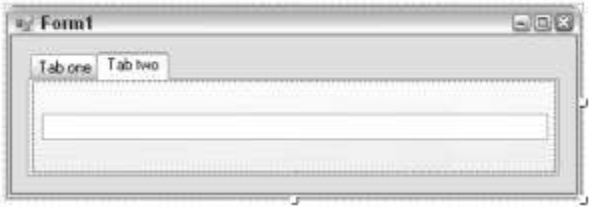

图　15-26

示例的说明

在标签页上访问一个控件，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。

在本章的前面，我们介绍了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。

要能处理标签控件，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，如果没有选择标签，就返回 null。而 SelectedIndex 返回标签的索引，如果没有选择标签，就返回 - 1。练习题(2)将使用这些属性。

添加事件处理程序

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。为了说明标签控件的用法，剩下要做的工作是添加一些代码，在用户单击一个选项卡上的 Show Message 按钮时，在另一个标签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)
{
   // Access the TextBox.

   MessageBox.Show(this.textBoxMessage.Text);
}
```

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，给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。

本章讨论的控件包括：

使用 Label 和 LinkedLabel 用户显示信息

使用 Button 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

使用 TextBox 和 RichTextBox 控件，让用户输入纯文本或格式化文本

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

使用 CheckedListView 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 CheckedListView 控件类似的列表，但没有复选框。

使用 ListBox 和 ImageList 控件提供一个列表，让用户以不同的方式查看。

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 章将讨论更复杂的控件，以及创建 Windows 窗体应用程序的方式。

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，支持在 Windows 窗体项目中使用各种样式。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，使其在 TabPage 中显示文本 You changed the current tab to <Text of the current tab>. It was just left>。

(3) 在 ListView 示例中，使其在 Tag 属性中保存文件夹和文件的绝对路径。修改这个操作，创建一个新类，使用这个新类的实例作为 ListView 中的项。在新类中保存文件和文件夹的信息。

第26章

LINQ 简 介

本章介绍 Language Integrated Query(LINQ)，这是 C# 3.0 语言中新增的一个扩展，C# 3.0 是 Visual C# 2008 中支持的 C#语言。LINQ 可以处理非常大的对象集合，这一般需要选择集合的一个子集，来完成执行程序的任务。

在过去，完成这类任务需要编写大量的循环代码，而额外的处理甚至需要更多的代码，例如，排序或组合所找到的对象。而 LINQ 使我们无需编写这些循环代码，就可以进行过滤和排序操作。这样我们就可以集中关注程序中的重要对象，提供查询。

除了提供一种简洁的查询语言，指定要搜索的对象之外，LINQ 还提供了许多扩展方法，更便于排序、组合和计算查询结果的统计数据。

LINQ 也允许查询大型数据库或复杂的 XML 文档，在大型数据库或复杂的 XML 文档中，需要高效地搜索或操作数百万甚至数十亿个对象。传统上，这个问题通常用一个特殊的类库来解决，甚至使用另一种语言来解决，例如，数据库查询语言 SQL。但是，给许多不同类型的对象扩展类库并不容易，而混合语言会导致类型不匹配的问题，对于两种语言都不熟悉的开发人员来说很难理解程序。LINQ 提供了一种内置于 C#语言的机制，解决了这些问题。

本章介绍了 LINQ 的各种特性，包括 LINQ to Objects、LINQ to SQL 和 LINQ to XML，还讨论了如下主题：

编写 LINQ 查询，LINQ 查询语言的组成部分

LINQ 方法语法和 LINQ 查询语法

使用λ表达式和 LINQ
对查询结果排序，包括多级排序
使用 LINQ 合计运算符的方式和场合
使用投射(Projection)在查询中创建新对象
使用 Distinct()、Any()、All()、First()、FirstOrDefault()、Take()和 Skip()运算符
组合查询
Set 运算符和联合

提示：

本章主要讨论 LINQ to Objects，但这些概念能应用于 LINQ 的所有变体，是后续章节 LINQ to SQL 和 LINQ to XML 的必备基础知识。

LINQ 非常大，完整地介绍它的所有特性和方法已超出了本书的讨论范围。但在本章中，我们将列举 LINQ 用户需要的所有运算符和语句的示例，并给出深入探讨这些主题的资料。

26.1 LINQ 的变体

Visual Studio 2008 带有 3 个内置的 LINQ 变体：LINQ to Objects、LINQ to SQL 和 LINQ to XML，它们为不同类型的数据提供了查询解决方案：

LINQ to Objects：为任意类型的 C#内存对象提供查询，例如数组、列表和其他集合类型。本章的所有示例都使用 LINQ to Objects。也可以把本章的技巧应用于 LINQ 的所有变体。

LINQ to SQL：为使用标准 SQL 数据库查询语言的关系数据库提供查询，例如，Micorsoft SQL Server、Oracle 和其他数据库。过去，C#访问这些数据库时，开发人员至少需要学习一些有关 SQL 的知识，但现在查询功能通过 LINQ 内置于 C#中，可以让 LINQ to SQL 来处理 SQL 转换。有关 LINQ to SQL 的内容详见第 27 章。

LINQ to XML：提供了 XML 文档的创建和处理功能，其语法与一般查询机制和其他 LINQ 变体相同。有关 LINQ to XML 的内容详见第 27 和 29 章。

LINQ Providers：除了对象、SQL 数据库和 XML 之外，还可以为其他数据源扩展 LINQ。只需为该类数据源编写一个 LINQ 提供程序即可。LINQ 提供程序的设计是一个高级主题，超出了本书的讨论范围，但所有的 C#开发人员都必须知道，可以编写新的 LINQ 提供程序(由 Micorsoft 和第三方编写)，所以以后一定会出现其他 LINQ 变体。

26.2 第一个 LINQ 查询

下面先介绍一个示例。在这个示例中，创建了一个查询，使用 LINQ 在一个简单的内存对象数组中查找一些数据，并输出到控制台上。

试试看：第一个 LINQ 查询

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-1-FirstLINQquery，打开主源文件 Program.cs。

(2) 注意，Visual C# 2008 默认在 Program.cs 中包含 LINQ 名称空间：

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

(3) 在 Program.cs 的 Main()方法中添加如下代码：

```
static void Main(string[] args)
{
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
"Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

var queryResults =
from n in namesn
where n.StartsWith("S")
select n;

Console.WriteLine("Names beginning with S: ");

foreach (var item in queryResults) {
Console.WriteLine(item);
}

Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
}
```

(4) 编译并运行程序(按下 F5 键即可开始调试)，列表中的名称以 S 开头，按照它们在数组中的声明顺序排列，如下所示。

```
Names beginning with S:
Smith
Smythe
Small
Singh
Samba
Program finished, press Enter/Return to continue:
```

按下回车键，结束程序，关闭控制台屏幕。如果使用 Ctrl+F5 组合键(启动时不使用调试功能)，就需要按下回车键两次，这会结束程序的运行。

示例的说明

第一步是引用 System.Linq 名称空间，这在创建项目时由 Visual C# 2008 自动完成：

```
using System.Linq;
```

所有的基本底层系统都支持 System.Linq 名称空间中用于 LINQ 的类。如果在 Visual C# 2008 外部创建 C#源文件或编辑以前的 Visual C# 2005 项目，就必须手动添加 using System.Linq 语句。

下一步是创建一些数据，在本例中就是声明并初始化 names 数组：

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe","Small",
 "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
```

这些数据很少，很适用于查询结果比较明显的示例。程序的下一部分是 LINQ 查询语句：

```
var queryResults =
    from n in names
     where n.StartsWith("S")
     select n;
```

这是一个看起来比较古怪的语句。它不像是 C#语言，实际上 from…where…select 类似于 SQL 数据库查询语言。但是，这个语句不是 SQL，而是 C#，在 Visual Studio 2008 中输入这些代码时，from、where 和 select 会突出显示为关键字，这个古怪的语法对编译器而言是完全正确的。

这个程序中的 LINQ 查询语句使用了 LINQ 声明性查询语法：

```
var queryResults =
    from n in names
    where n.StartsWith("S")
    select n;
```

该语句包括 4 个部分：以 var 开头的结果变量声明，使用查询表达式给该结果变量赋值，查询表达式包含 from 子句、where 子句和 select 子句。下面逐一介绍它们。

26.2.1 用 var 关键字声明结果变量

LINQ 查询首先声明一个变量，以包含查询的结果，这通常是用 var 关键字声明一个变量来完成的:

```
var queryResults =
```

如第 14 章所述，var 是 C# 3.0 中的一个新关键字，用于声明一般的变量类型，特别适合于包含 LINQ 查询的结果。var 关键字告诉 C#编译器，根据查询推断结果的类型。这样，就不必提前声明从 LINQ 查询返回的对象类型了——编译器会推断出该类型。如果查询返回多个结果，该变量就是查询数据源中的一个对象集合(在技术上它并不是一个集合，只是看起来像是集合而已)。

提示：

如果希望了解细节，查询结果将是实现了IEnumerable<>接口的类型。在本例中，编译器创建了System.Linq.OrderedSequence<string, string>的一个实例，这个特殊的 LINQ 数据类型提供了字符串的有序列表(因为数据源是一个字符串集合)。

另外，queryResult 名称是随意指定的，可以把结果命名为任何名称，例如，namesBeginning

WithS 或者其他在程序中有意义的名称。

26.2.2　指定数据源：from 子句

LINQ 查询的下一部分是 from 子句，它指定了要查询的数据：

```
from n in names
```

本例中的数据源是前面声明的字符串数组names。变量n 只是数据源中某一元素的代表，类似于 foreach 语句后面的变量名。指定 from 子句，就可以只查找集合的一个子集，而不用迭代所有的元素。

说到迭代，LINQ 数据源必须是可枚举的——必须是数组或集合，可以从中选择出一个或多个元素。

提示：

在技术上，这表示数据源必须支持 IEnumerable<>接口，所有的 C#数组或集合都支持这个接口。

数据源不能是单个值或对象，例如，单个 int 变量。如果只有一项，就没有必要查询了。

26.2.3　指定条件：where 子句

在 LINQ 查询的下一部分，可以用 where 子句指定查询的条件，如下所示：

```
where n.StartsWith("S")
```

可以在 where 子句中指定能应用于数据源中各元素的任意布尔(true 或 false)表达式。实际上，where 子句是可选的，甚至可以忽略，但在大多数情况下，都要指定 where 条件，把结果限制为我们需要的数据。where 子句称为 LINQ 中的限制运算符，因为它限制了查询的结果。

这个示例指定 name 字符串以字母S 开头，还可以给字符串指定其他条件，例如，长度超过 10(where n.Lenngth > 10)或者包含 Q(where n.Contains("Q"))。

26.2.4　指定元素：select 子句

最后，select 子句指定结果集中包含哪些元素。select 子句如下所示：

```
select n;
```

select 子句是必须的，因为必须指定结果集中有哪些元素。这个结果集并不是很有趣，因为在结果集的每个元素中都只有一项 name。如果结果集中有比较复杂的对象，使用 select 子句的有效性就比较明显，不过我们还是先完成这个示例。

26.2.5　完成：使用 foreach 循环

现在输出查询的结果。与把数组用作数据源一样，像这样的 LINQ 查询结果是可以枚举的，即可以用 foreach 语句迭代结果：

```
Console.WriteLine("Names beginning with S:");

foreach (var item in queryResults) {
        Console.WriteLine(item);
}
```

在本例中，匹配了 4 个名称：Singh、Small、Smythe 和 Samba，所以它们会显示在 foreach 循环中。

26.2.6　延迟执行的查询

foreach 循环并不是 LINQ 的一部分，它只是迭代结果。虽然 foreach 结构并不是 LINQ 的一部分，但它是实际执行 LINQ 查询的代码。查询结果变量仅保存了执行查询的一个计划，在访问查询结果之前，并没有提取 LINQ 数据，这称为查询的延迟执行。生成结果序列(即列表)的查询都要延迟执行。

现在回过头来看看代码。由于输出了结果，所以程序结束：

```
Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
```

这些代码仅确保按下一个键(甚至可以按下 F5 键，而不是 Ctrl+F5 组合键)后，控制台程序的结果就显示在屏幕上。在大多数其他 LINQ 示例中也使用这种结构。

26.3　使用 LINQ 方法语法和λ表达式

用 LINQ 完成同一任务有多种方式，但常常需要通过编程来实现。如前所述，前面的示例是用 LINQ 查询语法编写的，下一个示例是用 LINQ 的方法语法(也称为显式语法，但这里使用“方法语法”这个术语)编写的相同程序。

26.3.1　LINQ 扩展方法

LINQ 实现为一系列扩展方法，用于集合、数组、查询结果和其他执行了IEnumerable 接口的对象。在 Visual Studio IntelliSense 特性中可以看到这些方法。例如，在 Visual Studio 2008 中打开 FirstLINQquery 程序中的 Program.cs 文件，在 name 数组的下面输入对该数组的一个新引用：

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe", "Small",↵
"Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };
```

```
names.
```

输入 names 后面的句点后，就会看到 Visual Studio 2008 IntelliSense 列出的可用于 names 的方法，如图 26-1 所示。

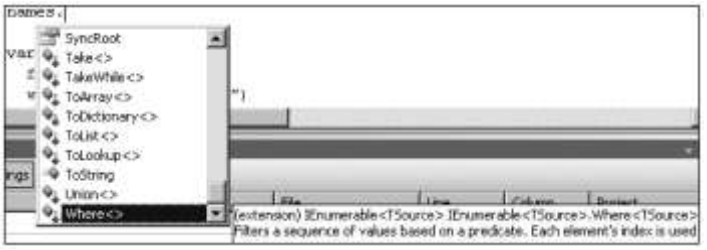

图　26-1

Where<>方法和大多数其他方法都是扩展方法(在 Where<>方法的右边显示了一个文档说明，它以 extension 开头)。这些方法在顶部注释掉了指令 using System.Linq，表示它们是 LINQ 扩展，但列表中 Where<>、Union<>、Take<>和大多数其他方法都没有这个指令。上一个示例使用的 from…where…select 查询表达式由 C#编译器转换为这些方法的一系列调用。使用 LINQ 方法语法时，就直接调用这些方法。

26.3.2　查询语法和方法语法

查询语法是在 LINQ 中编写查询的首选方式，因为它一般更容易理解，最常见的查询使用它们也更简单。但是，一定要基本了解方法语法，因为一些 LINQ 功能不能通过查询语法来使用，或者使用方法语法比较简单。

注意：

Visual C# 2008 在线帮助推荐，尽可能使用查询语法，仅在需要时使用方法语法。

本章主要使用查询语法，但会指出需要方法语法的场合，并说明如何使用方法语法来解决问题。

26.3.3　λ表达式

大多数使用方法语法的 LINQ 方法都要求传送一个方法或函数，来计算查询表达式。方法/函数参数以委托的形式传送，它一般引用一个匿名方法。

LINQ 很容易完成这个传送任务。使用特殊的 C# 3.0 结构(称为λ表达式)就可以创建方法/函数。λ表达式是编写匿名方法的简洁方式，用于执行查询表达式。下面是一个简单的λ表达式：

```
n = > n < 1000
```

运算符=>称为λ运算符。

提示：

“λ”来自于λ微积分学，这是底层程序编程语言的数学形式，是 LINQ 的基础编程方式。解释它的内容超出了本书的讨论范围，但若读者感兴趣，可以查看λ微积分学的 Wikipedia 文章。使用λ函数不需要理解这些数学知识，但理解编程有助于学习高级 LINQ 编程。请查看本章最后的“资源和进一步阅读”一节。

λ表达式是定义函数的缩写方式。对于示例 n=>n<1000，λ表达式定义了一个函数，其参数名是 n，如果 n 小于 1000，该函数就返回 true；如果 n 大于 1000，函数就返回 false。函数是没有名称的匿名方法，仅在把λ表达式传送给底层的 LINQ 方法时使用。

Visual C# 2008 在线帮助建议，这个λ表达式应读作“n goes to n 大于 1000”，但是对于像这样的 true/false λ表达式，把表达式读作“n such that n 小于 1000”比较好。

第一个示例中查询的λ表达式可以写作：

```
n = > n.StartsWith("S"));
```

读作“n such that n 以 S 开头”。

下一个示例将更清楚地说明这一点。

试试看：使用 LINQ 方法语法和λ表达式

按照下面的步骤在 VC# 2008 中创建示例：

(1) 可以修改 FirstLINQQuery 示例，或者在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-2 LINQMethodSyntax。打开主源文件 Program.cs。

(2) Visual C# 2008 会自动在 Program.cs 中包含 LINQ 名称空间：

```
using System.Linq;
```

(3) 在 Program.cs 的 Main()方法中添加如下代码，其中突出显示了与第一个示例不同的代码：

```
static void Main(string[] args)
{
          string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
"Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah"
};
                    var queryResults = names.Where(n = > n.StartsWith("S"));

                    Console.WriteLine("Names beginning with S: ");

                    foreach (var item in queryResults) {
                       Console.WriteLine(item);
}

                   Console.Write("Program finished, press Enter/Return to continue: ");
                   Console.ReadLine();
}
```

(4) 编译并执行程序(可以按下 F5 键)。结果也是以 S 开头的 names 列表，且按照它们在数组中声明的顺序排列，如下所示：

```
Names beginning with S:
```

```
Smith
Smythe
Small
Singh
Samba
Program finished, press Enter/Return to continue:
```

示例的说明

与前面一样，Visual C# 2008 会自动引用 System.Linq 名称空间：

```
using System.Linq;
```

再次声明和初始化 names 数组，创建相同的源数据：

```
string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe","Small",
"Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh","Samba", "Fatimah"};
```

LINQ 查询是不同的，它现在是 Where()方法的调用，而不是查询表达式：

```
var queryResults = names.Where(n = > n.StartsWith("S"));
```

C#编译器把λ表达式 n => n.StartWith(“S”))编译为一个匿名方法，Where()在 names 数组的每个元素上执行这个方法。如果λ表达式给某个元素返回 true，该元素就包含在 Where()返回的结果集中。C#编译器从输入数据源(这里是 names 数组)的定义中推断，该 Where()方法应把 string 作为每个元素的输入类型。

许多工作都是在一行代码中完成的。对于像这样最简单的查询，方法语法要比查询语法短，因为不需要 from 或 select 子句，但是，大多数查询都比这个复杂。

示例的剩余部分与前面的代码相同——在 foreach 循环中显示查询的结果，并暂停输出，以便在程序结束执行前看到结果：

```
foreach (var item in queryResults) {
   Console.WriteLine(item);
}

Console.Write("Program finished, press Enter/Return to continue:");
Console.ReadLine();
```

这里不重复这些代码行的说明，因为本章第一个示例已经解释过了。下面继续研究如何使用 LINQ 的更多功能。

26.4　排序查询结果

用 where 子句(或者 Where()方法调用)找到了感兴趣的数据后，LINQ 还可以方便地对得到的数据执行进一步处理，例如，给结果重新排序。下面的示例将以字母顺序给第一个查询的结果排序。

试试看；给查询结果排序

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 可以修改 FirstLINQQuery 示例，或者在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-3-OrderQueryResults。

(2) 打开主源文件 Program.cs，与以前一样，Visual C# 2008 会自动在 Program.cs 中包含 System.
Linq 名称空间。

(3) 在 Program.cs 的 Main()方法中添加如下代码，其中突出显示了与第一个示例不同的代码：

```
static void Main(string[] args)
{
          string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
"Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

               var queryResults =
                   from n in names
                   where n.StartsWith(“S”)
                   orderby n
                   select n;

                Console.WriteLine("Names beginning with S ordered alphabetically:
");

                foreach (var item in queryResults) {
                   Console.WriteLine(item);
                }

                Console.Write("Program finished, press Enter/Return to continue:
");
                Console.ReadLine();
}
```

(4) 编译并执行程序。结果是以 S 开头的 names 列表，且按字母顺序排序，如下所示：

```
Names beginning with S:
Samba
Singh
Small
Smith
Smythe
Program finished, press Enter/Return to continue:
```

示例的说明

这个程序与前一个程序几乎相同，只是在查询语句中增加了一行代码：

```
var queryResults =
                from n in names
                where n.StartsWith("S")
                orderby n
                select n;
```

26.5　orderby 子句

orderby 子句如下所示：

```
orderby n
```

与 where 子句一样，orderby 子句也是可选的。只要添加一行，就可以给任意查询的结果排序，如果不使用LINQ，就需要添加至少几行代码和几个方法或集合，来存储根据选择执行的排序算法重新排序的结果。如果有多个需要排序的类型，就需要为每个类型执行一系列排序方法。而使用 LINQ 不需要做这些工作，只需在查询语句中添加一个子句即可。

orderby 子句默认为升序(A 到 Z)，但可以添加 descending 关键字，指定降序(Z 到 A)：

```
orderby n descending
```

这会使示例的结果变成：

```
Smythe
Smith
Small
Singh
Samba
```

另外，可以按照任意表达式进行排序，而无需重新编写查询。例如，要按照姓名中的最后一个字母排序，而不是按一般的字母顺序排序，就只需添加如下orderby 子句：

```
orderby n.Substring(n.Length - 1)
```

结果如下：

```
Samba
Smythe
Smith
Singh
Small
```

注意最后一个字母按字母顺序排序(a e h h l)。

26.6　用方法语法排序

要给使用方法语法的查询添加排序等功能，只需给每个要在基于方法的 LINQ 查询上执行的 LINQ 操作添加一个方法调用，这也很简单。

试试看：用方法语法排序

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 可以修改 26-2 LINQMethodSyntax 示例，或者在 C:\Beg VCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-4-OrderMethodSyntax。

(2) 在 Program.cs 的 Main()方法中添加如下代码，Visual C# 2008 会自动在 Program.cs 中包含 System.Linq 名称空间。

(3) 下面的代码中突出显示了与前面方法语法示例不同的代码：

```
static void Main(string[] args)
{
          string[] names = { "Alonso", "Zheng", "Smith", "Jones", "Smythe",
"Small", "Ruiz", "Hsieh", "Jorgenson", "Ilyich", "Singh", "Samba", "Fatimah" };

          var queryResults = names.OrderBy(n = > n).Where(n = > n.StartsWith("S"));

          Console.WriteLine("Names beginning with S: ");
          foreach (var item in queryResults) {
               Console.WriteLine(item);
          }

          Console.Write("Program finished, press Enter/Return to continue: ");
          Console.ReadLine();
}
```

(4) 编译并执行程序。结果是以 S 开头的 names 列表，且按字母顺序排序，与前一个示例相同。

示例的说明

这个程序与前一个方法语法示例几乎相同，只是在 Where()方法前面添加了 LINQ 方法 OrderBy()的调用：

```
var queryResults = names.OrderBy(n = > n).Where(n = > n.StartsWith(“S”));
```

在 IntelliSense 中输入代码时会看到，OrderBy()方法返回一个 IOrderedEnumerable，这是 IEnumerable 接口的一个超集，所以可以在其上调用 Where()方法，与其他返回 IEnumerable 接口的方法一样。

提示：

编译器推断，目前处理的是 string 数据，所以在 IntelliSense 中显示的数据类型是 Iordered Enumerable<string>和 IEnumerable<string>。

需要把一个λ表达式传送给 OrderBy()方法，告诉它用于排序的函数是什么。我们传送了最简单的λ表达式 n=>n，因为只需要按照元素本身排序。在查询语法中，不需要创建这个额外的λ表达式。

为了给元素逆序排序，可以调用 OrderByDescending()方法：

```
varqueryResults=names.OrderByDescending(n=>n).Where(n=>n.StartsWith("S"));
```

这会生成与查询语法版本中使用的 orderby n descending 子句相同的结果。

为了不按照元素的值排序，可以修改传送给 OrderBy()方法的λ表达式。例如，要按照每个姓名的最后一个字母排序，可以使用λ表达式 n=>n.Substring(n.Length-1)，把它传送给 OrderBy()方法，如下所示：

```
var queryResults =
     names.OrderBy(n=>n.Substring(n.Length-1)).Where(n=>n.StartsWith("S"));
```

这会生成与上一个示例相同的结果，但按照每个姓名的最后一个字母排序。

26.7　查询大型数据集

这个 LINQ 语法非常好，但其要点是什么？我们只要查看源数组，就可以看出需要的结果，为什么要查询这种一眼就能看出结果的数据源呢？如前所述，有时查询的结果不那么明显。在下面的示例中，就创建了一个非常大的数字数组，并用 LINQ 查询它。

试试看：查询大型数据集

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-5-LargeNumberQuery。与以前一样，创建项目时，Visual Studio 2008 会自动在 Program.cs 中包含 LINQ 名称空间。

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

(2) 在 Main()方法中添加如下代码：

```
static void Main(string[] args)
{
          int[] numbers = generateLotsOfNumbers(12345678);

             var  queryResults =
                  from n in numbers
                  where n < 1000
                  select n
                  ;

             Console.WriteLine("Numbers less than 1000: ");
             foreach (var item in queryResults)
             {
                  Console.WriteLine(item);
             }
              Console.Write("Programfinished,pressEnter/Returntocontinue:");
             Console.ReadLine();
}
```

(3) 添加如下方法，生成一个随机数列表：

```
private static int[] generateLotsOfNumbers(int count)
{
     Random generator = new Random(0);
     int[] result = new int[count];
```

```
        for (int i = 0; i < count; i++)
        {
            result[i] = generator.Next();
        }
        return result;
    }
```

(4) 编译并执行程序。结果是一个小于 1000 的数字列表，如下所示：

```
Numbers less than 1000:
714
24
677
350
257
719
584
Program finished, press Enter/Return to continue:
```

示例的说明

与前面一样，第一步是引用 System.Linq 名称空间，这是在创建项目时由 Visual C# 2008 自动引用的：

```
using System.Linq;
```

下一步是创建一些数据，本例中是创建并调用 generateLotsOfNumbers()方法：

```
    int[] numbers = generateLotsOfNumbers(12345678);

private static int[] generateLotsOfNumbers(int count)
{
    Random generator = new Random(0);
    int[] result = new int[count];
    for (int i = 0; i < count; i++)
    {
        result[i] = generator.Next();
    }
    return result;
}
```

这不是一个小数据集，数组中有 120 万个数字！在本章最后的一个练习中，需要修改传送给 generateLotsOfNumbers()方法的 size 参数，生成数量不同的随机数，看看这会对查询结果有什么影响。在做练习时会看到，这里的 size 参数 12345678 非常大，足以找出一些小于 1000 的随机数，显示为第一个查询的结果。

数值应随机分布在有符号的整数范围内(从 0 到超过 20 亿)。用种子值 0 创建随机数生成器，可以确保每次创建相同的随机数集合，这是可以重复的，所以会获得相同的查询结果，但在尝试一些查询之前，并不知道查询结果是什么。而 LINQ 使这些查询很容易编写。

查询语句本身类似于前面用于 names 数组的查询，也是选择某些满足条件的数字(这里是条件的数字小于 1000)：

```
var queryResults =
    from n in numbers
    where n < 1000
    select n
```

这次不需要 orderby 子句，但处理时间略长(对于这个查询，处理时间的变化不太明显，但下一个示例会改变选择条件，处理时间的变化就比较明显了)。

用 foreach 语句输出查询的结果，与前面的示例相同：

```
Console.WriteLine(“Numbers less than 1000:”);
foreach (var item in queryResults) {
       Console.WriteLine(item);
 }
```

同样，将结果输出到控制台上，并读取一个字符以暂停输出：

```
Console.Write("Program finished, press Enter/Return to continue: ");
Console.ReadLine();
```

后面所有的示例都有暂停代码，但不再列出，因为每个示例的暂停代码都相同。

LINQ 很容易修改查询条件，演示数据集的不同特性。但是，根据查询返回的结果数，每次都输出所有的结果是没有意义的。下一节将说明 LINQ 提供的合计运算符是如何处理这种情况的。

26.8　合计运算符

查询给出的结果常常超出了我们的期望。如果要修改大数查询程序的条件，只需列出大于 1000 的数字，而不是小于 1000 的数字，这会得到非常多的查询结果，数字会不停地显示出来。

LINQ 提供了一组合计运算符，可以分析查询的结果，而无需迭代所有的结果。表 26-1 中列出的合计运算符是数字结果集最常用的，例如，大数查询的结果就常用这些合计运算符，如果读者使用过数据库查询语言如 SQL，就会很熟悉这些运算符。

表　26-1

运 算 符	说　明
Count()	结果的个数
Min()	结果中的最小值
Max()	结果中的最大值
Average()	数字结果的平均值
Sum()	所有数字结果的总和

还有更多的合计运算符，如 Aggregate()，它们可以执行代码，并允许编写自己的合计函数。但是，这些都用于高级用户，超出了本书的讨论范围。

提示：

合计运算符返回一个简单的标量类型，而不是一系列结果，所以使用它们会强制立即执行查询，而不是延迟执行。

下面的示例修改大数查询，并使用合计运算符和 LINQ 分析大数查询的大于版本中的结果集。

试试看：数字合计运算符

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 这个示例可以修改前面的 LargeNumberQuery 示例，或者在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台项目 26-6-NumericAggregates。

(2) 与以前一样，创建项目时，Visual Studio 2008 会自动在 Program.cs 中包含 LINQ 名称空间。只需修改 Main()方法，如下面的代码所示。与上一个例子一样，这个查询也不使用 orderby 子句。但是 where 子句中的条件与前一个例子相反(数字应大于 1000(n>1000)，而不是小于 1000)。

```
static void Main(string[] args)
{
    int[] numbers = generateLotsOfNumbers(12345678);

    Console.WriteLine("Numeric Aggregates");

    var queryResults =
        from n in numbers
         where n > 1000
        select n
       ;

    Console.WriteLine("Count of Numbers > 1000");
    Console.WriteLine(queryResults.Count());

    Console.WriteLine("Max of Numbers > 1000");
    Console.WriteLine(queryResults.Max());

    Console.WriteLine("Min of Numbers > 1000");
    Console.WriteLine(queryResults.Min());

    Console.WriteLine("Average of Numbers > 1000");
    Console.WriteLine(queryResults.Average());

    Console.WriteLine("Sum of Numbers > 1000");
    Console.WriteLine(queryResults.Sum(n = > (long) n));

    Console.Write("Program finished, press Enter/Return to continue: ");
    Console.ReadLine();
```

```
}
```

(3) 添加上一个示例中使用的 generateLotsOfNumbers()方法(如果不存在)：

```
private static int[] generateLotsOfNumbers(int count)
{
   Random generator = new Random(0);
   int[] result = new int[count];
   for (int i = 0; i < count; i++)
   {
      result[i] = generator.Next();
   }
   return result;
}
```

(4) 编译并执行，显示个数、最小值、最大值和平均值，如下所示：

```
Numeric Aggregates
Count of Numbers > 1000
12345671
Maximum of Numbers > 1000
2147483591
Minimum of Numbers > 1000
1034
Average of Numbers > 1000
1073643807.50298
Sum of Numbers > 1000
13254853218619179
Program finished, press Enter/Return to continue:
```

这个查询生成的结果多于上一个例子(超过 120 万)。在这个结果集上使用 orderby，对性能会有很显著的影响。结果集中的最大值超过 20 亿，最小值刚刚大于 1000。平均值大约是 10 亿，接近数字范围的中间值。看来，Rand()函数会生成分布均匀的数字。

示例的说明

程序的第一部分与上一个例子完全相同，也是引用 System.Linq 名称空间，然后用 generateLotsOfNumbers()方法生成源数据：

```
int[] numbers = generateLotsOfNumbers(12345678);
```

查询也与上一个例子相同，只是把 where 条件从小于改为大于：

```
var queryResults =
    from n in numbers
     where n > 1000
     select n;
```

如前所述，使用大于条件的这个查询生成的结果远远多于小于查询(对这个数据集而言)。使用合计运算符可以分析查询结果，而无需输出每个结果，或者在 foreach 循环中比较它们。每个合计运算符都类似于一个可以在结果集上调用的方法，也类似于在集合类型上调用的方法。

下面看看合计运算符的用法：

Count() 返回查询结果中的行数，在这个例子中是 12345671 行：

```
Console.WriteLine("Count of Numbers > 1000");
Console.WriteLine(queryResults.Count());
```

Max() 返回查询结果中的最大值，在这个例子中是大于 20 亿的一个数 2147483591，它非常接近 int 的最大值(int.MaxValue 或 2147483647)。

```
Console.WriteLine("Max of Numbers > 1000");
Console.WriteLine(queryResults.Max());
```

Min() 返回查询结果中的最小值，在这个例子中是 1034。

```
Console.WriteLine("Min of Numbers > 1000");
Console.WriteLine(queryResults.Min());
```

Average() 返回查询结果中的平均值，在这个例子中是 1073643807.50298，它非常接近 1000 到 20 亿的值范围的中间值。对于随机的大数而言，这个中间值没有什么意义，但说明了可以对查询结果进行分析。本章的最后一部分将使用这些运算符对面向业务的数据进行更切合实际的分析。

```
Console.WriteLine("Average of Numbers > 1000");
Console.WriteLine(queryResults.Average());
```

Sum() 注意，在此给 Sum()方法传送了λ表达式 n=>(long) n，以获得所有数字的总和。与 Count()、Min()、Max()等相同，Sum()有一个无参数的重载版本，但使用 Sum()方法的这个版本会导致溢出错误，因为数据集中有太多数字，它们的总和太大，不能放在 Sum()方法的无参数重载版本返回的标准的 32 位 int 中。λ表达式允许把 Sum()方法的结果转换为 64 位长整数，它可以保存超过 13^{10} 的数字 13254853218619179，而不出现溢出。λ表达式允许执行这个转换。

```
Console.WriteLine("Sum of Numbers > 1000");
Console.WriteLine(queryResults.Sum(n = > (long) n));
```

提示：

Count()返回 32 位 int，LINQ 还提供了一个 LongCount()方法，它在 64 位整数中返回查询结果的个数。但有一个特殊情况：如果需要数字的 64 位版本，所有其他运算符都需要一个λ表达式或转换方法调用。

26.9 查询复杂的对象

前面的例子说明了 LINQ 查询可以处理各种简单类型，例如，数字和字符串。下面看看如何使用 LINQ 查询较复杂的对象。我们要创建一个简单的 Customer 类，它带有足够的信息，可以创建有趣的查询。

试试看：查询复杂的对象

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-7-QueryComplexObjects。

(2) 在 Program.cs 的 Program 类开头，给 Customer 类添加如下简短的类定义：

```
class Customer
{
   public string ID { get; set; }
   public string City { get; set; }
   public string Country { get; set; }
   public string Region { get; set; }
   public decimal Sales { get; set; }

   public override string ToString()
   {
      return "ID: "+ ID + "City:" + City + "Country:" + Country +
                "Region: " + Region + " Sales: " + Sales;
   }
}
```

(3) 在 Program.cs 的 Main()方法中添加如下代码：

```
static void Main(string[] args)
{
      List < Customer > customers = new List < Customer > {
         new Customer { ID="A", City="New York", Country="USA",
                                        Region="North America", Sales=9999},
         new Customer { ID="B", City="Mumbai", Country="India",
                                        Region="Asia", Sales=8888 },
         new Customer { ID="C", City="Karachi", Country="Pakistan",
                                        Region="Asia", Sales=7777 },
         new Customer { ID="D", City="Delhi", Country="India",
                                        Region="Asia", Sales=6666 },
         new Customer { ID="E", City="S   o Paulo", Country="Brazil",
                                        Region="South America", Sales=5555 },
         new Customer { ID="F", City="Moscow", Country="Russia",
                                        Region="Europe", Sales=4444 },
         new Customer { ID="G", City="Seoul", Country="Korea", Region="Asia",
                                        Sales=3333 },
         new Customer { ID="H", City="Istanbul", Country="Turkey",
                                        Region="Asia", Sales=2222 },
        new Customer { ID="I", City="Shanghai", Country="China", Region="Asia",
                                        Sales=1111 },
         new Customer { ID="J", City="Lagos", Country="Nigeria",
                                        Region="Africa", Sales=1000 },
         new Customer { ID="K", City="Mexico City", Country="Mexico",
                                        Region="North America", Sales=2000 },
         new Customer { ID="L", City="Jakarta", Country="Indonesia",
```

```
                                        Region="Asia", Sales=3000 },
            new Customer { ID="M", City="Tokyo", Country="Japan",
                                        Region="Asia", Sales=4000 },
            new Customer { ID="N", City="Los Angeles", Country="USA",
                                        Region="North America", Sales=5000 },
            new Customer { ID="O", City="Cairo", Country="Egypt",
                                        Region="Africa", Sales=6000 },
            new Customer { ID="P", City="Tehran", Country="Iran",
                                        Region="Asia", Sales=7000 },
            new Customer { ID="Q", City="London", Country="UK",
                                        Region="Europe", Sales=8000 },
            new Customer { ID="R", City="Beijing", Country="China",
                                        Region="Asia", Sales=9000 },
            new Customer { ID="S", City="Bogotá", Country="Colombia",
                                        Region="South America", Sales=1001 },
            new Customer { ID="T", City="Lima", Country="Peru",
                                        Region="South America", Sales=2002 }
          };
            var  queryResults =
                 from c in customers
                 where c.Region == "Asia"
                 select c
               ;
            Console.WriteLine("Customers in Asia: ");
            foreach (Customer c in queryResults)
            {
                 Console.WriteLine(c);
            }
            Console.Write("Program finished, press Enter/Return to continue: ");
            Console.ReadLine();
          }
}
```

(4) 编译并执行程序，结果是来自亚洲的顾客列表：

```
Customers in Asia:
ID: B City: Mumbai Country: India Region: Asia Sales: 8888
ID: C City: Karachi Country: Pakistan Region: Asia Sales: 7777
ID: D City: Delhi Country: India Region: Asia Sales: 6666
ID: G City: Seoul Country: Korea Region: Asia Sales: 3333
ID: H City: Istanbul Country: Turkey Region: Asia Sales: 2222
ID: I City: Shanghai Country: China Region: Asia Sales: 1111
ID: L City: Jakarta Country: Indonesia Region: Asia Sales: 3000
ID: M City: Tokyo Country: Japan Region: Asia Sales: 4000
ID: P City: Tehran Country: Iran Region: Asia Sales: 7000
ID: R City: Beijing Country: China Region: Asia Sales: 9000
Program finished, press Enter/Return to continue:
```

示例的说明

在 Customer 类的定义中使用了 C# 3.0 的自动属性功能，来声明 Customer 类的公共属性

(ID、City、Country、Region、Sales)，而不必为每个属性显示编写私有实例变量和 get/set 代码：

```
class Customer
   {
        public string ID { get; set; }
        public string City { get; set; }
          …
```

为 Customer 类编写的唯一一个方法是 ToString()方法的重写版本，该方法为 Customer 实例提供字符串表示：

```
public override string ToString()
{
     return "ID:"+ID + "City:"+City +"Country:"+Country+
            "Region:"+Region +"Sales:"+ Sales;
}
```

使用这个 ToString()方法可以简化查询结果的输出，如后面的代码所示。

在 Program 类的 Main()方法中，用 C# 3.0 集合/对象初始化语法创建了一个强类型化的集合，其类型是 Customer，这样就不需要编写构造方法，再调用该构造方法创建每个列表成员了：

```
List < Customer > customers = new List < Customer > {
     new Customer { ID="A", City="New York", Country="USA",
                                    Region="North America", Sales=9999},
     new Customer { ID=”B”, City="Mumbai", Country="India",
                                       Region="Asia", Sales=8888 },
…
```

顾客位于世界各地，我们的数据中有足够的地理信息，可以为查询建立有趣的选择条件和组合条件。

在 Main()方法中，创建查询语句，本例选择来自亚洲的顾客：

```
var queryResults =
     from c in customers
     where c.Region =="Asia"
     select c
   ;
```

这个查询应很眼熟，在其他示例中使用的也是 from…where…select 查询，只是结果列表中的每一项都是一个对象(Customer)，而不是简单的 string 或 int。接着在 foreach 循环中输出结果：

```
Console.WriteLine("Customers in Asia: ");
foreach (Customer c in queryResults)
{
Console.WriteLine(c);
}
```

这个 foreach 循环不同于前面的示例。因这我们知道要查询 Customer 对象，所以把迭代变量显式地声明为 Customer 类型：

```
foreach (Customer c in queryResults)
```

可以用关键字 var 声明 c，编译器就会推断出这个迭代变量的类型应是 Customer，但显式声明它会使代码更容易理解。

在循环中，仅编写了一条语句：

```
{
    Console.WriteLine(c);
}
```

这里没有显式输出 Customer 的字段，因为给 Customer 类添加了 ToString()的重写版本。如果没有提供这个 ToString()重写版本，默认的 ToString()方法仅输出类型的名称，如下所示：

```
Customers in Asia:
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
BegVCSharp_26_7_QueryComplexObjects.Customer
Program finished, press Enter/Return to continue:
```

这并不是我们想要的结果。当然，总是可以输出令人感兴趣的 Customer 属性：

```
Console.WriteLine("Customer {0}: {1}, {2}", c.ID, c.City, c.Country);
```

但是，如果只对对象的几个属性感兴趣，把整个对象都放在查询中就很低效了。LINQ 很容易通过投射(projection)创建只包含所需元素的查询结果，如下一节所述。

26.10 投射：在查询中创建新对象

投射是在 LINQ 查询中从其他数据类型中创建新数据类型的技术术语。select 关键字是投射运算符，前面的示例中就使用了这个关键字。如果熟悉 SQL 数据查询语言中的 SELECT 关键字，就很熟悉从数据对象中选择某个字段的操作，而不是选择整个对象。在 LINQ 中，也可以这么做，例如，前面的例子要从 Customer 列表中选择 City 字段，只需修改查询语句中的 select 子句，只引用 City 属性：

```
var queryResults =
    from c in customers
    where c.Region == "Asia"
    select c.City
```

```
    ;
```

这会生成如下结果：

```
Mumbai
Karachi
Delhi
Seoul
Istanbul
Shanghai
Jakarta
Tokyo
Tehran
Beijing
```

甚至可以通过给 select 添加表达式，来转换查询中的数据，对于数字数据类型应如下所示：

```
select n + 1
```

对于字符串数据类型，应如下所示：

```
select s.ToUpper()
```

但是，与 SQL 不同，LINQ 不允许在 select 子句中有多个字段。这表示

```
select c.City, c.Country, c.Sales
```

会生成一个编译错误(需要分号)，因为 select 子句在其参数列表中只有一项。

在 LINQ 中，应在 select 子句中创建一个新对象，来保存查询的结果。如下面的示例所示。

试试看：投射——在查询中创建新对象

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 修改 26-7-QueryComplexObjects，或者在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-8-ProjectionCreateNewObjects。

(2) 如果选择创建新项目，就从 26-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码，这些代码与前面的代码相同。

(3) 在 Main()方法中，在 customers 列表初始化后，输入(或修改)查询，使处理循环如下所示：

```
var queryResults =
  from c in customers
  where c.Region == "North America"
  select new { c.City, c.Country, c.Sales }
  ;
foreach (var item in queryResults)
{
  Console.WriteLine(item);
}
```

(4) Main()方法的其余代码与前面示例的相同。

(5) 编译并执行程序，从来自北美的顾客中选择字段，如下所示：

```
{ City = New York, Country = USA, Sales = 9999 }
{ City = Mexico City, Country = Mexico, Sales = 2000 }
{ City = Los Angeles, Country = USA, Sales = 5000 }
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 customers 列表的初始化与前面例子中的相同。在查询中，把请求的区域改为北美，会使事情复杂一些。对于投射，有趣的改变在于 select 子句的参数：

```
select new { c.City, c.Country, c.Sales }
```

这里在 select 子句中直接使用 C# 3.0 匿名类型创建语法，创建一个未命名的对象类型，它带有 City、Country 和 Sales 属性。select 子句创建了新对象。这样，只会复制这3个属性，完成处理查询的不同阶段。

输出查询结果时，使用了一般的foreach 循环代码，在前面的示例中也使用了相同的循环代码，但 Customers 查询例外：

```
foreach (var item in queryResults)
{
   Console.WriteLine(item);
}
```

这段代码非常一般化，编译器会推断出查询结果的类型，给匿名类型调用正确的方法，我们无需显式编写类型代码。甚至不需要提供ToString()重写方法，因为编译器提供了ToString()方法的默认执行代码，以类似于对象初始化的方式输出属性名称和值。

26.11 投射：方法语法

投射查询的方法语法版本是通过把 LINQ 方法 Select()的调用关联到我们调用的其他 LINQ 方法上来实现的。例如，如果在 Where()方法调用上添加 Select()方法调用，就会得到相同的查询结果：

```
var queryResults = customers.Where(c = > c.Region == "North America")
     .Select(c = > new { c.City, c.Country, c.Sales });
```

在查询语法中需要 select 子句，但在以前并没有看到 Select()方法，因为在 LINQ 方法语法中不需要它，除非在进行投射(改变结果集中所查询的原始类型)。

方法调用的顺序不是固定的，因为 LINQ 方法调用返回的类型执行了 IEnumerable——可以在 Where()方法的结果上调用 Select()方法，反之亦然。但是，根据查询的特性，调用顺序也许很重要。例如，不能像下面这样颠倒 Select()和 Where()的顺序：

```
var queryResults = customers.Select(c = > new { c.City, c.Country, c.Sales })
          .Where(c = > c.Region == "North America");
```

Region 属性未包含在 Select()投射创建的匿名类型(c.City，c.Country，c.Sales)中，所以程

序会在 Where()方法上得到一个编译错误，表示匿名类型不包含 Region 的定义。

但是，如果 Where()方法根据包含在匿名类型中的字段来限制数据，就不会有问题——例如，下面的查询会被编译并执行：

```
var queryResults = customers.Select(c = > new {c.City, c.Country, c.Sales })
        .Where(c = > c.City == "New York");
```

26.12　单值选择查询

在 SQL 数据查询语言中，我们熟悉的另一类查询是 SELECT DISTINCT 查询，该查询可搜索数据中的唯一值，也就是说，值是不重复的。这是使用查询时的一个常见需求。

假定需要在前面示例使用的顾客数据中查找不同的区域，由于在这些数据中没有单独的区域列表，所以需要从顾客列表中找出唯一的、不重复的区域列表。LINQ 提供了 Distinct()方法，它能很容易地找出这些数据，如下面的示例所示。

试试看；投射——单值选择查询

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 修改前面的示例 26-8-ProjectionCreateNewObjects，或者在 C:\Beg VCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-9-SelectDistinctQuery。

(2) 从 26-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码，这些代码与前面的代码相同。

(3) 在 Main()方法中，在 customers 列表初始化后，输入(或修改)查询，如下所示：

```
var queryResults = customers.Select(c = > c.Region).Distinct();
```

(4) Main()方法的其余代码与前面例子的相同。

(5) 编译并执行程序，结果显示的是顾客所在的唯一区域，如下所示：

```
North America
Asia
South America
Europe
Africa
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 customers 列表的初始化与前面例子中的相同。在查询语句中，调用了 Select()方法，用一个简单的λ表达式从 Customer 对象中选择区域，再调用 Distinct()从 Select()中返回唯一的结果：

```
var queryResults = customers.Select(c = > c.Region).Distinct();
```

Distinct()只能在方法语法中使用，所以使用方法语法调用 Select()。还可以调用 Distinct()修改在查询语法中创建的查询：

```
var queryResults = (from c in customers select c.Region).Distinct();
```

查询语法由 C# 3.0 编译器转换为方法语法中的相同 LINQ 方法调用，所以如果对可读性和样式有意义，就可以混合和匹配它们。

26.13 Any 和 All

我们常常需要的另一类查询是确定数据是否满足某个条件，或者确保所有的数据都满足某个条件。例如，需要确定某个产品是否没货了(库存为 0)，或者是否发生了某个事务。

LINQ 提供了两个布尔方法：Any()和 All()，它们可以快速确定对于数据而言，某个条件是 true 还是 false。因此很容易查找到数据，如下面的示例所示。

试试看：使用 Any()和 All()

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 修改前面的示例 26-9-SelectDistinctQuery，或者在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-10-AnyAndAll。

(2) 从 26-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码，这些代码与前面的代码相同。

(3) 在 Main()方法中，在customers 列表初始化和查询声明后，删除处理循环，输入如下所示的代码：

```
bool anyUSA = customers.Any(c = > c.Country == "USA");
if (anyUSA)
{
     Console.WriteLine("Some customers are in the USA");
}
else
{
     Console.WriteLine("No customers are in the USA");
}
bool allAsia = customers.All(c = > c.Region == "Asia");
if (allAsia)
{
     Console.WriteLine("All customers are in Asia");
}
else
{
     Console.WriteLine("Not all customers are in Asia");
}
```

(4) Main()方法的其余代码与前面例子的相同。

(5) 编译并执行程序，信息显示一些顾客来自美国，并不是所有的顾客都来自亚洲：

```
Some customers are in the USA
Not all customers are in Asia
```

```
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 customers 列表的初始化与前面例子中的相同。在第一个查询语句中，调用了 Any()方法，用一个简单的λ表达式检查 Customer Country 字段的值是否是 USA：

```
bool anyUSA = customers.Any(c = > c.Country == “USA”);
```

LINQ 方法 Any()把传送给它的λ表达式 c=>c.Country == “USA”应用于 customers 列表中的所有数据，如果对于列表中的任意顾客，λ表达式是 true，就返回 true。

接着，检查 Any()方法返回的布尔结果变量，输出一个消息，显示查询的结果(Any()方法虽然仅返回 true 或 false，但它会执行一个查询，得到 true 或 false 结果)：

```
if (anyUSA)
{
   Console.WriteLine("Some customers are in the USA");
}
else
{
   Console.WriteLine("No customers are in the USA");
}
```

可以通过一些代码使这个消息更紧凑一些，但这里的代码比较直观，容易理解。anyUSA 变量设置为 true，因为数据集中的确有顾客居住在美国，所以看到了消息 Some customers are in the USA。

在下一个查询语句中，调用了 All()方法，利用另一个简单的λ表达式确定是否所有的顾客都来自亚洲：

```
bool allAsia = customers.All(c = > c.Region == "Asia");
```

LINQ 方法 All()把λ表达式应用于数据集，并返回 false，因为有一些顾客不是来自亚洲。然后根据 allAsia 的值返回相应的消息。

26.14　多级排序

处理了带多个属性的对象后，就要考虑另一种情形了：按一个字段给查询结果排序是不够的，需要查询顾客，并按照区域使结果以字母顺序排列，再按照区域中国家或城市名称来排序。LINQ 很容易完成这个任务，如下面的示例所示。

试试看：多级排序

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 修改前面的示例 26-8-ProjectionCreateNewObjects，或者在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-11-MultiLevelOrdering。

(2) 从 26-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码，这些代码与前面的代码相同。

(3) 在 Main()方法中，在 customers 列表初始化后，输入如下所示的查询：

```
var queryResults =
    from c in customers
    orderby c.Region, c.Country, c.City
    select new { c.ID, c.Region, c.Country, c.City }
   ;
```

(4) 结果处理循环和 Main()方法的其余代码与前面例子中的相同。

(5) 编译并执行程序，从所有顾客中选择出来的属性将先按区域排序，再按国家排序，最后按城市排序，如下所示：

```
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = K, Region = North America, Country = Mexico, City = Mexico City }
{ ID = N, Region = North America, Country = USA, City = Los Angeles }
{ ID = A, Region = North America, Country = USA, City = New York }
{ ID = E, Region = South America, Country = Brazil, City = Sao Paulo }
{ ID = S, Region = South America, Country = Colombia, City = Bogot á }
{ ID = T, Region = South America, Country = Peru, City = Lima }
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 customers 列表的初始化与前面例子的相同。在这个查询中，没有 where 子句，因为要查看所有的顾客，但按顺序列出了要排序的字段，它们放在orderby 子句的一个用逗号分开的列表中：

```
orderby c.Region, c.Country, c.City
```

这很容易，但不太直观，这个简单的字段列表允许放在 orderby 子句中，但不能放在 select 子句中，这就是 LINQ 的工作方式。如果知道 select 子句会创建一个新对象，而根据定义，orderby 子句会逐个字段地执行，就不会觉得这个字段列表难以理解了。

可以给列出的任意字段添加 descending 关键字，反转该字段的排序顺序。例如，若要对查询结果要按照区域升序排序，再按照国家降序排序，只需在列表中的 Country 后面加上 descending 关键字即可，如下所示：

```
orderby c.Region, c.Country descending, c.City
```

添加了descending 关键字后，结果如下：

```
{ ID = J, Region = Africa, Country = Nigeria, City = Lagos }
{ ID = O, Region = Africa, Country = Egypt, City = Cairo }
{ ID = H, Region = Asia, Country = Turkey, City = Istanbul }
{ ID = C, Region = Asia, Country = Pakistan, City = Karachi }
{ ID = G, Region = Asia, Country = Korea, City = Seoul }
{ ID = M, Region = Asia, Country = Japan, City = Tokyo }
{ ID = P, Region = Asia, Country = Iran, City = Tehran }
{ ID = L, Region = Asia, Country = Indonesia, City = Jakarta }
{ ID = D, Region = Asia, Country = India, City = Delhi }
{ ID = B, Region = Asia, Country = India, City = Mumbai }
{ ID = R, Region = Asia, Country = China, City = Beijing }
{ ID = I, Region = Asia, Country = China, City = Shanghai }
{ ID = Q, Region = Europe, Country = UK, City = London }
{ ID = F, Region = Europe, Country = Russia, City = Moscow }
{ ID = N, Region = North America, Country = USA, City = Los Angeles }
{ ID = A, Region = North America, Country = USA, City = New York }
{ ID = K, Region = North America, Country = Mexico, City = Mexico City }
{ ID = T, Region = South America, Country = Peru, City = Lima }
{ ID = S, Region = South America, Country = Colombia, City = Bogot á }
{ ID = E, Region = South America, Country = Brazil, City = Sao Paulo }
Program finished, press Enter/Return to continue:
```

注意，即使国家的顺序被反转了，印度和中国的城市仍按升序排序。

26.15　多级排序方法语法：ThenBy

使用方法语法进行多级排序时，后台的操作比较复杂，它使用了 ThenBy()和 OrderBy()方法。例如，下面的代码会得到与前面创建的示例相同的查询结果：

```
var queryResults = customers.OrderBy(c = > c.Region)
  .ThenBy(c = > c.Country)
  .ThenBy(c = > c.City)
  .Select(c = > new { c.ID, c.Region, c.Country, c.City });
```

多字段列表可以用在查询语法的orderby 子句中，这是很明显的，因为它会转换为一系列 ThenBy()方法调用，逐个字段地执行。编写这些方法调用的顺序非常重要，必须先编写 OrderBy()，因为ThenBy()方法只能在 IOrderedEnumerable 接口上使用，而 IOrderedEnumerable 接口是由OrderBy()生成的。但是，ThenBy()方法可以关联到任意多个 ThenBy()方法调用上，显然，查询语法比方法语法更容易编写。

如果第一个字段是以降序排序，就应调用 OrderByDescending()来指定；如果其他字段要以降序排序，就应调用 ThenByDescending()来指定。例如，在这个例子中，国家要以降序排列，方法语法的查询应如下所示：

```
var queryResults = customers.OrderBy(c = > c.Region)
  .ThenByDescending(c = > c.Country)
```

```
    .ThenBy(c = > c.City)
    .Select(c = > new { c.ID, c.Region, c.Country, c.City });
```

26.16 组合查询

组合查询把数据分解为组，允许按组来排序、计算合计值、比较。这常常是商务环境中最有趣的查询(它驱动了决策系统)。例如，要按照国家或区域比较销售量，确定在哪里开新店或雇佣更多的员工，如下面的示例所示。

试试看：组合查询

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-12-GroupQuery。

(2) 从 26-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer>customers)的代码，这些代码与前面的代码相同。

(3) 在 Main()方法中，在 customers 列表初始化后，输入如下所示的两个查询：

```
var queryResults =
    from c in customers
    group c by c.Region into cg
    select new { TotalSales = cg.Sum(c = > c.Sales), Region = cg.Key }
   ;
var orderedResults =
    from cg in queryResults
    orderby cg.TotalSales descending
    select cg
   ;
```

(4) 在 Main()方法中，添加下面的输出语句和 foreach 处理循环：

```
Console.WriteLine("Total\t: By\nSales\t: Region\n-----\t ------");
foreach (var item in orderedResults)
{
   Console.WriteLine(item.TotalSales + "\t: " + item.Region);
}
```

(5) 结果处理循环和Main()方法中的其余代码与前面例子中的相同。编译并执行程序，下面是组合结果：

```
Total  : By
Sales  : Region
-----   ------
52997  : Asia
16999  : North America
12444  : Europe
8558   : South America
7000   : Africa
```

示例的说明

Customer 类和 customers 列表的初始化与前面例子的相同。

组合查询中的数据通过一个键(key)字段来组合，每个组中的所有成员都共享这个字段值。在这个例子中，键字段是 Region：

```
group c by c.Region
```

要计算每个组的总和，应生成一个新的结果集 cg：

```
group c by c.Region into cg
```

在 select 子句中，投射了一个新的匿名类型，其属性是总销售量(通过引用cg 结果集来计算)和组的键字段值，后者是用特殊的组 Key 来引用的：

```
select new { TotalSales = cg.Sum(c = > c.Sales), Region = cg.Key }
```

组的结果集执行了 LINQ 接口 IGrouping，它支持 Key 属性。我们总是要以某种方式引用 Key 属性，来处理组合的结果，因为该属性表示创建数据中的每个组时使用的条件。

要按照TotalSales 字段对结果降序排序，以便查看某区域中的最高销售量、次高销售量等等，需要创建第二个查询，对组合查询的结果排序：

```
var orderedResults =
  from cg in queryResults
  orderby cg.TotalSales descending
  select cg
;
```

第二个查询是一个标准的 select 查询，带一个 orderby 子句，与前面示例中的相同。但它没有使用任何 LINQ 组合功能，只是数据源来自于前面的组合查询。

接着输出结果，用一些格式化代码显示带有列标题的数据，在总销售量与组名之间显示了分隔符：

```
Console.WriteLine("Total\t: By\nSales\t: Region\n-----\t ------");
foreach (var item in orderedResults)
{
     Console.WriteLine(item.TotalSales + "\t: " + item.Region);
}
```

可以用更复杂的方式进行格式化，指定字段宽度，总销售量右对齐，但这只是一个例子，不需要这么多格式，能看清数据，理解代码做了些什么就足够了。

26.17　Take 和 Skip

假定需要数据集中销售量位于前 5 名的顾客。我们事先并不知道这 5 名顾客的销售量是多少，所以不能使用 where 条件查找他们。

一些 SQL 数据库如 Microsoft SQL Server 实现了 TOP 运算符，所以可以执行命令 SELECT Top 5 FROM …，获得前 5 名顾客的数据。

与这个操作对应的 LINQ 方法是 Take()，它可以从查询结果中提取前 n 个结果。实际上，这个方法需要和 orderby 子句一起使用，才能获得前 n 个结果。但 orderby 子句并不是必需的，因为有时知道数据已经按指定的顺序排列好了，或者只需要前 n 个结果，而不必考虑它们的顺序。

Take()的反面是 Skip()，它可以跳过前 n 个结果，返回剩余的结果。Take()和 Skip()在 LINQ 文档说明中称为分区运算符，因为它们把结果集分为前 n 个结果(Take())和其余的结果(Skip())。

下面的示例对顾客列表数据使用了 Take()和 Skip()。

试试看：使用 Take()和 Skip()

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-13-TakeAndSkip。

(2) 从 26-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer> customers)的代码。

(3) 在 Main()方法中，在 customers 列表初始化后，输入如下所示的查询：

```
//query syntax
var queryResults =
    from c in customers
    orderby c.Sales descending
    select new { c.ID, c.City, c.Country, c.Sales }
   ;
```

(4) 输入两个结果处理循环，一个使用 Take()，另一个使用 Skip()：

```
Console.WriteLine("Top Five Customers by Sales");
foreach (var item in queryResults.Take(5))
{
     Console.WriteLine(item);
}

Console.WriteLine("Customers Not In Top Five");
foreach (var item in queryResults.Skip(5))
{
   Console.WriteLine(item);
}
```

(5) 编译并执行程序，结果显示的是前 5 名顾客和剩余的顾客：

```
Top Five Customers by Sales
{ ID = A, City = New York, Country = USA, Sales = 9999 }
{ ID = R, City = Beijing, Country = China, Sales = 9000 }
{ ID = B, City = Mumbai, Country = India, Sales = 8888 }
{ ID = Q, City = London, Country = UK, Sales = 8000 }
{ ID = C, City = Karachi, Country = Pakistan, Sales = 7777 }
Customers Not In Top Five
{ ID = P, City = Tehran, Country = Iran, Sales = 7000 }
{ ID = D, City = Delhi, Country = India, Sales = 6666 }
```

```
{ ID = O, City = Cairo, Country = Egypt, Sales = 6000 }
{ ID = E, City = Sao Paulo, Country = Brazil, Sales = 5555 }
{ ID = N, City = Los Angeles, Country = USA, Sales = 5000 }
{ ID = F, City = Moscow, Country = Russia, Sales = 4444 }
{ ID = M, City = Tokyo, Country = Japan, Sales = 4000 }
{ ID = G, City = Seoul, Country = Korea, Sales = 3333 }
{ ID = L, City = Jakarta, Country = Indonesia, Sales = 3000 }
{ ID = H, City = Istanbul, Country = Turkey, Sales = 2222 }
{ ID = T, City = Lima, Country = Peru, Sales = 2002 }
{ ID = K, City = Mexico City, Country = Mexico, Sales = 2000 }
{ ID = I, City = Shanghai, Country = China, Sales = 1111 }
{ ID = S, City = Bogotá, Country = Colombia, Sales = 1001 }
{ ID = J, City = Lagos, Country = Nigeria, Sales = 1000 }
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 customers 列表的初始化与前面例子中的相同。

主查询由查询语法的 from…orderby…select 语句组成，类似于本章前面创建的查询，只是其中没有 where 子句，因为我们要获得所有的顾客(按从高到低的销售量排序)：

```
var queryResults =
                from c in customers
                orderby c.Sales descending
                select new { c.ID, c.City, c.Country, c.Sales }
```

这个示例与前面的示例略有不同：这个示例在对查询结果执行 foreach 循环之前，并没有应用运算符，因为本例要重用查询结果。首先使用 Take(5)获得前 5 名顾客：

```
foreach (var item in queryResults.Take(5))
```

接着使用 Skip(5)跳过前 5 项(这些项刚才已经输出了)，从原来的查询结果集中输出剩余的顾客：

```
foreach (var item in queryResults.Skip(5))
```

输出结果和暂停屏幕显示的代码与前面示例中的相同，只是消息有一点儿变化，这里不再重复。

26.18　First 和 FirstOrDefault

假定需要在数据集中查找一名来自非洲的顾客，我们需要实际的数据，而不是 true/false 值或者包含所有匹配值的结果集。

LINQ 通过 First()方法提供了这个功能，它返回结果集中第一个匹配给定条件的元素。如果没有来自非洲的顾客，LINQ 还提供了方法 FirstOrDefault()来处理这种情况，而无需添加错误处理代码。

下面的示例给 customers 列表数据使用了 First()和 FirstOrDefault()方法。

试试看：使用 First 和 FirstOrDefault

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-14-FirstOrDefault。

(2) 从 26-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer>customers)的代码。

(3) 在 Main()方法中，在 customers 列表初始化后，输入如下所示的查询：

```
var queryResults = from c in customers
                   select new { c.City, c.Country, c.Region }
                   ;
```

(4) 使用 First()和 FirstOrDefault()方法输入如下查询：

```
Console.WriteLine("A customer in Africa");
Console.WriteLine(queryResults.First(c = > c.Region == "Africa"));

Console.WriteLine("A customer in Antarctica");
Console.WriteLine(queryResults.FirstOrDefault(c = > c.Region == "Antarctica"));
```

(5) 编译并执行程序，结果如下：

```
A customer in Africa
{ City = Lagos, Country = Nigeria, Region = Africa }
A customer in Antarctica
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 customers 列表的初始化与前面示例中的相同。

主查询由查询语法的 from…orderby…select 语句组成，类似于本章前面创建的查询，只是其中没有 where 和 orderby 子句，我们用 select 语句选择了感兴趣的字段。本例中选择了City、Country 和 Region 属性：

```
var queryResults = from c in customers
             select new { c.City, c.Country, c.Region }
             ;
```

First()运算符返回一个对象值，而不是结果集，所以不需要创建 foreach 循环，直接输出结果即可：

```
Console.WriteLine(queryResults.First(c = > c.Region == "Africa"));
```

这行代码找到了一名顾客，输出 City=Lagos, Country=Nigeria, Region=Africa。接着，使用 FirstOrDefault()查询南极洲区域：

```
Console.WriteLine(queryResults.FirstOrDefault(c = > c.Region == "Antarctica"));
```

这行代码找不到任何结果，所以返回空(空结果集)，输出空白。如果给南极洲查询使用 First()运算符，而不是 FirstOrDefault()，就会得到如下异常：

```
System.InvalidOperationException: Sequence contains no matching element
```

如果搜索条件不满足，FirstOrDefault()就会返回列表的默认元素，而 First()为这个匿名类型返回空。对于南极洲查询，会得到一个异常。

输出结果和暂停屏幕显示的代码与前面示例中的相同，只是消息有一点儿变化。

26.19　集运算符

LINQ 提供了标准的集运算符，如 Union()和 Intersect()，对查询结果执行操作。在前面编写 Distinct()查询时，就使用了一个集运算符。

下面的示例添加了一个简单的订单列表，它是由假想的顾客提交的，并使用标准的集运算符来匹配已有的顾客。

试试看：集运算符

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-15-SetOperators。

(2) 从 26-7-QueryComplexObjects 示例中复制创建 Customer 类的代码和初始化顾客列表(List<Customer>customers)的代码。

(3) 在 Customer 类的后面添加如下 Order 类：

```
class Order
{
   public string ID { get; set; }
   public decimal Amount { get; set; }
}
```

(4) 在 Main()方法中，在 customers 列表初始化后，用如下数据创建并初始化一个 orders 列表：

```
List < Order > orders = new List < Order > {
  new Order { ID="P", Amount=100 },
  new Order { ID="Q", Amount=200 },
  new Order { ID="R", Amount=300 },
  new Order { ID="S", Amount=400 },
  new Order { ID="T", Amount=500 },
  new Order { ID="U", Amount=600 },
  new Order { ID="V", Amount=700 },
  new Order { ID="W", Amount=800 },
  new Order { ID="X", Amount=900 },
  new Order { ID="Y", Amount=1000 },
  new Order { ID="Z", Amount=1100 }
};
```

(5) 在 orders 列表初始化后，输入如下所示的查询：

```
var customerIDs =
   from c in customers
   select c.ID
```

```
    ;
var orderIDs =
    from o in orders
    select o.ID
    ;
```

(6) 用 Intersect()输入如下所示的查询：

```
var customersWithOrders = customerIDs.Intersect(orderIDs);
Console.WriteLine("Customer IDs with Orders: ");
foreach (var item in customersWithOrders)
{
    Console.Write("{0} ", item);
}
Console.WriteLine();
```

(7) 接着，用 Except()输入如下所示的查询：

```
Console.WriteLine("Order IDs with no customers: ");
var ordersNoCustomers = orderIDs.Except(customerIDs);
foreach (var item in ordersNoCustomers)
{
     Console.Write("{0} ", item);
}
Console.WriteLine();
```

(8) 最后，用 Union()输入如下所示的查询：

```
Console.WriteLine("All Customer and Order IDs: ");
var allCustomerOrderIDs = orderIDs.Union(customerIDs);
foreach (var item in allCustomerOrderIDs)
{
    Console.Write("{0} ", item);
}
Console.WriteLine();
```

(9) 编译并执行程序，结果如下：

```
Customers IDs with Orders:
P Q R S T
Order IDs with no customers:
U V W X Y Z
All Customer and Order IDs:
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 customers 列表的初始化与前面示例中的相同。新的 Order 类类似于 Customer 类，也使用了 C# 3.0 自动属性功能来声明公共属性(ID、Amount)：

```
class Order
{
```

```
    public string ID { get; set; }
    public decimal Amount { get; set; }
}
```

与 Customer 类一样，这也是一个简化的例子，只包含足以使查询工作的数据。

使用两个简单的 from…select 查询从 Customer 类和 Order 类中获得 ID 字段：

```
var customerIDs =
    from c in customers
    select c.ID
    ;
var orderIDs =
    from o in orders
    select o.ID
    ;
```

接着使用 Intersect()集运算符查找在 orderIDs 结果中有订单的顾客 ID。只有在两个结果集中都有的 ID 才包含在交集中：

```
var customersWithOrders = customerIDs.Intersect(orderIDs);
```

提示：

集运算符要求集成员有相同的类型，才能确保得到希望的结果。这里利用了一个事实：两个对象类型中的 ID 都是字符串，有相同的语义(类似于数据库中的外键)。

结果集的输出利用了一个事实：ID 是单个字符，所以使用 Console.Write()调用，而不是 WriteLine()调用，直到 foreach 循环的最后才使用 WriteLine()，使输出紧凑、简洁：

```
Console.WriteLine("Customer IDs with Orders: ");
foreach (var item in customersWithOrders)
{
    Console.Write("{0} ", item);
}
Console.WriteLine();
```

在其余的 foreach 循环中也使用了这个输出逻辑。

接着，使用 Except()运算符查找没有匹配顾客的订单 ID：

```
Console.WriteLine("Order IDs with no customers: ");
var ordersNoCustomers = orderIDs.Except(customerIDs);
```

最后，使用 Union()运算符查找所有顾客 ID 和订单 ID 字段的并集：

```
  Console.WriteLine("All Customer and Order IDs: ");
  var allCustomerOrderIDs = orderIDs.Union(customerIDs);
```

注意，ID 的输出顺序与它们在顾客和订单列表中的顺序相同，只是删除了重复的项。

暂停屏幕显示的代码与前面示例中的相同。

集运算符非常有用，但使用它们的实际优势因为受到所有对象的类型都必须相同的限制而缩小了。需要处理类型类似的结果集时，这些运算符是很有用的，但在需要处理不同的对

象类型时，应采用专门为处理不同对象类型而设计的机制，例如，join 语句。

26.20 Join 查询

刚才用一个共享的键字段(ID)创建的 customers 和 orders 列表等数据集可以执行 join 查询，即可以用一个查询搜索两个列表中相关的数据，用键字段把结果连接起来。这类似于 SQL 数据查询语言中的 JOIN 操作。LINQ 在查询语法中提供了 join 命令，如下面的示例所示。

试试看：Join 查询

按照下面的步骤在 Visual Studio 2008 中创建示例：

(1) 在 C:\BegVCSharp\Chapter26 目录下创建一个新的控制台应用程序 26-16-JoinQuery。

(2) 从前面的示例中复制创建 Customer 类和 Order 类的代码，以及初始化顾客列表(List<Customer>customers)和订单列表(List<Order>orders)的代码。

(3) 在 Main()方法中，在 customers 和 orders 列表初始化后，输入如下所示的查询：

```
var queryResults =
    from c in customers
    join o in orders on c.ID equals o.ID
    select new { c.ID, c.City, SalesBefore = c.Sales, NewOrder = o.Amount,
                                                SalesAfter = c.Sales+o.Amount };
```

(4) 用标准的 foreach 查询处理循环结束程序：

```
foreach (var item in queryResults)
{
   Console.WriteLine(item);
}
```

(5) 编译并执行程序，结果如下：

```
{ ID = P, City = Tehran, SalesBefore = 7000, NewOrder = 100, SalesAfter = 7100 }
{ ID = Q, City = London, SalesBefore = 8000, NewOrder = 200, SalesAfter = 8200 }
{ ID = R, City = Beijing, SalesBefore = 9000, NewOrder = 300, SalesAfter = 9300 }
{ ID = S, City = Bogotá, SalesBefore = 1001, NewOrder = 400, SalesAfter = 1401 }
{ ID = T, City = Lima, SalesBefore = 2002, NewOrder = 500, SalesAfter = 2502 }
Program finished, press Enter/Return to continue:
```

示例的说明

Customer 类和 Order 类、customers 和 orders 列表的声明和初始化与前面示例中的相同。

查询使用 join 关键字通过 Customer 类和Order 类的 ID 字段，把每个顾客与其对应的订单连接起来：

```
var queryResults =
    from c in customers
    join o in orders on c.ID equals o.ID
```

on 关键字在关键字段 ID 的后面，equals 关键字指定另一个集中的对应字段。查询结果

仅包含两个集中 ID 字段值相同的对象数据。

select语句创建了一个带指定属性的新数据类型，因此可以看到最初的总销售量、新订单和最终的新总销售量：

```
select new { c.ID, c.City, SalesBefore = c.Sales, NewOrder = o.Amount,
                                    SalesAfter = c.Sales+o.Amount };
```

这个程序没有在 customer 对象中递增总销售量，但可以在自己的业务逻辑中完成这一任务。

foreach 循环的逻辑和查询中值的显示与本章前面示例中的相同。

26.21　资源和进一步阅读

方法语法中有太多的 LINQ 方法，不可能在入门图书中包含这些方法。更多的细节和示例可参阅 LINQ 的 Microsoft 在线文档。每个 LINQ 方法的小示例可参阅 MSDN 步骤中的“101LINQ Samples”主题(或 http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx)。

Eric White 在 http://blogs.msdn.com/ericwhite/pages/FP-Tutorial.aspx 上提供的有关函数编程的教程是学习 LINQ 中函数编程的一个好资料，该网页还提供了 LINQ 方法语法的完整教程。

26.22　小结

本章学习了如下内容：

LINQ 的不同变体，包括 LINQ to Objects、LINQ to SQL 和 LINQ to XML。

如何给 LINQ to Objects 编写 LINQ 查询，LINQ 查询语句的组成，包括 from、where、select 和 orderby 子句。

如何使用 foreach 语句迭代 LINQ 查询的结果，如何使查询在执行 foreach 语句时才执行。

LINQ 建立在扩展方法的基础上。如何使用方法语法和λ表达式。

如何使用 LINQ 合计运算符获得大型数据集的信息，而无需迭代每个结果。

如何使用投射技术改变数据类型，在查询中创建新对象。

如何使用 Distinct()、Any()、All()、First()、FirstOrDefault()、Take()和 Skip()运算符。

组合查询和多级排序。

集运算符和 join 查询。

LINQ 使 C#中编写的查询非常简单和强大。第 27 章将学习如何使用 LINQ to SQL 查询关系数据库，以高效地处理大型数据集。

26.23　练习

(1) 修改第一个示例程序 26-1-FirstLINQquery，将结果降序排列。

(2) 在大数程序示例 26-5 LargeNumberQuery 中修改传送给 generateLotsOfNumbers()方法的数字，创建不同规模的结果集，看看查询结果所受的影响。

(3) 给大数程序示例 26-5 LargeNumberQuery 中的查询添加一个 orderby 子句，看看这会如何影响性能。

(4) 修改大数程序示例 26-5 LargeNumberQuery 中的查询条件，选择数字列表中的较大和较小子集，看看这会如何影响性能？

(5) 修改方法语法示例 26 子句，这会生成多少个结果？

(6) 修改查询复杂对象程 ，用对应于字段的条件选择查询字段的不同子集。

(7) 给第一个示例程序 2 符，哪些简单的合计运算符能适用于这种非数字的结果集？

第35章

Windows Communication Foundation

第 21 章学习了 Web 服务，以及如何使用它们在应用程序之间提供简单的通信。探讨了如何使用 HTTP GET 和 POST 技术与 Web 服务交换数据，以及如何使用 SOAP。自从 Web 服务可用于.NET 开发人员以来，尽管 Web 服务很强大，但这种技术的扩展显然是有一定限制的。为此，微软公司发布了 Web Service Enhancements(WSE)插件来解决这个问题。WSE 允许 Web 服务开发人员通过消息的安全保护、路由技术和各种其他策略来提高 Web 服务，但提高的幅度仍很有限。

另一种.NET 技术——远程技术可以在一个进程中创建对象实例，在另一个进程中使用它们。远程技术甚至允许在一台计算机上使用位于另一台计算机上的对象。这种技术虽然对以前的技术如 DCOM 而言是一个巨大的进步，但仍有它自己的问题。远程技术是有限制的，刚入门的程序员要掌握它也不容易。

Windows Communication Foundation(WCF)是 Web 服务和远程技术的替代品，它从 Web 服务中提取了服务、独立于平台的 SOAP 消息传输等概念，把它们与远程技术中的主机服务器应用程序、高级绑定功能结合在一起，所以可以将这种技术看作一个超集，包含了 Web 服务和远程技术，但比 Web 服务强大，比远程技术更易于掌握。使用 WCF 可以从简单的应用

程序转向使用面向服务的体系结构(SOA)的应用程序。SOA 表示可以分散处理，并在需要时连接本地网络和 Internet 上的服务和数据，使用分布式处理。

本章将学习 WCF 原理，以及如何在应用程序代码中创建和使用 WCF 服务。主要内容如下：

WCF 是什么
WCF 概念
WCF 编程

提示：

不能在 VCE 中创建 WCF 服务，但可以在 VS 的完全版本中创建。还可以在 Visual Web Developer 2008 Expression Edition 中创建基于 IIS 的 WCF 服务，但本章使用 VS 来介绍所有的选项。

35.1　WCF 是什么

WCF 技术允许创建服务，访问跨进程、机器和网络的其他应用程序。这些服务可以共享多个应用程序中的服务，提供数据源，或者抽象复杂的过程。

与Web 服务一样，WCF 服务提供的功能也封装为该服务的方法。每个方法——在 WCF 术语中称为“操作”——都有一个端点，用于交换数据。在这一点上，WCF 与 Web 服务不同。在 Web 服务中，只能在 HTTP 上通过 SOAP 与端点通信。而在 WCF 服务中，可以选择要使用的协议。端点甚至可以通过多个协议来通信，这取决于通过什么网络连接服务和特定的要求。

在 WCF 上，端点可以有多个绑定，每个绑定都指定了一种通信方式。绑定还可以指定其他信息，例如，必须满足什么安全要求才能与端点通信。例如，绑定可能需要用户名和密码验证或者 Windows 用户账户令牌。在连接一个端点时，绑定使用的协议会影响所使用的地址，如后面所述。

一旦连接了一个端点，就可以使用 SOAP 消息与它通信。所使用的消息形式取决于所进行的操作和该操作收发消息所需的数据结构。WCF 使用合同指定所有这些信息。通过与服务交换的元数据可以查找合同。这类似于 Web 服务使用 WSDL 描述其功能。实际上，可以用 WSDL 格式获得 WCF 服务的信息，但 WCF 服务还可以用其他方式描述。

识别出要使用的服务和端点，知道了要使用的绑定和需要的合同之后，就可以与 WCF 服务通信，这与使用在本地定义的对象一样简单。与 WCF 服务通信可以是简单的单向事务、请求/响应消息，也可以是从通信通道任一端发出的双向通信。还可以在需要时使用消息负载优化技术，如 Message Transmission Optimization Mechanism(MTOM)，打包数据。

WCF 服务在存储它的计算机上运行为许多不同进程中的一个。Web 服务总是运行在 IIS 上，而 WCF 服务可以选择适合的主机进程。可以使用 IIS 运行 WCF 服务，也可以使用 Windows 服务或可执行程序。如果使用 TCP 在本地网络上与 WCF 服务通信，就不需要在运行服务的 PC 上安装 IIS。

WCF 架构允许定制本节介绍的几乎所有方面。但这是一个高级主题，本章仅使用.NET 3.5

默认提供的技术。

了解了 WCF 服务的基础知识后，下面将详细介绍这些概念。

35.2　WCF 概念

本节描述 WCF 的如下方面：

WCF 通信协议

地址、端点和绑定

合同

消息模式

行为

主机

35.2.1　WCF 通信协议

如前所述，可以通过许多传输协议与 WCF 服务通信。在.NET 3.5 Framework 中定义了 4 个协议：

HTTP：它允许与任何地方的 WCF 服务通信，包括 Internet。可以使用 HTTP 通信技术创建 WCF Web 服务。

TCP：如果正确配置了防火墙，它允许与本地网络或 Internet 上的 WCF 服务通信。TCP 比 HTTP 高效，功能也比较多，但配置起来比较复杂。

指定的管道：它允许与 WCF 服务通信，该 WCF 服务必须与调用代码位于同一台机器的不同进程上。

MSMQ：这是一种排队技术，允许应用程序发送的消息通过队列路由到目的地。MSMQ 是一种可靠的消息传输技术，可以确保发送给队列的消息一定达到该队列。MSMQ 还是一种异步技术，所以只有排在前面的消息都处理完了，服务仍有效时，才能处理当前的消息。

这些协议常常允许建立安全连接。例如，可以使用 HTTPS 协议建立 Internet 上的安全 SSL 连接。TCP 使用 Windows 安全架构为本地网络上的安全性能提供了更多的可能性。

图 35-1 中列出了这些传输协议如何把应用程序与不同位置上的 WCF 服务连接起来。本章将介绍所有这些协议，但 MSMQ 除外，这个主题需要较深入的讨论。

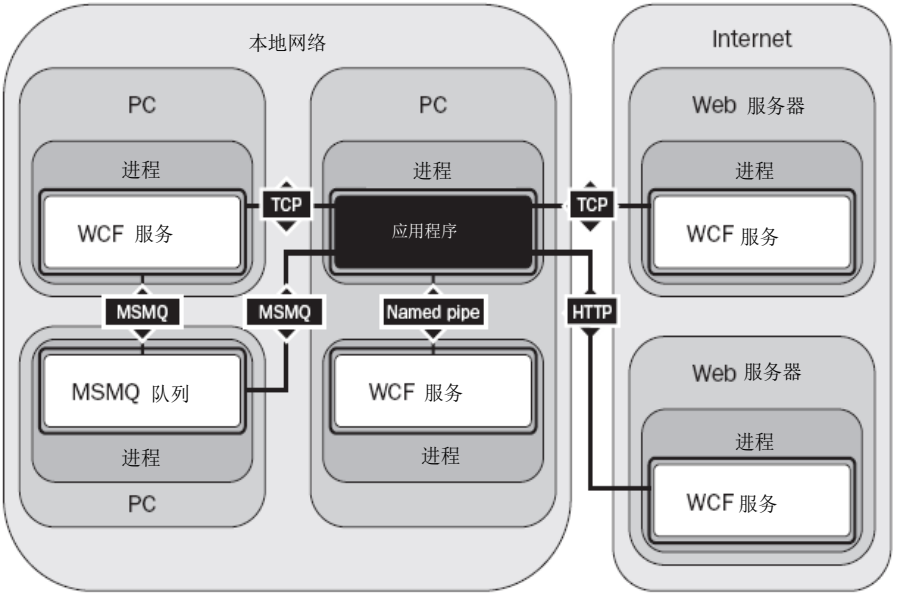

图　35-1

为了连接 WCF 服务，必须知道它在什么地方。这表示必须知道端点的地址。

35.2.2　地址、端点和绑定

用于服务的地址类型取决于所使用的协议。本章前面介绍的 3 个协议都需要格式化的服务地址：

HTTP：HTTP 协议的地址是 URL，其格式很常见：http://<server>:<port>/<service>。对于 SSL 连接，也可以使用 https://<server>:<port>/<service>。如果在 IIS 中存储服务，<service>就是扩展名为.svc 的文件(.svc 文件类似于 Web 服务中使用的.asmx 文件)。IIS 地址可能包含比这个示例更多的子目录，即.svc 文件之前有更多用/字符分隔的部分。

TCP：TCP 的地址采用 net.tcp:// <server>:<port>/<service>形式。

指定的管道：指定的管道连接的地址与上述类似，但没有端口号。其形式是 net.pipe://<server>/<service>。

服务的地址是一个基地址，它可用于为表示操作的端点创建地址。例如，在 net.tcp://<server>:<port>/<service>/operation1 上有一个操作。

假定创建一个 WCF 服务，它有一个操作，绑定了前面介绍的3个协议，我们就可以使用下面的基地址：

```
http://www.mydomain.com/services/amazingservices/mygreatservice.svc
net.tcp://myhugeserver:8080/mygreatservice
net.pipe://localhost/mygreatservice
```

接着就可以给操作使用下面的地址：

```
http://www.mydomain.com/services/amazingservices/mygreatservice.svc/greatop
net.tcp://myhugeserver:8080/mygreatservice/greatop
net.pipe://localhost/mygreatservice/greatop
```

如前所述，绑定不仅指定了操作使用的传输协议，还可以指定在传输协议上通信的安全要求、端点的事务处理功能、消息编码等。

绑定提供了非常大的灵活性，所以.NET Framework 提供了一些可用的预定义绑定。还可以把这些绑定用作起点，修改它们，得到需要的绑定类型。预定义绑定有一些必须遵循的原则。每种绑定类型都用 System.ServiceModel 名称空间中的一个类表示。表 35-1 中列出了这些绑定及其基本信息。

表 35-1

绑 定	说 明
BasicHttpBinding	最简单的 HTTP 绑定，Web 服务使用的默认绑定，它的安全功能有限，不支持事务处理
WSHttpBinding	HTTP 绑定的一种高级形式，可以使用 WSE 中引入的所有额外功能
WSDualHttpBinding	扩展了 WSHttpBinding 功能，包含双向通信功能。在双向通信中，服务器可以启动与客户机的通信，还可以进行一般的消息交换
WSFederationHttpBinding	扩展了 WSHttpBinding 功能，包含联合功能。联合功能允许第三方实现单向广播(single sign-on)和其他专用安全措施。这是一个高级主题，本章不讨论
NetTcpBinding	用于 TCP 通信，允许配置安全性、事务处理等
NetNamedPipeBinding	用于指定管道的通信，允许配置安全性、事务处理等
NetPeerTcpBinding	允许与多个客户机进行广播通信，是一个高级类，本章不讨论
NetMsmqBinding 和 MsmqIntegrationBinding	这些绑定用于 MSMQ，本章不讨论

这个表中的许多绑定类有可用于其他配置的类似属性。例如，它们都有可用于配置超时值的属性。本章后面介绍编码时会详细讨论。

35.2.3 合同

合同定义了 WCF 服务的用法。可以定义如下几种合同：

服务合同：包含服务的一般信息和服务提供的操作。例如，该合同可以包含服务使用的名称空间。在为 SOAP 消息定义模式时，服务使用唯一的名称空间，以避免与其他服务冲突。

操作合同：定义操作的用法，这包括操作方法的参数和返回类型，以及其他信息，例如，方法是否返回响应消息。

消息合同：允许定制 SOAP 消息内部的信息格式化方式。例如，数据应包含在 SOAP 标题中还是 SOAP 消息体中。创建必须与以前的系统集成的 WCF 服务时，就可以使用消息合同。

错误合同：定义操作可能返回的错误。使用.NET 客户程序时，错误会导致可以捕获的异常，并以通常方式处理。

数据合同：如果使用复杂的类型，如用户定义的结构和对象，作为操作的参数或返回类型，就必须为这些类型定义数据合同。数据合同根据通过属性显示的数据定义类型。一般使用特性把合同添加到服务类和方法中，如本章后面所述。

35.2.4　消息模式

上一节提到，操作合同可以定义操作是否返回一个值，WSDualHttpBinding 允许进行双向通信。这些都是消息模式，消息模式有 3 种类型：

请求/响应消息传输：交换消息的“一般”方式，每个发送给服务的消息都会从客户机中得到一个响应。这并不意味着客户机等待响应，因为可以用通常的方式异步调用操作。

单向消息传输：消息从客户机传输给 WCF 操作，但不发送响应。不需要响应时，就可以使用这种消息模式。例如，创建一个 WCF 操作，它会重启 WCF 主机服务器，此时不需要等待响应。

双向消息传输：一种比较高级的模式，客户机可以用作服务器，服务器也可以用作客户机。启动后，双向消息传输允许客户机和服务器彼此发送消息，这些消息可能有响应，也可能没有。这类似于创建一个对象，并订阅该对象提供的事件。

本章的后面将使用这些消息模式。

35.2.5　行为

行为是把没有直接提供给客户机的其他配置应用于服务和操作的方式。给服务添加行为，可以控制如何实例化行为、主机进程如何使用行为，行为如何参与事务处理，在服务中如何解决多线程问题等。操作行为可以控制在操作执行过程中是否使用模仿功能，各个操作行为如何影响事务处理等。

本章仅介绍 WCF 服务的基本知识，讨论行为的最基本功能。

35.2.6　主机

本章的引言曾提到，WCF 服务可以存储在几个不同的进程中，包括：

Web 服务器：基于 IIS 的 WCF 服务是 WCF 提供的最接近 Web 服务的服务。还可以使用 WCF 服务中的高级功能和安全特性，这些功能和特性很难在 Web 服务中实现。也可以集成 IIS 特性，如 IIS 安全特性。

可执行文件：可以把 WCF 服务存储在.NET 中创建的任意应用程序类型中，如控制台应用程序、Windows 窗体应用程序和 WPF 应用程序。

Windows 服务：可以把 WCF 服务存储在 Windows 服务中，这表示可以使用 Windows 服务提供的有用特性，包括自动启动和错误恢复。

Windows Activation Service(WAS)：专门用于存储 WCF 服务，基本上是 IIS 的一个简化版本，可以在任何没有 IIS 的地方使用。

上述列表中的两个选项 IIS 和 WAS 为 WCF 服务提供了有用的特性，例如激活、处理循环和

对象池。如果使用另外两个存储选项，WCF 服务就是自存储的。这不一定是件坏事，因为我们可能不需要主机环境提供的其他功能。但是对于自存储的服务需要编写更多的代码。

35.3　WCF 编程

前面介绍了基础知识，下面开始编写一些代码。本节首先看一个在 Web 服务器上存储的简单 WCF 服务和一个控制台应用程序。介绍了所创建的代码结构后，学习 WCF 服务和客户应用程序的基本结构。之后详细探讨一些重要主题：

定义 WCF 服务合同

自存储的 WCF 服务

试试看：一个简单的 WCF 服务和客户程序

(1) 在目录C:\BegVCSharp\Chapter35 下创建一个新的 WCF 服务应用程序项目 Ch35Ex01

(2) 在解决方案中添加一个控制台应用程序 Ch35Ex01Client。

(3) 在 Build 菜单上单击 Build Solution 选项。

(4) 在 Solution Explorer 中右击 Ch35Ex01Client，选择 Add Service Reference 选项。

(5) 在 Add Service Reference 对话框中，单击 Discover。

(6) 开始开发 Web 服务器，加载 WCF 服务的信息后，展开该引用，查看其细节，如图 35-2 所示(读者的端口号可能与本图中的不同)。

图　35-2

(7) 单击 OK 按钮，添加服务引用。

(8) 在 Ch35Ex01Client 应用程序中修改 Pragram.cs 中的代码，如下所示：

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

```
using System.Text;
using Ch35Ex01Client.ServiceReference1;

namespace Ch35Ex01Client
{
    class Program
    {
        static void Main(string[] args)
        {
            string numericInput = null;
            int intParam;
            do
            {
                Console.WriteLine(
                  "Enter an integer and press enter to call the WCF service.");
                numericInput = Console.ReadLine();
            }
            while (!int.TryParse(numericInput, out intParam));
            Service1Client client = new Service1Client();
            Console.WriteLine(client.GetData(intParam));
            Console.WriteLine("Press an key to exit.");
            Console.ReadKey();
        }
    }
}
```

(9) 在 Solution Explorer 中右击解决方案，选择 Set StartUp Projects 选项。

(10) 把两个项目都选择为启动项目，如图 35-3 所示，单击 OK 按钮。

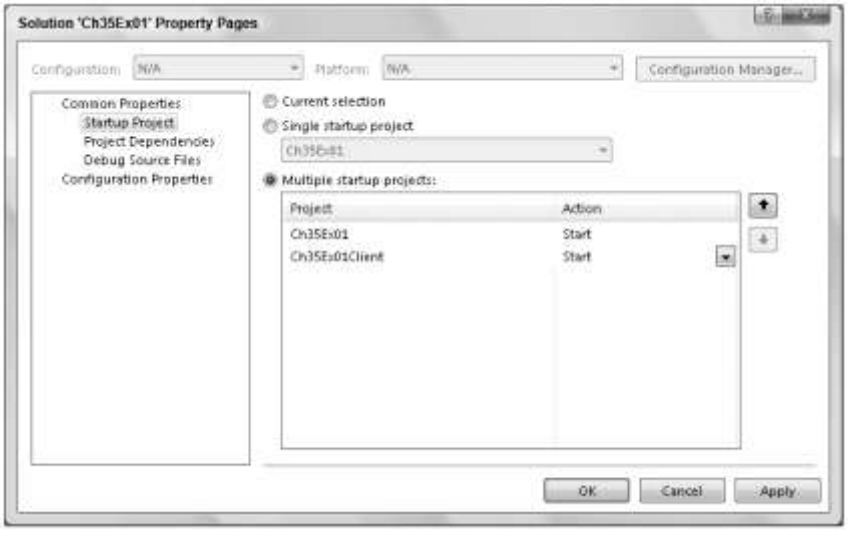

图　35-3

(11) 右击 Ch35Ex01 中的 Service1.svc，单击 Set as StartUp Page。

(12) 运行应用程序。出现提示后，单击 OK 按钮激活 Web.config 中的调试功能。在控制台应用程序窗口中输入一个数字，按下回车键。结果如图 35-4 所示。

(13) 查看窗口中的信息，如图 35-5 所示。

图　35-4

图　35-5

(14) 单击 Web 页面顶部的链接，查看服务的 WSDL。现在还不需要知道 WSDL 文件中的内容。

示例的说明

这个示例中创建了一个存储在 Web 服务器上的简单 Web 服务和控制台客户程序。我们为 WCF 服务项目使用了默认的 VS 模板，这说明不必添加任何代码，而使用这个默认模板中定义的一个操作 GetData()。对于这个示例，使用什么操作并不重要，而应关注代码的结构及其工作方式。

首先看看服务项目 Ch35Ex01，它包含：

Service1.svc 文件，它定义了服务的主机。

类定义 CompositeType，它定义了服务使用的数据合同。

接口定义 IService1，它定义了服务合同和两个操作合同。

类定义 Service1，它执行 IService1 接口，定义了服务的功能。

配置段<system.serviceModel>(在 Web.config 中)，它配置了服务。

Service1.svc 文件包含如下代码（要查看这行代码，应在 Solution Explorer 中右击该文件，再单击 View Markup：

```
< %@ ServiceHost Language="C#" Debug="true" Service="Ch35Ex01.Service1"
CodeBehind="Service1.svc.cs"%>
```

这是一个 ServiceHost 指令，用于告诉 Web 服务器(本例是 Web 开发服务器，尽管这也应用于 IIS)把什么服务存储在这个地址上。定义服务的类在 Service 属性中声明，定义这个类的代码文件在 CodeBehind 属性中声明。这个指令是必须的，以获得 Web 服务器的主机功能，如前面几节所述。

显然，没有存储在 Web 服务器上的 WCF 服务不需要这个文件。本章后面将学习自存储的 WCF 服务。

接着在 IService1.cs 文件中定义数据合同 CompositeType。从代码中可以看出，数据合同只是一个类定义，在类定义中包含了 DataContract 属性，在类成员上包含了 DataMember 属性：

```
[DataContract]
public class CompositeType
{
    bool boolValue=true;
    string stringValue ="Hello";

    [DataMember]
    public bool BoolValue
    {
        get { return boolValue; }
        set { boolValue =value; }
    }

    [DataMember]
    public string StringValue
    {
        get { return stringValue; }
        set { stringValue = value; }
    }
}
```

这个数据合同通过元数据提供给客户应用程序(查看示例中的 WSDL 文件，就会看到这些元数据)。这允许客户应用程序定义一个类型，该类型可以序列化到窗体上，该窗体又可以由服务解序到 CompositeType 对象上。客户程序不需要知道这个类型的定义，实际上，客户程序使用的类可以有不同的执行代码。定义数据合同的这种方式虽简单但非常强大，允许在 WCF 服务及其客户程序之间交换复杂的数据结构。

IService1.cs 文件还包含服务合同，该服务合同定义为带有[ServiceContract]属性的接口。这个接口也在服务的元数据中进行了完整的描述，并可以在客户应用程序中重建。接口成员构建了服务的操作，每个操作都应用 OperationContract 属性创建一个操作合同。示例代码包含两个操作，每个操作都使用了前面的数据合同：

```
[ServiceContract]
```

```
public interface IService1
{
    [OperationContract]
    string GetData(int value);

    [OperationContract]
    CompositeType GetDataUsingDataContract(CompositeType composite);
}
```

前面介绍的 4 个合同定义属性都可以用特性进一步配置，如下一节所述。实现服务的代码与其他类定义类似：

```
public class Service1 : IService1
{
    public string GetData(int value)
    {
        return string.Format("You entered: {0}", value);
    }

    public CompositeType GetDataUsingDataContract(CompositeType composite)
    {
        if (composite.BoolValue)
        {
            composite.StringValue += "Suffix";
        }
        return composite;
    }
}
```

注意这个类定义不需要继承自特定的类型，也不需要任何特定的属性，只需实现定义了服务合同的接口。实际上，可以在这个类及其成员中添加属性，以指定行为，但这些都不是强制的。

把服务的实现代码(类)和服务合同(接口)分开是很好的。客户程序不需要知道类的任何信息，类包含的功能可能远远超过了服务实现的功能。一个类甚至可以实现多个服务合同。

最后看看 Web.config 文件中的配置。在配置文件中，WCF 服务的配置是从.NET 远程技术中提取出来的一个特性，可以处理所有类型的 WCF 服务(非自存储的服务和自存储的服务)和 WCF 服务的客户程序(稍后介绍)。这个配置的语法允许把任何配置应用于服务，甚至可以扩展其语法。

WCF 配置代码包含在 Web.config 或 app.config 文件的配置段<system.serviceModel>中。在这个示例的 Web.config 文件中，配置段包含两个子段：

<services>：定义项目中的服务。每个服务都在一个<services>子段中定义。

<behaviors>：定义<services>段中各个元素使用的行为。在<behaviors>子段中定义的行为可以在多个其他元素中重用。

这个示例只有一个服务。在配置代码中，给服务指定了一个名称，并关联了一个在<behaviors>段中定义的指定行为：

```
<configuration>
  ...
  <system.serviceModel>
```

```
    <services>
      <service name="Ch35Ex01.Service1"
        behaviorConfiguration="Ch35Ex01.Service1Behavior">
```

<service>元素包含两个子元素<endpoint>，每个子元素都定义了服务的一个端点。实际上，这些端点是服务的基端点。操作的端点可以从这些端点中推断出。

端点地址在 address 属性中定义。在 Web 服务器存储的服务中，地址相对于服务的.svc 文件。端点绑定在 binding 属性中定义，绑定的服务合同用接口名指定。第一个端点是服务的主端点，所以使用 IService1 接口作为其服务合同，它还使用默认地址和 WSHttpBinding 绑定类型：

```
<endpoint  address="  "binding="wsHttpBinding"contract="Ch35Ex01.IService1">
  <identity>
    <dns value="localhost"/>
  </identity>
</endpoint>
```

在元素<endpoint>中可以有各种元素，如这里使用的<identity>元素，它指定端点的基服务器地址。这是一个正在开发的服务，所以使用 localhost。

示例服务在 mex 地址上包含第二个端点，mex 是元数据交换(metadata exchange)的缩写，它允许客户程序获得 WCF 服务的描述。WCF 服务与 Web 服务不同，不默认提供服务描述。添加一个使用 IMetadataExchange 合同的端点，就可以获得服务描述。服务描述端点根据所使用的协议，使用 mexHttpBinding、mexHttpsBinding、mexNamedPipeBinding 或 mexTcpBinding 中的一个绑定。

```
      <endpoint address="mex" binding="mexHttpBinding"
        contract="IMetadataExchange"/>
    </service>
  </services>
```

在这个例子中，可以得到 WSDL 描述。这需要把?wsdl 添加到服务地址的后面，实际上这是通过一个行为得到的，不是通过元数据交换端点得到的。这个行为应用于服务，其定义如下：

```
<behaviors>
  <serviceBehaviors>
    <behavior name="Ch35Ex01.Service1Behavior">
      <serviceMetadata httpGetEnabled="true"/>
```

这个行为还为传输到客户机上的错误提供了异常信息，在开发时常常允许这么做：

```
        <serviceDebug includeExceptionDetailInFaults="false"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
</configuration>
```

这就完成了服务器的定义。

在客户应用程序中，我们使用 Add Service Reference 工具添加对服务的引用，使用服务的元数据(即服务的 WSDL)构建代理类。这不是访问 WCF 服务的唯一方式，但它是最简单的方式。另一个常见方式是在一个独立的程序集中为 WCF 服务定义合同，由主机项目和客户项目引用。接着客户程序直接使用这些合同生成代理，而不是通过元数据生成代理。

也可以浏览 Add Service Reference 工具生成的代码(显示项目中的所有文件，包括隐藏的文件)，但目前最好不要浏览代码，因为有许多容易混淆的代码。

这里要注意，工具创建了访问服务所需的所有类，包括服务的代理类和从数据合同中生成的客户端类(CompositeType)，服务的代理类包含服务的所有操作方法(Service1Client)。

该工具还为项目添加了一个配置文件 app.config，这个配置定义了两个内容：

服务端点的绑定信息

端点的地址和合同

绑定信息从服务描述中提取，在客户程序中，每个可配置的选项都被复制到配置文件中：

```
<configuration>
  <system.serviceModel>
    <bindings>
      <wsHttpBinding>
        <binding name="WSHttpBinding IService1" closeTimeout="00:01:00"
         openTimeout="00:01:00"receiveTimeout="00:10:00"sendTimeout=00:01:00"
         bypassProxyOnLocal="false" transactionFlow="false"
         hostNameComparisonMode="StrongWildcard"maxBufferPoolSize="524288"
         maxReceivedMessageSize="65536" messageEncoding="Text"
         textEncoding="utf-8" useDefaultWebProxy="true"allowCookies="false">
        <readerQuotas maxDepth="32" maxStringContentLength="8192"
          maxArrayLength="16384" maxBytesPerRead="4096"
          maxNameTableCharCount="16384"/>
        <reliableSession ordered="true" inactivityTimeout="00:10:00"
          enabled="false"/>
        <security mode="Message">
          <transport clientCredentialType="Windows"proxyCredentialType="None"
            realm=" "/>
          <message clientCredentialType="Windows"
            negotiateServiceCredential="true" algorithmSuite="Default"
            establishSecurityContext="true"/>
        </security>
      </binding>
  </wsHttpBinding>
</bindings>
```

这个绑定、服务的基地址(这是 Web 服务器存储的服务的.svc 文件地址)和合同的客户端版本 IService1 在端点配置中使用：

```
      <client>
        <endpoint address="http://localhost:51173/Service1.svc"
           binding="wsHttpBinding"
           bindingConfiguration="WSHttpBinding IService1"
           contract="ServiceReference1.IService1" name="WSHttpBinding_IService1">
```

```
            <identity>
              <dns value="localhost"/>
            </identity>
          </endpoint>
        </client>
      </system.serviceModel>
    </configuration>
```

Add Service Reference 工具是非常全面的。实际上，大多数信息都不是必要的，因为我们使用的是默认绑定 WSHttpBinding。可以用下面的代码替代这个配置文件：

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address="http://localhost:51173/Service1.svc"
         binding="wsHttpBinding" contract="ServiceReference1.IService1"
         name="WSHttpBinding_IService1">
        <identity>
          <dns value="localhost"/>
        </identity>
      </endpoint>
    </client>
  </system.serviceModel>
</configuration>
```

这段代码删除了<endpoint>元素的 bindingConfiguration 属性，这表示客户程序将使用默认的绑定配置。

但是为了学习 WCF 服务，掌握工具的全面性是非常重要的。它会显示包含在 WSHttpBinding 默认绑定中的所有设置。本章不深入探讨 WCF 服务配置，但介绍了其中的一些配置，如超时设置，这些配置的命名很简单，很容易理解。

这个示例介绍了许多基础知识，下面总结一下前面的内容：

WCF 定义

- 服务由服务合同接口定义，其中包括操作合同成员
- 服务在实现了服务合同接口的类中实现
- 数据合同只是使用数据合同属性的类型定义

WCF 服务配置

- 可以使用配置文件(Web.config 或 app.config)来配置 WCF 服务

WCF Web 服务器主机：

- Web 服务器主机把.svc 文件用作服务基地址

WCF 客户机配置：

- 可以使用配置文件(web.config 或 app.config)来配置 WCF 服务客户机

下面详细介绍合同。

35.3.1　定义 WCF 服务合同

前面的示例说明了 WCF 体系结构如何便于给 WCF 服务定义合同，包括类、接口和属性。本节将深入介绍这种技术。

1. 数据合同

要给服务定义数据合同，需要把 DataContractAttribute 属性应用于类定义。这个属性在名称空间 System.Runtime.Serialization 中。可以使用表 35-2 中所示的属性配置它。

表　35-2

属　性	用　法
Name	用不同于类定义的名称来命名数据合同。这个名称在 SOAP 消息和服务元数据定义的客户端数据对象上使用
Namespace	指定数据合同在 SOAP 消息中使用的名称空间

当需要与已有的SOAP 消息格式交互操作时(类似于其他合同的对应属性)，需要使用这两个属性，否则就不需要它们。

数据合同中的每个类成员都必须使用 DataContractAttribute 属性，它在名称空间 System.Runtime.Serialization 中。这个属性具有表 35-3 中所示的特性。

表　35-3

特　性	用　法
Name	指定序列化时数据成员的名称(默认为成员名称)
IsRequired	指定成员是否必须显示在 SOAP 消息中

(续表)

特　性	用　法
Order	int 值，指定序列化或解序成员的顺序，如果一个成员必须在另一个成员之前出现，这个顺序就是必须的。Order 较低的成员先出现
EmitDefaultValue	把它设置为 false，如果成员的值是默认值，就禁止该成员包含在 SOAP 消息中

2. 服务合同

把System.ServieceModel.ServieceContractAttribute 属性应用于接口定义，就定义了服务合同。表 35-4 中所示的属性可用于定制服务合同。

表　35-4

属　性	用　法
Name	按照 WSDL 中<portType>元素中的定义，指定服务合同的名称
Namespace	定义 WSDL 中<portType>元素使用的服务合同的命称空间

ConfigurationName	在配置文件中使用的服务合同名称
HasProtectionLevel	指定服务使用的消息是否有明确定义的保护级别。保护级别允许签名消息，或者签名和加密消息
ProtectionLevel	保护级别，用于保护消息
SessionMode	确定是否为消息启用会话。如果启用会话，就可以确保关联上发送给服务的不同端点的消息，即它们使用同一个服务实例，因此可以共享状态
CallbackContract	对于双向消息传输，客户机提供了合同和服务。这是因为，如前所述，双向通信中的客户机也用作服务器。这个属性允许指定客户机使用的合同

3. 操作合同

在定义服务合同的接口中，应用 System.ServieceModel.OperationContractAttribute 属性，就可以把成员定义为操作。这个属性具有表 35-5 中所示的特性。

表　35-5

属　性	说　明
Name	指定服务操作的名称。默认为成员名称
IsOneWay	指定操作是否返回一个响应。如果把它设置为 true，则客户机不等待操作完成，就会继续执行
AsyncPattern	设置为true，操作就会执行为两个方法：Begin<methodName>和 End<method Name>，这两个方法可用于异步调用操作
HasProtectionLevel	参见表 35-4
ProtectionLevel	参见表 35-4
IsInitiating	如果使用会话，这个属性就确定调用这个操作是否可以启动新会话

(续表)

属　性	说　明
IsTerminating	如果使用会话，这个属性就确定调用这个操作是否会中断当前会话
Action	如果使用寻址功能(WCF 服务的一个高级功能)，操作就有一个关联的动作名称，通过这个属性可以指定该名称
ReplayAction	同上，但为操作的响应指定动作名称

4. 消息合同

前面的示例中没有使用消息合同规范。如果使用消息合同，就应定义一个表示消息的类，再给类应用 MessageContractAttribute 属性。接着给这个类的成员应用 MessageBodyMemberAttribute、MessageHeaderAttribute 或 MessageHeaderArrayAttribute 属性。所有这些属性都在 System. ServieceModel 名称空间中。如果要高度控制 WCF 服务使用的 SOAP 消息，就不要使用消息合同，所以这里不详细讨论它。

5. 误合同

如果客户应用程序可以使用特定的异常类型，如定制异常，就可以给可能生成该异常的操作应用 System.ServieceModel.FaultContractAttribute 属性。在最初使用 WCF 时不希望这么做。

试试看：WCF 合同

(1) 创建一个新WCF 服务应用程序项目 Ch35Ex02，将其保存在 C: \BegVCSharp \Chapter35 目录下。

(2) 给解决方案添加一个类库项目 Ch35Ex01Contracts，删除 Class1.cs 文件。

(3) 在 Ch35Ex01Contracts 项目中添加对 System.Runtime.Serialization 和 System.Serviece Model.dll 程序集的引用。

(4) 在 Ch35Ex01Contracts 项目中添加 Person 类，修改 Person.cs 中的代码，如下所示：

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

namespace Ch35Ex02Contracts
{
    [DataContract]
    public class Person
    {
        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public int Mark { get; set; }
    }
}
```

(5) 在 Ch35Ex01Contracts 项目中添加 IAwardService 类，修改 IAwardService.cs 中的代码，如下所示：

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

namespace Ch35Ex02Contracts
{
    [ServiceContract(SessionMode=SessionMode.Required)]
    public interface IAwardService
    {
```

```
        [OperationContract(IsOneWay=true,IsInitiating=true)]
        void SetPassMark(int passMark);

        [OperationContract]
        Person[] GetAwardedPeople(Person[] peopleToTest);
    }
}
```

(6) 对于 Ch35Ex01 项目，添加对 Ch35Ex01Contracts 的引用。

(7) 删除 Ch35Ex01 项目中的 IService1.cs 和 Service1.svc。

(8) 在 Ch35Ex01 中添加一个新的 WCF 服务。

(9) 删除 Ch35Ex01 项目中的 IAwardService.cs 文件。

(10) 修改 AwardService.svc.cs 文件中的代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
using Ch35Ex02Contracts;

namespace Ch35Ex02
{
    public class AwardService : IAwardService
    {
        private int passMark;

        public void SetPassMark(int passMark)
        {
            this.passMark = passMark;
        }

        public Person[] GetAwardedPeople(Person[] peopleToTest)
        {
            List < Person > result = new List < Person > ();
            foreach (Person person in peopleToTest)
            {
                if (person.Mark > passMark)
                {
                    result.Add(person);
                }
            }
            return result.ToArray();
        }
    }
}
```

(11) 修改 Web.config 中的服务配置段，如下所示：

```
<system.serviceModel>
  <services>
    <service name="Ch35Ex02.AwardService">
      <endpoint address="" binding="wsHttpBinding"
        contract="Ch35Ex02Contracts.IAwardService"/>
      </service>
  </services>
</system.serviceModel>
```

(12) 将 Ch35Ex02 的启动页面设置为 AwardService.svc。

(13) 在调试模式下运行 Ch35Ex02 项目，记下浏览器中使用的 URL(包括端口号，后面需要使用它)。

(14) 停止调试，在解决方案中添加一个新的控制台项目 Ch35Ex02Client。

(15) 在 Ch35Ex01Client 项目中添加对 System.ServieceModel.dll 程序集和 Ch35Ex01Contracts 的引用。

(16) 在 Ch35Ex01Client 项目中修改 Program.cs 中的代码，如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using Ch35Ex02Contracts;

namespace Ch35E02Client
{
   class Program
   {
      static void Main(string[] args)
      {
         Person[] people = new Person[]
         {
            new Person { Mark = 46, Name="Jim"},
            new Person { Mark = 73, Name="Mike"},
            new Person { Mark = 92, Name="Stefan"},
            new Person { Mark = 84, Name="George"},
            new Person { Mark = 24, Name="Arthur"},
            new Person { Mark = 58, Name="Nigel"}
         };

         Console.WriteLine("People: ");
         OutputPeople(people);

          IAwardService client = ChannelFactory < IAwardService > .CreateChannel(
            new WSHttpBinding(),
             new EndpointAddress("http://localhost:51425/AwardService.svc"));
           client.SetPassMark(70);
           Person[] awardedPeople = client.GetAwardedPeople(people);
```

```
            Console.WriteLine();
            Console.WriteLine("Awarded people: ");
            OutputPeople(awardedPeople);

            Console.ReadKey();
        }

        static void OutputPeople(Person[] people)
        {
            foreach (Person person in people)
            {
                Console.WriteLine("{0}, mark: {1}", person.Name, person.Mark);
            }
        }
    }
}
```

(17) 运行应用程序，结果如图 35-6 所示。

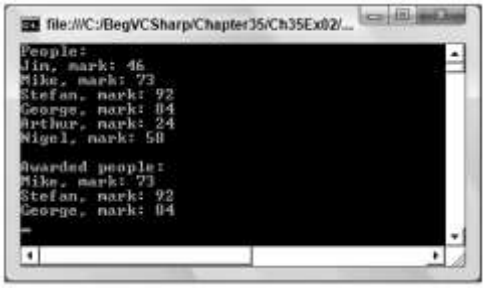

图　35-6

示例的说明

这个示例在类库项目中创建了一系列合同，在 WCF 服务和客户程序中使用了这个类库。与前面的示例一样，这个服务也存储在 Web 服务器上。这个服务的配置也被减少到最低程度。

在这个示例中，主要区别是客户程序不需要元数据，因为客户程序可以访问合同程序集。客户程序不是从元数据中生成一个代理类，而是通过另一种方法获得服务合同接口的引用。这个示例中另一个值得注意的地方是使用会话维护服务中的状态。

这个示例使用的数据合同是一个简单的类 Person，它有一个 string 属性 Name 和一个 int 属性 Mark。使用的 DataContractAttribute 属性和 DataMemberAttribute 属性没有进行定制，也不需要给这个合同重复迭代代码。

定义服务合同时，给 IAwardService 接口应用 ServiceContractAttribute 属性。这个属性的 SessionMode 特性设置为 SessionMode.Required，因为这个服务需要状态：

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IAwardService
{
```

第一个操作合同 SetPassMark()设置状态，因此其 OperationContractAttribute 属性的

IsInitiating 特性设置为 true。这个操作不返回任何值，所以将 IsOneWay 设置为 true，把操作定义为单向操作：

```
[OperationContract(IsOneWay=true,IsInitiating=true)]
void SetPassMark(int passMark);
```

另一个操作合同 GetAwardedPeople()不需要任何定制，使用前面定义的数据合同：

```
    [OperationContract]
    Person[] GetAwardedPeople(Person[] peopleToTest);
}
```

这两个类型 Person 和 IAwardService 都可以用于服务和客户程序。服务在 AwardService 类型中实现了 IAwardService 合同，它不包含任何可标记的代码。这个类与前面的服务类的唯一区别是，这个类是有状态的。这是允许的，因为定义了一个会话，来关联来自客户程序的消息。

客户程序比较有趣，主要是因为下面这行代码：

```
IAwardService client = ChannelFactory < IAwardService > .CreateChannel(
    new WSHttpBinding(),
    new EndpointAddress("http://localhost:51425/AwardService.svc"));
```

客户程序没有用 app.config 文件配置来与服务的通信，也没有从元数据中定义代理类，来与服务通信。而是通过 ChannelFactory<T>.CreateChannel()方法创建代理类。这个方法创建了一个执行 IAwardService 客户程序的代理类，但在后台生成的类与服务通信，就像前面通过元数据生成的代理一样。

提示：

如果以这种方式创建代理类，通信通道就默认为在 1 分钟后超时，导致通信错误。使连接一直处于激活状态有许多方式，但这些都超出了本章的讨论范围。

以这种方式创建代理类是一种非常有用的技术，可以快速生成客户应用程序。

35.3.2 自存储的 WCF 服务

本章前面介绍了存储在 Web 服务器上的 WCF 服务。它们可以在 Internet 上通信，但对于本地网络通信而言，这并不是最高效的方式。一方面，需要用计算机上的 Web 服务器存储服务，另一方面，在应用程序的体系结构上出现一个独立的 WCF 服务可能并不合适。

因此应使用自存储的 WCF 服务。自存储的 WCF 服务存在于创建它的进程中，而不存在于特别建立的主机应用程序(如 Web 服务器)的进程中。这样，就可以使用控制台应用程序或 Windows 应用程序存储服务了。

要建立自存储的 WCF 服务，需要使用 System.ServieceModel.ServieceHost 类。用要存储的服务类型或服务类的一个实例来实例化这个类。通过属性或方法可以配置服务主机，也可以通过配置文件来配置。实际上，主机进程(如 Web 服务器)使用 ServieceHost 实例完成该存储任务。自存储时，区别是直接与这个类交互操作。但是，在主机应用程序的 app.config 文

件中，<system.servieceModel>段中的配置使用的语法与本章前面的配置段中的相同。

可以通过任意协议提供自存储的WCF服务，但是一般在这种类型的应用程序中使用TCP或指定管道绑定。通过 HTTP 访问的服务常常位于 Web 服务器进程中，因为可以获得 Web 服务器提供的额外功能，如安全性等。

如果要存储 MyService 服务，可以使用下面的代码创建 ServieceHost 的一个实例：

```
ServiceHost host = new ServiceHost(typeof(MyService));
```

如果要存储 MyService 的实例 MyServiceObject，可以编写如下代码，创建 ServieceHost 的一个实例：

```
MyService myServiceObject = new MyService();
ServiceHost host = new ServiceHost(myServiceObject);
```

注意，只有配置了服务，使调用总是可以路由到同一个对象实例上，才能使用后一种技术。为此，必须给服务类应用 ServieceBehaviorAttribute 属性，将这个属性的 InstanceContextMode 特性设置为 InstanceContextMode.Single。

创建了 ServieceHost 实例后，就可以通过属性配置服务、其端点和绑定。另外，如果把配置放在.config 文件中，ServieceHost 实例就会自动配置。

有了配置好的ServieceHost 实例后，为了开始存储服务，使用 ServieceHost.Open()方法。同样，通过 ServieceHost.Close()方法可以停止存储服务。第一次存储 TCP 绑定的服务时，如果启用它，可能会收到 Windows 防火墙服务发出的一个警告，因为它阻塞了默认的 TCP 端口。必须给这个服务打开 TCP 端口，才能开始监听该端口。

下面的示例使用自存储技术通过 WCF 服务提供 WPF 应用程序的一些功能。

试试看：自存储的 WCF 服务

(1) 创建一个新的 WPF 应用程序 Ch35Ex03，将其保存在 C:\Beg VCSharp\Chapter35 目录下。

(2) 使用 Add New Item 向导给项目添加一个新的 WCF 服务 AppControlService。

(3) 修改 Window1.xaml 中的代码，如下所示：

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Ch35Ex03.Window1"
  Title="Solar Evolution" Height="450" Width="430"Loaded="Window_Loaded"
  Closing="Window_Closing">
  <Grid Height="400" Width="400" HorizontalAlignment="Center"
    VerticalAlignment="Center">
   <Rectangle Fill="Black" RadiusX="20" RadiusY="20" StrokeThickness="10">
    <Rectangle.Stroke >
      <LinearGradientBrush EndPoint="0.358,0.02" StartPoint="0.642,0.98">
        <GradientStop Color="#FF121A5D" Offset="0"/>
          <GradientStop Color="#FFB1B9FF" Offset="1"/>
        </LinearGradientBrush >
```

```
        </Rectangle.Stroke >
      </Rectangle >
    <Ellipse Name="AnimatableEllipse" Stroke="{x:Null}" Height="0"
      Width="0"HorizontalAlignment="Center" VerticalAlignment="Center">
      <Ellipse.Fill >
        <RadialGradientBrush >
          <GradientStop Color="#FFFFFFFF" Offset="0"/>
          <GradientStop Color="#FFFFFFFF" Offset="1"/>
        </RadialGradientBrush>
      </Ellipse.Fill>
    <Ellipse.BitmapEffect>
    <OuterGlowBitmapEffect GlowColor="#FFFFFFFF" GlowSize="16"/>
        </Ellipse.BitmapEffect>
      </Ellipse>
    </Grid>
  </Window>
```

(4) 修改 Window1.xaml.cs 中的代码，如下所示：

```
...
using System.Windows.Shapes;
using System.ServiceModel;
using System.Windows.Media.Animation;

namespace Ch35Ex03
{
    /// < summary >
    /// Interaction logic for Window1.xaml
    /// < /summary >
    public partial class Window1 : Window
    {
        private AppControlService service;
        private ServiceHost host;

        public Window1()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            service = new AppControlService(this);
            host = new ServiceHost(service);
            host.Open();
        }

        private void Window_Closing(object sender,
            System.ComponentModel.CancelEventArgs e)
        {
            host.Close();
```

```
        }

        internal void SetRadius(double radius, string foreTo, TimeSpan duration)
        {
            if (radius > 200)
            {
                radius = 200;
            }
            Color foreToColor = Colors.Red;
            try
            {
                foreToColor = (Color)ColorConverter.ConvertFromString(foreTo);
            }
            catch
            {
                // Ignore color conversion failure.
            }
            Duration animationLength = new Duration(duration);

            DoubleAnimation radiusAnimation = new DoubleAnimation(
                radius * 2, animationLength);
            ColorAnimation colorAnimation = new ColorAnimation(
                foreToColor, animationLength);
           AnimatableEllipse.BeginAnimation(Ellipse.HeightProperty,
radiusAnimation);
            AnimatableEllipse.BeginAnimation(Ellipse.WidthProperty,
radiusAnimation);
            ((RadialGradientBrush)AnimatableEllipse.Fill).GradientStops[1]
              .BeginAnimation(GradientStop.ColorProperty, colorAnimation);
        }
    }
}
```

(5) 修改 IAppControlService.cs 中的代码，如下所示：

```
[ServiceContract]
public interface IAppControlService
{
    [OperationContract]
    void SetRadius(int radius, string foreTo, int seconds);
}
```

(6) 修改 AppControlService.cs 中的代码，如下所示：

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class AppControlService : IAppControlService
{
    private Window1 hostApp;

    public AppControlService(Window1 hostApp)
    {
```

```
            this.hostApp = hostApp;
        }

        public void SetRadius(int radius, string foreTo, int seconds)
        {
            hostApp.SetRadius(radius, foreTo, new TimeSpan(0, 0, seconds));
        }
    }
```

(7) 修改 app.config 中的代码，如下所示：

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Ch35Ex03.AppControlService">
        <endpoint address="net.tcp://localhost:8081/AppControlService"
          binding="netTcpBinding" contract="Ch35Ex03.IAppControlService"/ >
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

(8) 在项目中添加一个新的控制台应用程序 Ch35Ex03Client。

(9) 配置解决方案，使之有多个启动项目，让两个项目同时启动。

(10) 在 Ch35Ex03Client 项目中添加对 System.ServieceModel.dll 和 Ch35Ex03 的引用。

(11) 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 Ch35Ex03;
using System.ServiceModel;

namespace Ch35Ex03Client
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Press enter to begin. ");
            Console.ReadLine();
            Console.WriteLine("Opening channel...");
            IAppControlService client =
               ChannelFactory < IAppControlService > .CreateChannel(
                  new NetTcpBinding(),
                  newEndpointAddress("net.tcp://localhost:8081/AppControlService"))
;
```

```
                Console.WriteLine("Creating sun...");
                client.SetRadius(100,"yellow",3);
                Console.WriteLine("Press enter to continue.");
                Console.ReadLine();
                Console.WriteLine("Growing sun to red giant...");
                client.SetRadius(200, “Red”, 5);
                Console.WriteLine("Press enter to continue.");
                Console.ReadLine();
                Console.WriteLine("Collapsing sun to neutron star...");
                client.SetRadius(50, "AliceBlue", 2);
                Console.WriteLine("Finished. Press enter to exit.");
                Console.ReadLine();
            }
        }
    }
```

(12) 运行解决方案。出现提示时，打开 Windows 防火墙 TCP 端口，使 WCF 可以监听连接。

(13) 显示 Solar Evolution 窗口和控制台应用程序窗口时，在控制台窗口中按下回车键。结果如图 35-7 所示。

(14) 在控制台窗口中继续按下回车键，继续太阳演化循环。

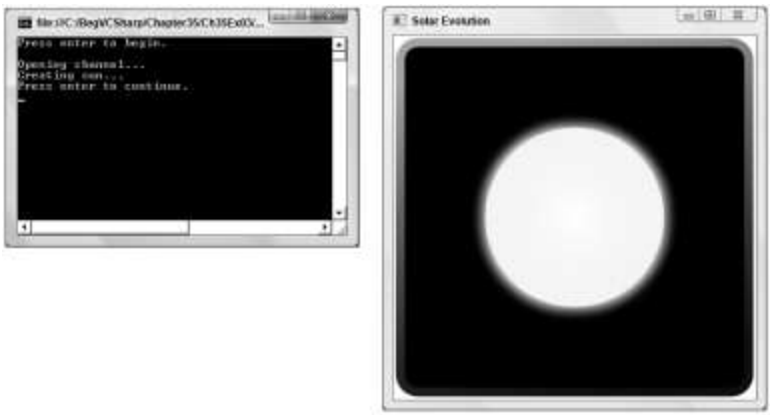

图　35-7

示例的说明

这个示例在 WPF 应用程序中添加了一个 WCF 服务，用它控制 Ellipse 控件的动画。我们创建了一个简单的客户应用程序来测试服务。如果不熟悉WPF，不必过多地考虑示例中的 XAML 代码，我们只对 WCF 感兴趣。

WCF 服务 AppControlService 有一个操作 SetRadius()，客户程序调用这个操作来控制动画。这个方法与和它同名的方法通信，这个方法在 WPF 应用程序中为 Window1 类定义。为此，服务必须引用应用程序，所以必须存储该服务的一个对象实例。如前所述，这表示服务必须使用行为属性：

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
```

```
public class AppControlService : IAppControlService
{
  ...
}
```

在 Window1.xaml.cs 中，服务实例在 Windows_Loaded()事件处理程序中创建。这个方法也为服务创建了一个 ServieceHost 对象，并调用了其 Open()方法，启动存储任务：

```
public partial class Window1 : Window
{
   private AppControlService service;
   private ServiceHost host;
   ...

   private void Window_Loaded(object sender, RoutedEventArgs e)
   {
      service = new AppControlService(this);
      host = new ServiceHost(service);
      host.Open();
   }
```

在 Window_Closing()事件处理程序中，应用程序关闭时，存储过程中断。

配置文件非常简单，它定义了 WCF 服务的一个端点，监听端口 8081 的 net.tcp 地址，使用默认的 NetTcpBinding 绑定：

```
<service name="Ch35Ex03.AppControlService">
  <endpoint address="net.tcp://localhost:8081/AppControlService"
    binding="netTcpBinding" contract="Ch35Ex03.IAppControlService"/>
</service>
```

这与客户应用程序中的代码相匹配，如前面的示例所示(但这里使用了 TCP 绑定)：

```
IAppControlService client =
  ChannelFactory < IAppControlService > .CreateChannel(
     new NetTcpBinding(),
     new EndpointAddress("net.tcp://localhost:8081/AppControlService"));
```

客户程序创建客户代理类时，可以调用 SetRadius()方法，给它传递半径、颜色和动画持续时间等参数，这些都会通过服务传递给 WPF 应用程序。接着，WPF 应用程序中的简单代码定义并使用动画，改变椭圆的大小和颜色。

如果使用的机器名不是 localhost，网络允许在指定的端口上通信，这段代码就可以在网络上工作。另外，还可以把客户程序和主机应用程序分开得远一些，并通过 Internet 连接起来。无论采用什么方式，WCF 服务都提供了很好的通信方式，启动它不需要费太大的劲。

35.4　小结

本章介绍了使用 WCF 在应用程序、进程和计算机之间通信的基本技术。首先学习了 WCF 服务的概念，它与 Web 服务或远程技术的区别，使用 WCF 服务需要了解的概念。接着讨论了如何编写 WCF 服务，如何在客户程序中使用 WCF 服务，如何以各种方式存储 WCF 服务。

本章的主要内容如下：

WCF 服务的概念

WCF 服务如何配置地址、端点、绑定、合同和行为

存储 WCF 服务的各种方式

创建 WCF 服务需要的类、属性、接口和配置

如何通过 WCF 元数据为客户应用程序创建代理类

如何定义和使用数据、服务和操作合同

如何从 WCF 合同中生成代理客户类

如何自存储 WCF 服务

这是在应用程序中使用 WCF 服务所必需的最少知识。本章仅涉及 WCF 服务的皮毛，尤其是.config 文件配置和行为。WCF 架构允许集成高级的安全体系，以任意方式定制通信。

如果希望学习 WCF 服务的更多知识，可以阅读 Scott Klein 编著的 *Professional* WCF *Programming*(Wrox，2007)。第 36 章将介绍.NET 3.5 中的最后一种新技术：Workflow Foundation。

35.5　练习

(1) 下述哪些应用程序可以存储 WCF 服务？

a. Web 应用程序

b. Windows 应用程序

c. Windows 服务

d. COM+应用程序

e. 控制台应用程序

(2) 如果要与 WCF 服务交换 MyClass 类型的参数，应执行什么类型的合同？需要什么属性？

(3) 如果把 WCF 服务存储在 Web 应用程序中，应对服务使用的基端点进行什么扩展？

(4) 在自存储 WCF 服务时，必须设置 ServiceHost 类的属性，调用它的方法，来配置服务。对吗？

(5) 提供服务合同 IMusicPlayer 的代码，它定义了 Play()、Stop()和 GettrackInformation()操作。在合适的地方使用单向方法。还要为这个服务定义的其他合同是什么？